VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

POKRYTELNOST PRO PARALELNÍ PROGRAMY

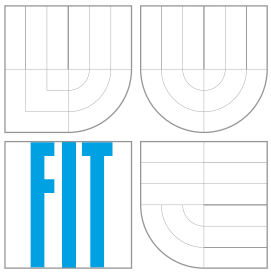
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LENKA TUROŇOVÁ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

POKRYTELNOST PRO PARALELNÍ PROGRAMY

COVERABILITY FOR PARALLEL PROGRAMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LENKA TUROŇOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. LUKÁŠ HOLÍK, Ph.D.

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá automatickou verifikací systémů s paralelně běžícími procesy. Práce diskutuje existující metody a možnosti jejich optimalizace. Stávající techniky jsou založeny na hledání induktivního invariantu (například pomocí techniky zjemňování abstrakce řízené protipříklady (CEGAR)). Efektivnost metod závisí na velikosti nalezeného invariantu. V rámci této diplomové práce jsme nalezli možnost zlepšení metod díky zaměření se na hledání invariantů minimální velikosti. Naimplementovali jsme nástroj, který zajišťuje prohledávání prostoru invariantů systému. Naše experimentální výsledky ukazují, že mnoho existujících systémů užívaných v praxi má skutečně mnohem menší invarianty než ty, které lze nalézt stávajícími metodami. Závěry a výsledky této práce budou sloužit jako základ budoucího výzkumu, jehož cílem bude navržení optimální metody pro vypočítání malých invariantů paralelních systémů.

Abstract

This work is focusing on automatic verification of systems with parallel running processes. We discuss the existing methods and certain possibilities of optimizing them. Existing techniques are essentially based on finding an inductive invariant (for instance using a variant of counterexample-guided abstract refinement (CEGAR)). The effectiveness of these methods depends on the size of the invariant. In this thesis, we explored the possibility of improving the methods by focusing on finding invariants of minimal size. We implemented a tool that facilitates exploring the space of invariants of the system under scrutiny. Our experimental results show that many practical existing systems indeed have invariants that are much smaller than what can be found by the existing methods. The conjectures and the results of the work will serve as a basis of future research of an efficient method for finding small invariants of parallel systems.

Klíčová slova

Paralelní systémy, formální verifikace, Petriho sítě, pokrytelnost, abstrakce, dobře kvasi-uspořádané přechodové systémy.

Keywords

Parallel systems, formal verification, Petri nets, coverability, abstraction, well-quasi-ordered transition systems.

Citace

Lenka Turoňová: Pokrytelnost pro paralelní programy, diplomová práce, Brno, FIT VUT v Brně, 2015

Pokrytelnost pro paralelní programy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Mgr. Lukáše Holíka, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Lenka Turoňová
27. května 2015

Poděkování

Na tomto místě bych ráda poděkovala panu Mgr. Lukáši Holíkovi, Ph.D., za vedení a poskytnuté podněty a rady při přípravě této diplomové práce a dále také Prof. Ing. Tomáši Vojnarovi, Ph.D., za poskytnuté konzultace.

Obsah

Kapitola 1

Introduction

The pace of the entire world is speeding up. World population grows and thereby hungers for new and faster machines, connections and technologies in general that would bring more comfort, spare money and especially time. Time is actually the most valuable commodity. You cannot buy it and therefore you should use every single second. One of the approach how to reduce the amount of time is parallelizing of tasks.

Let us look at the every day's problem. Consider a situation where you want to build a house. Would you hire only a single worker to build the entire house? No. You would hire a few workers that could be able to work simultaneously and thus the house would be built in less time. And this is the same for the world of technologies.

The idea of parallelization of tasks was first discussed in the year 1958 by Stanley Gill who presented in his article *Parallel Programming* [16] the idea of controlling two or more operations which are executed virtually simultaneously. It led to the development of multiple-core processors and the parallel programming software.

On one hand, the systems are constantly getting more complex, powerful and faster. On the other, the software engineers have to deal with higher possibility of an error occurrence associated with simultaneous data access and modification requests which are extremely difficult tasks in the case of parallel systems. However, it is an important step in the software development. The developers spend excessive amount of time and money verifying their codes by testing debugging. According to recent Cambridge University research, the global cost of debugging software has risen to $312 billion per year. Hence any technique that can automatically check the correctness of these systems is appreciated.

In some cases, a small error can lead to a massive problems. A bright example in history could be considered the Therac-25 medical radiation therapy device where a side effect of the buggy software powering the device was the overdose of radiation that several patients received [12].

1.1 Approaches to verifications

One of the approaches proving that system works correctly is formal verification. Formal verification denotes methods based on formal, mathematical roots and (at least potentially) capable of proving error freeness of a system with respect to correctness specification. One of the technique of formal verification is systematically searching the state space of the system. The aim of formal verification is to check whether a given concurrent program is deadlock-free, or how many elements can appear in a buffer. Formal verification is expected

to be fully automated (no human help needed), sound (if it claims that a system is correct wrt. a given specification, it is indeed correct), complete (if it announces that a system is not correct, there is indeed an error in the system i.e., no false alarms (false positives) are possible) and always terminate [17]. The main disadvantage of formal verification is that it is not suitable for large-scale systems unless a high level of abstraction is used. The most common approaches used in computer-aided formal analysis and verification are the following.

Theorem proving is usually a semi-automated approach using some inference system for deducing theorems about the examined system from the facts known about the system and from general theorems and axioms of various logical theories.

Model checking is an approach of automated checking whether a system (or its model) satisfy a certain correctness specification based on a systematic exploration of the state space of the system. Model checking is sound, complete and automatic but its basic variant requires the closed system i.e., the system together with its environment and the bounded states space.

Static analysis is usually characterized as the analysis that collects some information about the behavior of a system based on rather syntactical criteria without regard to its original semantics [4].

There is also testing which is widely used for checking correctness of systems, but its disadvantages are that it cannot be used to prove the system is safe, and is not suitable for covering corner cases. The systems with a bounded number of processes has a problem that there are many variants of interleaving of instructions among processes and that an error can occur only in the certain interleaving that may not be checked during testing.

1.2 Formal verification of parallel programs

The system with parallel running processes even if a state space of the single process is small can generate a huge number of interleavings. This case is called a state explosion (the size of the state space is exponentially to the number of the processes). The state explosion problem is a demanding technical challenge. On top of that, some systems has the number of parallel running processes described by a parameter that can be unbounded and thus, their state space is infinite. So it is even more important to use a technique that reduces the state space.

A number of state reduction approaches have been proposed. Among these techniques, abstraction is considered the most general and flexible for handling the state explosion problem. Generally, abstraction interpretation is a special case of static analysis where a program is run within an abstract domain. The abstraction is used in the most recent approaches of model checking as well and thus the line between these two fields is not so clear.

Intuitively, abstraction amounts to removing or simplifying details as well as removing entire components of the original design that are irrelevant to the property under consideration. The evident rationale is that verifying the simplified ("abstract") model is more efficient than verifying the original one. However, the information loss incurred by simplifying the model has a price: verifying an abstract model potentially leads to wrong results [5].

The technique called *counterexample-guided abstraction refinement* (CEGAR) is a technique that iteratively refines an abstract model using counterexamples. A counterexample

is a witness of a property violation. In software verification, the counterexamples are error paths, i.e., paths through the program that violate the property [15].

We focus on parallel systems with infinite many threads that have infinite state space. This area has attracted the attention of a large number of people from various research communities. Due to the unbounded number of processes, the systems have infinite number of possible configurations. There is a relation on configurations which is monotonic and the system of configurations is a well quasi ordered system (*WSTS*). These two properties help to decide the coverability problem of incorrect configurations for parallel programs.

The *WSTS* include Petri nets, broadcast protocols, and lossy channel systems. Petri nets offer a mathematical concept for modeling such systems and are used in a process of verification of the correctness of programs. They help practitioners to make their models more methodical, and theoreticians to make their models more realistic. Using Petri net we can simulate the behavior of the system and check analytically their properties concerning safety, coverability or liveness. Moreover, they offer a possibility to describe systems in a graphical way. Petri net abstracts away from the time consumption of any action and the data dependencies among conflict decisions.

We discuss the existing research approaches in the field of verification parallel programs, especially we focus on the coverability property of Petri nets. The existing algorithms inspect state space in order to find a safe inductive invariant that would be used for proving correctness of systems. The aim of this work is the proof of existence of smaller invariants than those that have been already found by the existing methods since the smaller invariant can speed up the overall verification process.

1.3 Thesis outline

The work is divided into several chapters. Chapter 2 explains a notion of well-quasi-ordered transition system. The chapter also presents a general scheme for checking the coverability problem. Chapter 3 presents an introduction to Petri nets and thread transition systems. In chapter 4, the existing algorithms for coverability checking are discussed, namely the algorithms presented in articles [9], [14] and [3]. Chapter 5 describes a representation of the invariant and opportunities for improvement of existing approaches. In chapter 6, the algorithm of the abstract forward run and the backward analysis is introduced. The implementation of the algorithm is described in chapter 7. The chapter 8 is presented the experimental results. The conjectures and the results are summarized in chapter 9.

Kapitola 2

Well-quasi-ordered Transition Systems

The well-quasi-ordered transition systems, *WSTS* for short, are systems with infinitely many states with a well-quasi-ordering (*wqo*), and whose transitions satisfy a monotonicity property. A general decidability result shows that the coverability problem (reachability in an upward-closed set) is decidable for *WSTS*.

Formally, according to [9], *WSTS* is a tuple $(\Sigma, I, \rightarrow, \succeq)$ where Σ stands for a set of states, a finite set $I \subset \Sigma$ is a set of initial states, $\rightarrow \subset \Sigma \times \Sigma$ is a transition on Σ, and a $\succeq \subset \Sigma \times \Sigma$ is a well-quasi ordering which satisfies the two following properties.

The transition relation $\rightarrow$ between states is *monotonic* wrt. to the relation $\succeq$ if for each two configurations s_0 and s_1 such that $s_0 \succeq s_1$ and a relation $s_0 \rightarrow s_0'$ there is a configuration s_1' such that $s_1 \rightarrow s_1'$ (see Figure 2.1).

The pre-order $\succeq$ is defined over a set S such that for any infinite sequence $s_0, s_1, s_1, \ldots$, there are i, j with $i < j$ and $s_i \succeq s_j$. If $\succeq$ is an equivalence relation, then the condition of $\succeq$ being a *wqo* amounts to the equivalence relation having a finite index [11].

Given a set of states S and a pre-order $\succeq$ defined over a set S and let $T \subseteq S$ be a set of states its upward-closure $T\uparrow$ is defined as a set:

$$T\uparrow \stackrel{\text{def}}{=} \{s \in S \mid \exists t \in T : t \succeq s\}. \tag{2.1}$$

While a downward-closure of T is defined as:

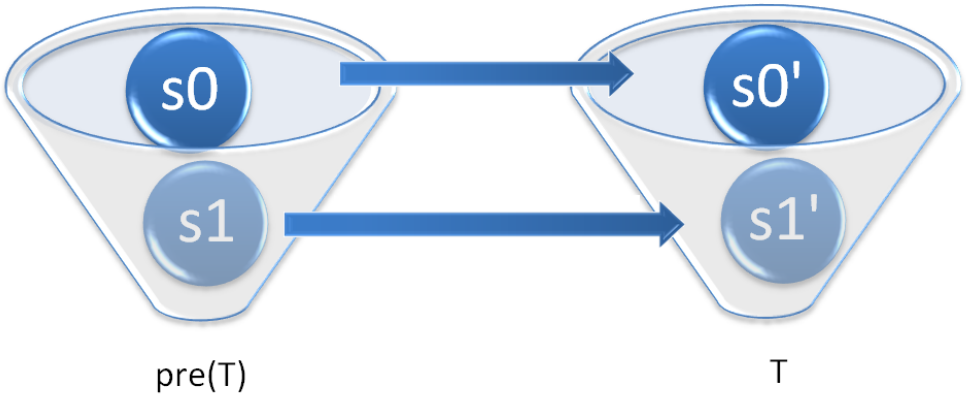
$$T\downarrow \stackrel{\text{def}}{=} \{s \in S \mid \exists t \in T : s \succeq t\}. \tag{2.2}$$

We define a set T to be an upward closed set (*UCS*), respectively a downward closed set (*DCS*), iff $T\downarrow = T$, respectively $T\uparrow = T$. If T is an *UCS* its complement $S\backslash T$ is a *DCS*, and, conversely, if T is a *DCS*, its complement is an *UCS* [1].

Based on the monotocity of $\rightarrow$, for any *UCS*, the set of its predecessors is an *UCS*. According to [11], if the system is *wqo*, then it can be proved that the reachability of an *UCS* of configurations (wrt. $\succeq$) can be checked automatically. Every *UCS* U can be characterized by its *finite* set of minimal elements $min(U)$ consisting of states that are pairwise incomparable. Starting with the empty set U, the iterative computation of the reachable configurations from U eventually terminates since only a finite number of steps are necessary to capture all minimal elements $min(U)$.

Let x, $y \in \Sigma$. If $x \rightarrow y$ we call x a *predecessor* of y and y a *successor* of x. We write

$$pre(x) := \{y \mid y \rightarrow x\} \tag{2.3}$$



Obrázek 2.1: Monotonicity, pre(T), and upward closedness

for the *set of predecessors* of x, and

$$post(x) := \{y \mid x \to y\} \tag{2.4}$$

for the *set of successors* of x. For $X \subset \Sigma$, $pre(X)$ and $post(X)$ are defined as natural extensions, i.e.

$$pre(X) = \bigcup_{x \in X} pre(x) \tag{2.5}$$

and

$$post(X) = \bigcup_{x \in X} post(x). \tag{2.6}$$

According to [11], we write $x{\to}y$ to denote that $(x, y) \in \to$. For sets X and Y of states, we use $X{\to}Y$ to denote that there are $x \in X$ and $y \in Y$ such that $x \to y$. If there are states $x_0, \ldots, x_k \in \Sigma$ such that $x_0 = x, x_k = y$ and $x_i{\to}x_{k+1}$ for $0 \leq i < k$, then we write $x \xrightarrow{k} y$.

Furtheremore, $\xrightarrow{*}$ represents the reflexive transitive closure of $\to$. A set X of states is said to be reachable if $X_{init} \xrightarrow{*} X$. The set of *k-reachable* states, reachable in at most k steps, is defined as:

$$Reach_k := \{y \in \Sigma \mid \exists k' \leq k, \exists x \in I, x \xrightarrow{k'} y\}. \tag{2.7}$$

Additionally, a set of reachable states is formally defined as:

$$Reach := \bigcup_{k \geq 0} Reach_k = \{y \in \Sigma \mid \exists x \in I, x \xrightarrow{k} y\}. \tag{2.8}$$

Given a $WSTS$ $S = (\Sigma, I, \to, \succeq)$, we denote by $Cover$ the covering set of S consisting of all states covered by some of the reachable states as:

$$Cover(S) \stackrel{\text{def}}{=} post^*(I)\downarrow. \tag{2.9}$$

Using DCS, we can define the *k-th cover* $Cover_k$ and the *cover* $Cover$ of the $WSTS$ as $Cover_k := Reach_k\downarrow$ and $Cover := Reach\downarrow$.

2.1 Coverability Problem

Let us given a $WSTS$ $S_0 = (\Sigma, I, \rightarrow, \succeq)$ and a finite set of minimal incorrect configurations $minBad$. The coverability problem means to decide whether a configuration from the set bad, $bad = minBad\uparrow$, is reachable from the set of initial states I or not. A system where bad is not reachable is called safe. Formally, we say that the set $minBad$ is coverable if $Cover(S_0) \cap bad \neq \emptyset$.

According to [11], the coverability problem can be solved for $WSTS$ by Algorithm 1. The algorithm can effectively compute the predecessors of an UCS since any UCS can be represented by its finite set of minimal elements (due to wqo of $\succeq$), the monotonicity of $\rightarrow$ wrt. $\succeq$ applies that the set of predecessors of any UCS is an UCS, and the algorithm can be computed symbolically on the representatives using:

$$minpre(T) \stackrel{\text{def}}{=} min((pre(T\uparrow))\uparrow) \tag{2.10}$$

.

Algorithm 1 *Backward Reachability*

Input:

- $S = (\Sigma, I, \rightarrow, \succeq)$: $WSTS$
- bad: UCS of configurations.

Output:
Is bad reachable?

$i \leftarrow 0$
$U_0 := bad$
repeat
  $U_{i+1} \leftarrow U_i \cup Pre(U_i)$
  $i \leftarrow i + 1$
until $U_i = U_{i-1}$
if $I \cap U_i \neq \emptyset$ **then**
  return true
else
  return false
end if

The input of the algorithm is given as the transition system $S = (\Sigma, I, \rightarrow, \succeq)$ and the upward closed set bad of configurations. Its aim is to check whether bad is reachable.

The algorithm applies the function Pre repeatedly starting with an initial set U_0 generating a sequence $U_0, U_1, U_2, \ldots$ of sets of configurations such that $U_{i+1} := U_i \cup Pre(U_i)$ for $i \geq 0$ and a set U_i containing states that can be reached within i steps from the set bad.

The algorithm terminates if for some $i > 0$: $U_i = U_{i-1}$. Then U_i contains all the states that are reachable from the set of bad configurations. We say that $minBad$ is coverable if and only if the intersection $I \cap U_i$ is not empty.

Kapitola 3

Examples of Well-quasi-ordered Transition Systems

WSTS capture many important infinite-state models. In this section, we will introduce the models that will be considered later during the discussion of articles.

3.1 Petri Nets

Petri nets are a powerful, simple and natural model for concurrent systems and programs with an unbounded number of threads or thread creation. They are used in process of verification whether the systems satisfy required correctness criteria. Examples of such criteria are, according to [10]:

- **boundedness** - the number of tokens in any place cannot grow indefinitely,
- **liveness** - from any marking any transition can become fireable,
- **reachability** - marking M is reachable from marking M_0 if there exists a sequence of firing $\sigma = t_1 t_2 t_3 \dots t_n$, that transforms M_0 to M or
- **coverability** (see Section 2.1) which is the problem we will discuss.

A Petri net (PN) is a tuple (S, T, W) where

- $S = \{s_1, s_2, \dots, s_m\}$ is a finite set of places,
- $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions disjoint from S and
- $W : (S \times T) \cap (T \times S) \to N$ is the arc multiplicity function.

For each transition $t \in T$ and for each place $s \in S$ we define, according to [7], sets:

$$t^\bullet = \{s \in S \mid W(t, s) > 0\}$$

$$^\bullet t = \{s \in S \mid W(s, t) > 0\}$$

$$s^\bullet = \{t \in T \mid W(s, t) > 0\}$$

$$^\bullet s = \{t \in T \mid W(t, s) > 0\}$$

In other words, for place $s \in S$, $s^\bullet$ is informally a set of transitions from which place can take a token while ${}^\bullet s$ is a set of transitions where a place can give a token.

A marking is used to describe an actual state of PN. It is a function $m : S \to N$ that indicates how many tokens are at place $s \in S$. If $m(s) \geq W(s,t)$ for all $s \in S$, the transition $t \in T$ is enabled at marking m. The transition from m to m' on firing t is denoted as $m|t\rangle m'$ where $m'(s) = m(s) - W(s,t) + W(t,s)$.

In a certain work, presented lately in Section 4.3, a set of places of PN $S = s_0, \ldots, s_n$ is assumed to be ordered, a marking m is a vector $(m(s_0), \ldots, m(s_n))$ and each transition t is defined as a pair

$$(g, d) \in N^n \times Z^n, \tag{3.1}$$

where

$$g = (W(s_1,t), \ldots, W(s_n,t)) \tag{3.2}$$

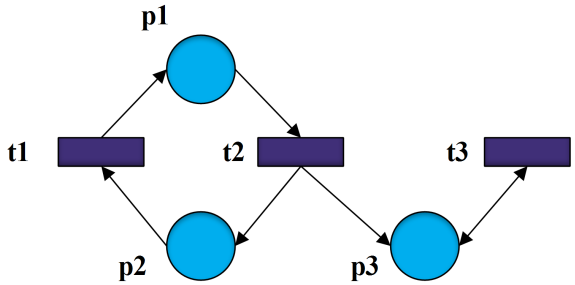
represents enabling condition and

$$d = (W(t,s_1) - W(s_1,t), \ldots, W(t,s_n) - W(s_n,t)) \tag{3.3}$$

represents the difference between the number of tokens in a place if the transition fires and the current number of tokens [9].

Example 1. To clarify the definitions, let us show the following example of PN (see Figure 3.1) defined as:

$P = \{p_1, p_2, p_3\}$
$T = \{t_1, t_2, t_3\}$
$F = \{(p_1, t_1, 1), (p_3, t_3, 2), (t_3, p_1, 0), \ldots\}$
$F(p_1, t_2) = 1$
$F(p_3, t_2) = 2$
$F(t_3, p_1) = 0$
$\{p_1, p_2\}^\bullet = \{t_1, t_2\}$



Obrázek 3.1: Simple Petri net [7]

The expression $M \xrightarrow{t} M'$ denotes that M enables the transition t and that the marking reached by firing t is M'. A finite or infinite sequence $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots$ is called a *firing sequence*. The maximal firing sequences of a PN are called *runs*. Given a sequence $\sigma = t_1 t_2 \ldots t_n$, $M \xrightarrow{\sigma} M$ denotes that there exist markings M_1, $M_2, \ldots$, M_{n-1} such that $M \xrightarrow{t_1} M_1 \ldots M_{n-1} \xrightarrow{t_n} M'$ [8].

The PN (S, T, W) induces a *WSTS* $(\Sigma, I, \rightarrow, \succeq)$ where I is a set of initial markings and Σ is the set of markings. We define the transition relation between the markings in the following way. There is an edge $m \rightarrow m'$ if and only if there is a transition $t \in T$ such that $m|t\rangle m'$ where $m, m' \in \Sigma$. We say that $m \preceq m'$ if for each $s \in S$, $m(s) \leq m'(s)$.

3.2 Thread Transition Systems

Thread transition system (*TTS*) is a class of *WSTS* that is used for modeling multi-threaded asynchronous software. It is an equivalent to Petri nets.

A thread transition system consists of threads which have access to local and shared variables. Formally, it is defined as a pair (T, Δ) where $T = S \times L$ where T is a configuration of the system from a view of a single thread consisting of the state of its local variables and the shared variables, and $\Delta \subseteq T \times T$ is a set of moves of a single thread.

Let us given a set $I_s \subseteq S$ of initial shared states and a set $I_l \subseteq L$ of initial local states. The thread transition system induces a *WSTS* $(V, I, \mapsto, \succeq)$ where $V = \bigcup_{n=0}^{\infty}(S \times L^n)$ contains n-tuples that consist of a shared state S and a vector L of local states of threads and I is a set of initial states such that $I = I_s \times (\bigcup_{n=0}^{\infty} I_l^n)$. Each transition is reflected a move of a single thread which changes its local and a shared state according to Δ. Formally, we define a transition as:

$$(s \mid l_1, \ldots, l_n) \mapsto (s' \mid l_1', \ldots, l_n'), \tag{3.4}$$

if $(s, l_i, s', l_i') \in \Delta$ for some $i \in 1, \ldots, n$ and $l_j = l_i$ for all $j \neq i$. The execution of the transition system M is a finite or infinite sequence of states of V that are pairwise related by $\mapsto$ started from an initial state of I. The states in the execution sequences are consider to be reachable states.

A relation $\succeq$ over V is defined as:

$$(s \mid l_1, \ldots, l_n) \succeq (s' \mid l_1', \ldots, l_n') \tag{3.5}$$

where $s = s'$ and $\{l_1', \ldots, l_n'\} \supseteq \{l_1, \ldots, l_n\}$. Indeed, $\succeq$ is a *wqo* and the transition relation $\mapsto$ is monotonic wrt. *wqo*.

Kapitola 4

Existing Algorithms for Coverability Checking

The following section is focused on the existing algorithms that are used for checking the coverability problem of infinite-state systems. The main aim is to present the most relevant existing works, namely [14], [9] and [3], and discuss their strengths and weaknesses.

4.1 A Complete Abstract Interpretation Framework for Coverability Properties of *WSTS*

The article [14] presents an approach that runs a program in an abstract domain. The sets of reachable states in the forward algorithm can be represented by downward-closed sets. The domain is refined using CEGAR to obtain sufficiently precise overapproximation to decide the coverability problem. In each iteration of CEGAR, the algorithm performs a forward search followed by refinement of the domain based on the backward analysis.

Initially, the input of the algorithm is given as an $WSTS$ $S = (\Sigma, I, \rightarrow, \succeq)$ and a UCS bad. The aim of the abstract interpretation algorithm is to decide whether the set bad is reachable from the initial set I or not. The abstract domain is parametrized by a parameter D. The set D defines the precision of the abstract domain and contains only the elements that are *interesting* for the computation. Intuitively, the abstraction function selects from a given DCS E only the elements contained in the parameter D of the abstract domain. Formally, it is defined as:

$$\forall E \in 2^X : \alpha[D](E) \stackrel{\text{def}}{=} E{\downarrow} \cap D. \tag{4.1}$$

While the concretisation function is defined as:

$$\forall P \in 2^D : \gamma[D](P) \stackrel{\text{def}}{=} \{x \in X \mid x{\downarrow} \cap D \subseteq P\}. \tag{4.2}$$

The result of the function applied to P is a set of all the elements such that all *interesting* elements in its DCS are contained in the set P.

An abstract post operator in the domain parametrized by D is defined in a standard way as:

$$post^\sharp[D] \stackrel{\text{def}}{=} \alpha[D] \circ post \circ \gamma[D]. \tag{4.3}$$

In particular, an abstract representation of the set P is computed, then a forward step using the procedure *post* is carried out and, subsequently, the concretisation function γ is calculated.

To find out all reachable states from the set P the forward steps are carried out till no new elements within the abstract domain parametrized by D are found:

$$post^\sharp[D]^*(P) \stackrel{\text{def}}{=} \bigcup_{i \geq 0} post^\sharp[D]^i(P). \tag{4.4}$$

When the concretisation function γ is applied, a potentially infinitely large set is created. Therefore, it is necessary to compute $post^\sharp$ in a symbolic manner. In particular, it is checked whether there is an element x in the parameter D of the abstraction domain such that its predecessor is in the concretization of the elements that have been already found to be in the set P. Formally defined:

$$x \in post^\sharp(P) \Leftrightarrow (x \in D \wedge \neg(pre(x\uparrow) \subseteq (D \backslash P)\uparrow)). \tag{4.5}$$

Thus, the set of reachable states is computed and it is necessary to check whether it contains any element from the set of incorrect states *bad*. The set *bad* is unreachable if

$$post^\sharp[D]^* \cap bad = \emptyset. \tag{4.6}$$

Otherwise, if

$$post^\sharp[D]^* \cap bad \neq \emptyset, \tag{4.7}$$

then the system can reach an incorrect state (the set *bad* is reachable) or the precision of the abstract domain parametrized by D is not sufficient and has to be refined. To find out which elements are necessary to add to the parameter of the abstract domain the backward computation is performed. It begins from the set of incorrect elements *bad* and the backward analysis computes several steps backward (the number of the steps depends on a number of iteration of CEGAR) such that the domain is enriched by newly discovered elements that are not yet included.

Let us now present functions that are necessary for the explanation of the backward algorithm. The set of predecessors forms an *UCS*. Since any *UCS* can be represented by a set of its minimal elements, the transition relation $\rightarrow$ is monotonic wrt. $\succeq$ and the set of predecessors of any *UCS* is an *UCS*, as it was discussed in Section 2.1, we can compute a set of predecessors in the following way:

$$minpre[O](T) \stackrel{\text{def}}{=} minpre(T) \cap O \tag{4.8}$$

where O is a subset X.

The backward analysis is based on the Algorithm 1 with the differences that only a few steps backward are computed and the backward search is carried out within the set O_i of elements discovered during the forward search:

$$O_i = \gamma[D_i](R_i) \tag{4.9}$$

where i is the iteration number, D_i is the parameter of the current abstraction domain and R_i is a set of all the elements reached in the forward algorithm within the current abstraction domain parametrized by D_i.

Based on the above mentioned function, the backward algorithm can be defined as:

$$R'_i \stackrel{\text{def}}{=} \left\{ min \left(\bigcup_{k=0}^{i+1} minpre[S, O_i]^k \left(bad \right) \right) \right\}. \tag{4.10}$$

The backward algorithm performs $i + 1$ steps backward within the set O_i from the set of incorrect states *bad*.

Finally, the parameter D_i of the current abstract domain is refined such that the parameter D_{i+1} of the following one is a subset of the union of D_i and the set of the new elements R'_i computed during the backward search:

$$D_{i+1} \supseteq D_i \cup R'_i. \tag{4.11}$$

A number i of iterations of CEGAR is increased and the next iteration of the algorithm is performed.

The whole algorithm can be summarized in the following way, according to [14]:

Algorithm 2 *Refinement loop*

Input:

- an $IWSTS$ S_0
- a set $bad \in UCS(X)$

Output:
Is *bad* reachable?

Let $min((bad)) \subseteq D_0$
for $i = 0, 1, 2, \ldots$ **do**
  Compute R_i defined to be $((post^\sharp[D_i])^* \circ \alpha[D_i])(x_0)$
  Let O_i denote $\gamma[D_i](R_i)$
  if $O_i \cap bad = \emptyset$ **then**
   return UNREACHABLE
  else
    Compute R'_i defined to be $min(\bigcup_{k=0}^{i+1} minpre[O_i]^k(bad))$
    if $\{x_0\} \cap R'_i{\uparrow} = \emptyset$ **then**
      choose $D_{i+1} \supseteq D_i \cup R'_i$
    else
      return REACHABLE
    end if
  end if
end for

4.2 Incremental, Inductive Coverability

The article [9] describes a procedure based on the IC3 algorithm [1] for checking coverability. The algorithm has been originally proposed to finite-state hardware verification. The article [9] generalizes it to coverability of *WSTS*.

An input of the algorithms is a *WSTS* $(\Sigma, I, \rightarrow, \preceq)$ and a *DCS* of *correct* states $P\downarrow$. On the abstract level, the algorithm computes the sequence $U = U_0\uparrow, U_1\uparrow, \ldots, U_N\uparrow$ of *UCS* defined as:

$$U_0\uparrow = \Sigma \backslash P\downarrow, \tag{4.12}$$

$$U_{i+1}\uparrow = U_i\uparrow \cup pre(U_i\uparrow). \tag{4.13}$$

The sequence U represents the states from where errors are reachable, and it eventually stabilizes, i.e. there exists L: $U_L\uparrow = U_{L+i}\uparrow$ for all $i \geq 0$ since $\preceq$ is *wqo*. The set $U_0\uparrow$ corresponds to a set *bad* in article [14] which consists of incorrect configurations.

Given an initial set I, the algorithm creates a path from a state in I to a covering set for $P\downarrow$ or to a state not in $P\downarrow$ (if $Cover \not\subseteq P\downarrow$). The set $Cover$ is a set of reachable states from I.

If the initial set I does not contain any of bad states from a set $U\uparrow$, $I \cap U\uparrow = \emptyset$, then $Cover \subseteq P\downarrow$. In addition, if the condition of the disjunction of the initial set and the set of bad configurations is fulfilled, then the complement $\Sigma\backslash U\uparrow$ contains the initial set I and satisfies $post(\Sigma\backslash U\uparrow) \subseteq \Sigma\backslash U\uparrow$.

A set $C\downarrow$ is considered to be a covering set for $P\downarrow$ if it has the following properties:

- $I \subseteq C\downarrow$ (includes the initial set),
- $C\downarrow \subseteq P\downarrow$ (is safe) and
- $post(C\downarrow) \subseteq C\downarrow$ (is inductive).

Let $(\Sigma, I, \rightarrow, \succeq)$ be a *WSTS* and a set $R\downarrow$ such that $I \subseteq R\downarrow$. We say that $S\downarrow$ is inductive relative to $R\downarrow$ if $I \subseteq S\downarrow$ and

$$post(R\downarrow \cap S\downarrow) \subseteq S\downarrow, \tag{4.14}$$

then also

$$pre(\Sigma\backslash S\downarrow) \cap R\downarrow \cap S\downarrow = \emptyset. \tag{4.15}$$

On the contrary, an *UCS* $U\uparrow$ is inductive relative to $R\downarrow$ if $I \subseteq S\downarrow$ and

$$post(R\downarrow\backslash U\uparrow) \subseteq \Sigma\backslash U\uparrow, \tag{4.16}$$

then also

$$pre(U\uparrow) \cap R\downarrow\backslash U\uparrow = \emptyset. \tag{4.17}$$

A state of the algorithm is described with a pair $R \mid Q$ where Q is a priority queue and R is a sequence of *DSC* $R_0^\downarrow, \ldots, R_N^\downarrow$ such that $R_0^\downarrow = I\downarrow$. Each *DCS* R_i is an over-approximation of the set $Cover_i$ denoting the set of the states reachable in i or less steps from the initial set I.

The algorithm works with sets that are not necessarily inductive by themselves, however, they are inductive relative to some other sets. The algorithm maintains an invariant such that $R_{i+1}^\downarrow$ is inductive relative to $R_i^\downarrow$ in each step and $R_i^\downarrow \subseteq P\downarrow$ for $i < N$. If the vector stabilizes, i.e. $R_{N-1}^\downarrow = R_N^\downarrow$, then R_N can be considered an inductive covering set for I.

First, it is checked whether the set R_N contains possible counter-example. If a counterexample exists it is stored in the priority queue Q as a pair $\langle a, N\rangle$ where $a \in \Sigma$ is a state and N is the level of the element or also the priority of the element. Subsequently, the algorithm creates a counterexample trace backward through all the levels of the vector R searching for predecessors of a such that $a_0 \to \ldots \to a_N$, where $a_i \in R_i^{\downarrow}$ for all i. All the predecessors a_i are also stored in the priority queue Q with the priority i.

If the counter-example cannot be extended backwards within $R_i^{\downarrow}$, the set $R_i^{\downarrow}$ is refined by removing some $b\uparrow$ from $R_0^{\downarrow}, \ldots, R_i^{\downarrow}$ such that $a \in b\uparrow$. However, not only the states from the counter-example trace are removed. We can also remove the whole set of states that are valid generalizations of states in Q relative to some set $R_i^{\downarrow}$ defined as:

$$Gen_i(a) := \{b \mid b \preceq a \wedge b\uparrow \cap I = \emptyset \wedge pre(b\uparrow) \cap R_i^{\downarrow} \backslash b\uparrow = \emptyset\}. \tag{4.18}$$

The set consists of the states that are a part of the counterexample traces and their most general elements that do not have any predecessors within R.

If no more updating of vector R is possible, N is increased and a forward step is carried out. This phase is called inductive straightening. The algorithm terminates in states **valid** ($Cover \subseteq P\downarrow$) or **invalid** ($Cover \not\subseteq P\downarrow$).

The algorithm in [9] is described precisely as a procedure which applies the following rules (in an arbitrary order):

[Initialize] The algorithm starts with the set $R_0^{\downarrow} = I\downarrow$ and the empty queue Q.

[CandidateNondet] If a state $a \in R_N^{\downarrow}$ that is not in $P^{\downarrow}$ is found, it is added to the queue Q as an element $\langle a, N\rangle$.

[DecideNondet] The state a of the lowest level i is picked from the queue Q and all its predecessors b such that $b \in pre(a\uparrow)$ and $b \in R_{i-1}^{\downarrow}$ are added to Q with the level $i-1$.

[Model] If a state a from 0 level is contained in Q, the algorithm terminates in the state **invalid** because a counter-example trace has been found.

[Conflict] If a does not have a predecessor within $R_{i-1}^{\downarrow}\backslash a\uparrow$, then a belongs to a spurious counter-example trace. The state a cannot be reachable in i steps and therefore its upward-closed set $a\uparrow$ can be removed from all the sets $R_1^{\downarrow}, \ldots, R_i^{\downarrow}$. Moreover, also a bigger set containing all the states $b \in Gen_{i-1}(a)$ can be removed. After the updating of R, the element $\langle a, i\rangle$ is removed from the queue Q. Subsequently, the element $\langle a, i+1\rangle$ is added to Q to speed up the search.

[Induction] In the case, when the states $r_{i,1}, \ldots, r_{i,m}$ that have been removed from $R_i^{\downarrow}$ become inductive relative to $R_i^{\downarrow}$, i.e. such that $post(R^{\downarrow}\backslash r_{i,j}\uparrow) \subseteq \Sigma\backslash r_{i,j}\uparrow$, then the states in $r_{i,j}$ cannot be reached also in $i+1$ steps. They can be then safely removed also from $R_{i+1}^{\downarrow}$ as well as their generalizations $b \in Gen_i(r_{i,j})$.

[Valid] The algorithm terminates with the answer **valid** if there is a $R_i^{\downarrow}$ such that $R_i^{\downarrow} = R_{i+1}^{\downarrow}$.

[Unfold] If the priority queue Q is empty, the forward step is carried out: $R_{N+1}^{\downarrow} = \Sigma$ is attached to the vector R.

In an implementation, these rules are usually applied in the following way. The predecessors of the possible counter-example a create a chain $a_0, \ldots, a_N$, $a_i \in R_i^{\downarrow}$. First, a possible end-point of the chain from the lowest level is found using [CandidateNondet]. Then the procedure [DecideNondet] is applied to search its predecessor. If no predecessor from level 0 (checking by [Model]) is found, then $R_1^{\downarrow}, \ldots, R_i^{\downarrow}$ is refined applying [Conflict] that removes unreachleable states. If there is no more progress in refining of the vector R, [Unfold] is applied.

In the phase of inductive strengthening, the procedure [Induction], is repeatedly applied. If some pre-conditions fail to hold, it is checked whether [Valid] applies. If not, the algorithm will continue to the next round.

4.3 Coverability for Petri Nets

In Section 4.2, the algorithm for the coverability problem was presented. Now, the implementation of the general algorithm for the coverability problem for Petri nets will be discussed.

The algorithm in [9] works with a downward-closed set $R_i^{\downarrow}$ that initially contains all the states. Consider $b\uparrow$ as a *UCS* of bad states that are blocked by b at level i. The aim of the algorithm is to remove states of the *UCS* $b\uparrow$ from $R_i^{\downarrow}$ where $b \in \mathbb{N}^n$. Thus, the set $R_i^{\downarrow}$ will be defined as:

$$R_i^{\downarrow} = \Sigma \backslash \{b_1, \ldots, b_l\}\uparrow. \tag{4.19}$$

However, it is conceptually represented as $\{b_1, \ldots, b_l\}$.

The algorithm uses delta encoding. Consider finite sets B_i and B_{i+1}. Let $R_i^{\downarrow} = \Sigma \backslash B_i\uparrow$ and $R_{i+1}^{\downarrow} = \Sigma \backslash B_{i+1}\uparrow$, then $B_{i+1}\uparrow \subseteq B_i\uparrow$. Since b is blocked on the level i and lower, only a vector $F = \{F_0, \ldots, F_N, F_{\text{inf}}\}$ is needed to be maintained such that $b \in F_i$, F_i is a difference between $B_i^{\downarrow}$ and $B_{i+1}^{\uparrow} \cup \ldots \cup B_N^{\uparrow}$, F_{inf} represents a set of states that can never be reached.

For

$$(R_0^{\downarrow}, R_1^{\downarrow}, R_2^{\downarrow}) = (\{i_1, i_2\}, \{b_1, b_2, b_3, b_4\}, \{b_2, b_3\}), \tag{4.20}$$

where b_i are blocked elements in R_i, the matching vector F is defined as:

$$(F_0, F_1, F_2, F_{\text{inf}}) = (\{i_1, i_2\}, \{b_1, b_4\}, \{b_2, b_3\}, \emptyset). \tag{4.21}$$

The implementation involves the rules presented in Section 4.2 adapted to Petri nets.

[CandidateNondet] Testing whether a is contained in a set $R_k^{\downarrow}$ using the delta-encoded vector F means iterating over F_i for $k \leq i \leq N+1$ and checking if there is any $c \preceq a$ such that $c \in F_i$. If such a state exists, then a is blocked by c, otherwise, $a \in R_k^{\downarrow}$. In case of $k = 0$, we search for c only in F_0.

[DecideNondet], [Conflict], [Induction] In these rules, it is necessary to find predecessors $pre(a\uparrow)$ in $R_i^{\downarrow} \backslash a\uparrow$ and their generalizations. If no such predecessors exist, then relative

inductiveness is concluded. The following lema shows how to compute the most general predecessor along a fixed transition directly.

Lemma 1 *Let $a \in N^n$ be a state and $t = (g, d)$ be a transition. Then $b \in pre(a\uparrow)$ is a predecessor along t if and only if $b \succeq max(a - d, g)$* [9].

Therefore, it is needed to iterate through all transitions $t = (g, d)$ and find the transition such that $max(a-d, g) \in R_i^{\downarrow} \backslash a\uparrow$. If there is no such transitions, then $a\uparrow$ is inductive relative to $R_i^{\downarrow}$ and thus, predecessor of a, $max(a - d, g)$, is blocked by a itself, or a state $c_t \in F_{i_t}$, for some $i_t \geq i$, such that $c_t \preceq max(a - d, g)$.

We define

$$i' := min\{i_t \mid t \text{ is a transition}\}, \tag{4.22}$$

where $i_t := N + 1$ for $t = (g, d)$ if $max(a - d, g)$ is blocked by a itself. Then $i' \geq i$ and $a\uparrow$ is inductive relative to $R_{i'}^{\downarrow}$.

The following lema shows how to compute a state $a' \preceq a$ such that for all transitions $t = (g, d)$, $max(a' - d, g)$ remains blocked by a' itself, or by c_t.

Lemma 2 *Let $a, c \in \mathbb{N}^n$ be states and $t = (g, d)$ be a transition.*

- *Let $c \preceq max(a-d, g)$. If $g_j < c_j$, then $a''_j := c_j + d_j$ and if $g_j \geq c_j$, then $a''_j := 0$, for all $j = 1, \ldots, n$. Then $a'' \preceq a$ and for each a' such that $a'' \preceq a' \preceq a$, $c \preceq max(a' - d, g)$.*
- *If $a \preceq max(a-d, g)$, then for each a' such that $a' \preceq a$, it holds that $a' \preceq max(a'-d, g)$* [9].

If the predecessor of a, $max(a - d, g)$, is blocked for each transition $t = (g, d)$, then a'' is defined in the following way.

- If the predecessor is blocked by some state $c_t \in F_{i_t}$, then a''_t is defined as in Lemma 8 and the predecessors of a'' remain blocked by c_t.
- If the predecessor is blocked by a itself, then $a''_t := (0, \ldots, 0)$ and the predecessors of a'' remain blocked by a'' itself.

The state a'' is defined to be the pointwise maximum of all states a''_t.

However, if a'' is in $R_0^{\downarrow}$, then $a' := max(a'', c)$ where c is any $c \in F_0$ that blocks a (such a state exists because $a \notin R_0^{\downarrow}$) is a valid generalization: $a' \preceq a$ and $a'\uparrow$ is inductive relative to $R_{i'}^{\downarrow}$.

In [Conflict] and [Induction], it is necessary when blocking a generalized upward-closed set $a'\uparrow$ inductive relative to $R_{i'}^{\downarrow}$ for $i' < N$ to update the vector F by adding a' to $F_{i'+1}$. However, if $i' = N$ or $i' = N + 1$, a' is added to $F_{i'}$. Moreover, for $1 \leq k \leq i' + 1$ (or $1 \leq k \leq i'$) all states $c \in F_k$ such that $a' \preceq c$ are removed.

[Valid] The algorithm terminates, if F_i is empty for some $i < N$.

[Unfold] If all bad states from $R_N^{\downarrow}$ were removed, N is increased and an empty set is inserted to position N in the vector F, thus, F_{inf} is pushed from position N to $N+1$.

4.4 Efficient Coverability Analysis by Proof Minimization

The algorithm represented in [3] is based on backward search 1. It tries to generalize the found predecessors of *bad* by guessing smaller candidates. This way accelerates the computation and minimizes the proof of uncoverability, thus, failing to contribute to the proof minimization. To eliminate such unhelpful canditates, a forward search is used to simultaneously generate a set of coverable elements. The simultaneously executed forward search does not effect the overall results only the speed of the algorithm. During the forward search represents an underapproximation, the backward search represents overapproximation. In the notation of the paper, the set *bad* is called initial and denoted I.

Let q be an uncoverable state and I a set of an initial states. Brs denotes an upwared-closed set of states that have an emanating execution leading to a state q. Since q is uncoverable, $Brs \cap I = \emptyset$. Its overapproximation is the set $Brs^\sharp$, also fulfilling the condition of being disjoint with the initial states

$$q \in Brs^\sharp,\ CPre(Brs^\sharp) \subseteq Brs^\sharp,\ Brs^\sharp \cap I = \emptyset \tag{4.23}$$

where a procedure $CPre$ is defined as:

$$CPre(v) = min(pre(v\uparrow)) \tag{4.24}$$

and computes the predecessors of a state $v \in V$. The function $CPre$ returns a set of general predecessors of UCS $v\uparrow$. The uncoverability proof $Brs^\sharp$ for q is minimal if

$$min(Brs^\sharp) \subseteq min(V \backslash Cover) \tag{4.25}$$

where $Cover$ are states reachable from the initial states and no upward-closed subset $X \subset Brs^\sharp$ is an uncoverability proof for q.

The algorithm maintains the following sets during the computation:

- a set U of labeled and identified vertices with encountered states,
- a set W of unprocessed vertices,
- a set D of coverability results,
- a set E of edges within a set U, $E \subseteq U \times U$ and
- a mapping ζ.

The mapping ζ associates each vertex with exactly one vertex, ζ: $U \rightarrow U$. Each vertex $u \in U$ is called *candidate vertex* if $\zeta(u) = u$, otherwise, the vertex u is called *predecessor vertex*. Moreover, the mapping ζ can be extended to sets X, $\zeta(X) = \{\zeta(x) \mid x \in X\}$ thus the vertices are clustered into $\mid \zeta(U) \mid$ partitions, one per candidate vertex.

The backward search starts with the working set W and the set U containing only the target q that represents a state where error occurs. The set E of edges is empty and the function ζ associates q to itself. The algorithm consists of three procedures - *Enlarge* -

creating new candidates, *Backtrack* - removing the partitions of unhelpful candidates and *Mcov* - the main routine. The algorithm ensures any time, restricting of the partitioned graph (U, E, ζ) to any equivalence class of vertices with the same associated candidate vertex forms a tree with the candidate vertex as a root and all other vertices as their predecessor vertices according to the relation E.

Let $s \in U$ be a vertex. The set U expands by adding next candidate that leads to s. The set of potential candidates $C(s) \subseteq V$ leading to s is defined as:

$$C(s) = \{v \in V \mid v \prec s \wedge v \notin D\}. \tag{4.26}$$

The states in the set lead to s but are not yet marked.

Subsequently, one candidate p from the set of minimal candidate vertices, $min\ C(p)$, is picked and new minimal candidate vertices are created using the *Enlarge* routine. The routine takes vertex u as an input and, if it is a new vertex, it adds the vertex to the set U and the working set W. Then the graph is repartitioned by adjusting ζ and every vertex in the set to preserve a shape of tree

$$\Lambda(u) = \{r \in U \mid u = r \vee (r \rightarrow^* u \wedge \zeta(r) = \zeta(u))\} \tag{4.27}$$

is associated with vertex u. Thus, $r \in \Lambda(u)$ now entails $\zeta(r) = u$.

In the next step, a minimal vertex from the working set W is picked. If no more unprocessed vertex is in the set W, then the algorithm terminates with a result $q \notin Cover$, thus proving the uncoverability of the target q. Otherwise, all covering predecessors p of vertex w are found using the procedure $Cpre(w)$ and proceed if they are $\zeta(w)$-minimal, as defined below.

Definition *Let $v \in V$, and $u \in \zeta(U)$. State v is u-minimal if $v \not\geq u$ and for all $s, s' \in U$ such that $s \rightarrow s'$ and $\zeta(s') = u$, we have $v \not\geq s$* [3].

If p is not coverable, $p \notin D$, then the graph is expanded by adding an edge (p, w) to the set E. If p is a new vertex, $p \notin U$, then a predecessor p is added to the sets U and W and the mapping function ζ is changed such that $\zeta(p) = w$. Finally, *Enlarge* routine is called to create new candidates.

In case that p is coverable but not q, then $\downarrow p$ is added to the set of coverable states D because if p is coverable, then also its *UCS* is coverable and *Backtrack* routine is evoked. The purpose of the backward search is to delete unhelpful candidates $P \subseteq \zeta(U)$ and their partitions. However, a part of the partition may be shared with remaining candidates and thus it is necessary to ensure that only unhelpful candidates are deleted and to preserve the parts of the partition that are shared with remaining candidates. Therefore, for P-conflict edges $(r, s) \in E$ such that $\zeta(r) \in P$ and $\zeta(s) \notin P$, the vertices in $\Lambda(r)$ are reassociated to $\zeta(s)$. Then all unhelpful candidates can be removed from the sets U and W and also all edges associated with these candidates can be removed from E.

If there is a state u in the set $min(U)$ such that $u \in min(U) \cap \uparrow p$, then new candidates are created using the routine *Enlarge*.

If no previous condition is fulfilled, then $q \in D$, and hence the algorithm terminates in a state $q \in Cover$.

The algorithm eventually terminates due to the finiteness of downward-closures and the fact that during backtracking, only the conflicting edges are removed. If the algorithm

terminates in a state $q \notin Cover$, which means that the target q is uncoverable, then all the remaining minimal nodes from the set U represent an uncoverability proof for q: $Brs^\sharp = \uparrow U$.

Kapitola 5

Determination of inductive invariant

In this chapter, we will explain the notion of inductive invariant. The existing approaches to finding the invariant will be discusses along with their pros and cons and a improved method will be proposed.

5.1 Inductive invariants

The most recent methods for formal verification of systems with unbounded parallelism are based on the concept of the forward search combined with the backward analysis to find an invariant. It is a property that holds in every reachable state and can be proved by simple induction. Our effort is to find the invariant with the smallest possible representations since the smaller is its representation, the more effective its testing usually is, and the faster it can be generated. A safe inductive invariant is a set of states of the system with three following properties:

- contains its initial states (base),
- does not intersect with the undesired states (safety), and
- is closed under the transition relation (induction).

The properties together are an inductive proof of safety of the system. Let us illustrate the notion of safe inductive invariant on an example of the Peterson's algorithm (Algorithm 3). The basic variant of the algorithm ensures exclusive access to critical section in a system with two processes. It corresponds to the situation where the first process is in a state q_4 and the second one in a state r_4. The configuration where these two states simultaneously occur is a part of the set *bad* representing the incorrect configurations of the system. The task is to verify that no incorrect configuration from *bad* is reachable thus, the system is safe.

Algorithm 3 *Peterson's mutual exclusion*

```
int x  = 0; int y = 0; int T;

q1:     x = 1;                          r1:     y = 1;
q2:     T = 1;                          r2:     T = 0;
q3:     while (y == 1 && T == 1)        r3:     while (x == 1 && T == 0)
          {                                       {
              // busy wait                            // busy wait
          }  // critical section                   }  // critical section
q4:             ...                     r4:             ...
       // end of critical section                 // end of critical section
          x = 0;                                  y = 0;
```

For an example of Peterson's mutual exclusion algorithm a simple invariant can be expressed as a formula over literals talking about states of the system:

$$\begin{aligned}\neg((q_3 \wedge x_0) \vee (q_4 \wedge x_0) \vee (q_2 \wedge x_0) \vee (r_4 \wedge y_0) \vee \\ (r_3 \wedge y0) \vee (r_2 \wedge y_0) \vee (q_4 \wedge r_3 \wedge T_1) \vee \\ (q_3 \wedge r_4 \wedge T_0) \vee (q_4 \wedge r_4))\end{aligned} \tag{5.1}$$

where q_i or r_i means that the control of the left or of the right process, respectively, is at the i-th line, x_i, y_i and T_i means that a state of the appropriate variables is i.

To ensure that the formula represents an inductive invariant we have to verify that the three above described properties are fulfilled.

[Base] The initial state of the system is a configuration where the two processes are in the states q_1 and r_1. The formula holds for these states, therefore, this property is fulfilled.

[Safety] Since the configuration with two processes in the critical section, represented by a predicate $q_4 \wedge r_4$, is excluded by the formula the invariant does not intersect with the incorrect states.

[Induction] The formula can be expressed as a conjunction of formulas: $\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5$ where $\phi_1 = (q_i \Rightarrow x_1, i \in \{2, \ldots, 4\})$, $\phi_2 = (r_i \Rightarrow y_1, i \in \{2, \ldots, 4\})$, $\phi_3 = (q_4 \wedge r_3 \Rightarrow T_0)$, $\phi_4 = (q_3 \wedge r_4 \Rightarrow T_1)$ and $\phi_5 = (\neg q_4 \vee \neg r_4)$. The property of induction is fulfilled if any transitions from a configuration where the formula is satisfied lead to a configuration where the formula is satisfied as well. We will argue that it is true for each formula ϕ_i, for $i = \{1, \ldots, 4\}$.

First, we will discuss the formula ϕ_1. Performing transition from q_1 to q_2, x is set to 1. Since the process q does not change the value of x on its way from a state q_2 to q_4, and since the process r does not change the state of x at all, the formula ϕ_1 is satisfied. The argument of the formula ϕ_2 is symmetric to ϕ_1.

Consider the formula ϕ_2, the configuration with states q_4 and r_3 is reachable by transitions $q_3 \to q_4$ or $r_2 \to r_3$. The transition $q_3 \to q_4$ is possible if y_0 or T_0. The case y_0 is excluded by the formula ϕ_2. In case of T_0, ϕ_3 is satisfied (since T_0 holds).

Finally, the transition $r_2 \to r_3$ sets T to 0 which guarantees that ϕ_3 is satisfied. The argument of the formula ϕ_4 is symmetric to ϕ_3.

Consider the formula ϕ_5, the configuration where q_4 and r_4 can occur by performing transitions $q_3 \to q_4$ or $r_3 \to r_4$. The process can change its state from $q_3 \to q_4$ if y_0 which is excluded by the formula ϕ_2, or if T_1 which is excluded by the formula ϕ_4. The transition $r_3 \to r_4$ can be fired if x_0 which is excluded by the formula ϕ_1, or if T_1 which is excluded by the formula ϕ_3, thus the formula ϕ_5 is satisfied.

Methods of finding invariant Let us now present how the notion of inductive invariant is represented in the state of the art methods presented in Section 4. The algorithm from Section 4.1 is based on running a program in an abstract domain. The abstract domain is parametrized by a set of configurations D which defines its precision. The algorithm converges if the parameter D has a subset V such that a complement of the *UCS* $V\uparrow$ is an inductive invariant. Since elements of D are almost the only data the algorithm works with (besides transition function) we can say that the smaller inductive invariant is, the faster the algorithm is.

The algorithm from Section 4.2 is based on a method of eliminating configurations $a_0, a_1, \ldots$ that are a part of spurious counterexample paths. The convergence of the algorithm is guaranteed if the complement of the *UCS* of the upward closure of these configurations forms an inductive invariant. In the notation of 4.2, the inductive invariant is then represented by the vector R. Certain emphasis on finding more succinct invariants is noticeable in the design of the algorithm. Particularly, the procedure *Gen* defined by the equation (4.2) tries to guess generalizations of the configurations from the spurious counterexample paths which leads to inferring number of smaller configurations (that represent the invariant).

The core of the algorithm from Section 4.4 is backward search for minimal uncoverable elements from a set of incorrect configurations. During the backward analysis, a tree of possible predecessors is created and the unhelpful candidates are removed. The algorithm converges in case that an *UCS* of a set $U\uparrow$ of minimal nodes covering q, denoted by Brs, is an inductive invariant. The algorithm attempts to guess generalizations of potential candidates $C(p)$ (this is similar to the role of the procedure *Gen* in the algorithm from 4.2). The smaller the covering predecessors we examine, the shorter the paths the algorithm needs to traverse.

All the three algorithms learn the invariant of the same form - as the complement of an *UCS*. It is evident that a heuristic leading to finding more succinct invariants would help to accelerate the overall performance of the algorithm. The last two methods are about to guess the succinct invariants in a certain way. The performance of the three algorithms is similar, as it is shown in the result of the experiments in [11]. We have found experimentally that the invariant in case of the algorithm from Section 4.1 has a size of 135 disjuncts, as discussed detailedly in Section 8, which offers a possibility of improving the method since we discovered the existence of the invariant of the size 9 disjuncts (see formula (5.1)).

Analysis of the optimization The current methods learn invariants based on the counterexample runs. The backward analysis explores the state space of the system using an operation *pre* starting in a set of incorrect configurations. The search is performed in a rather Breadth-first manner, and essentially all found configurations are being added to the constructed representation of the invariant (modulo some rather local optimizations

such as the "generalization" of algorithms 4.2 and 4.4). Due to the absence of a global view of the counterexample runs, the methods cause an unnecessary increase in the number of inspected configurations that form an approximation of the invariant and thereby the inductive invariant is unnecessarily verbose. The example can be the case of Peterson's algorithm where the method presented in [14] found an invariant corresponding to a formula with 135 disjuncts, while we proved the existence of the invariant with 9 disjuncts. We thus believe that many systems have much more succinct invariants than what can be found by the current methods, and that by designing techniques focusing on the search of succinct invariants, we can improve efficiency of the current approaches significantly.

This work is inspired by certain initial ideas about the search for an invariant within the method of Section 4.1 since it can be directed towards more succinct invariants. Particularly, we will replace the breadth-first backward state space exploration of all configurations that can reach the set *bad* by exterminating of so called minimal (abstract) counterexample runs. A minimal counterexample run is consider to be a run within an abstract domain, leading from an initial set to a set of incorrect configurations *bad* which is executed using a minimal set of transitions. Using a more detailed analysis, we will determine a minimal (most succinct and most general) reason that the run is spurious - it is not feasible in the concrete domain. This minimal reason (in a form of a set of configurations) will then be used to refine the abstract domain (it will be included into D). The minimal reason has a potential to be a part of the minimal inductive invariant since 1) it is necessary to refute the examined spurious counterexample run and 2) it is a "minimal" such "reason".

Overall, the modification of the method of Section 4.1 has two steps. First, its counterexample analysis loop must be modified to proceed rather depth-first instead of breadth-first search, and generate minimal counterexample runs. Second, a method of analysis produced minimal counterexample runs is needed such that would determine the most succinct invariants (that can refute them).

The aim of this work is not to design the whole method yet. This would exceed usual bounds of a master thesis. We rather focus on verifying that it is sensible as a research direction, and on verifying that succinct invariants indeed exist for many practical systems. Therefore, in the practical part of the thesis, we will proceed as follows.

We will implement the modification of the algorithm presented in [14] and re-implement the original algorithm as well, in order to compare its performance with the performance of its modified version. The modification is based on the implementation of the depth-first search whose effort would be to find minimal runs. The second part will be implemented as a simple and brute-force analysis of minimal counterexample runs. An efficient method of the analysis will be a subject of a further research.

Kapitola 6

Modified analysis

This chapter describes the core of the technical part of this thesis. We first discuss the use of Petri nets for encodings of programs, the instantiation of the framework of [14] to Petri nets, followed by a representation of abstraction and an implementation of the algorithm presented in [14] and its modified version, discussed in the previous section.

6.1 Petri net

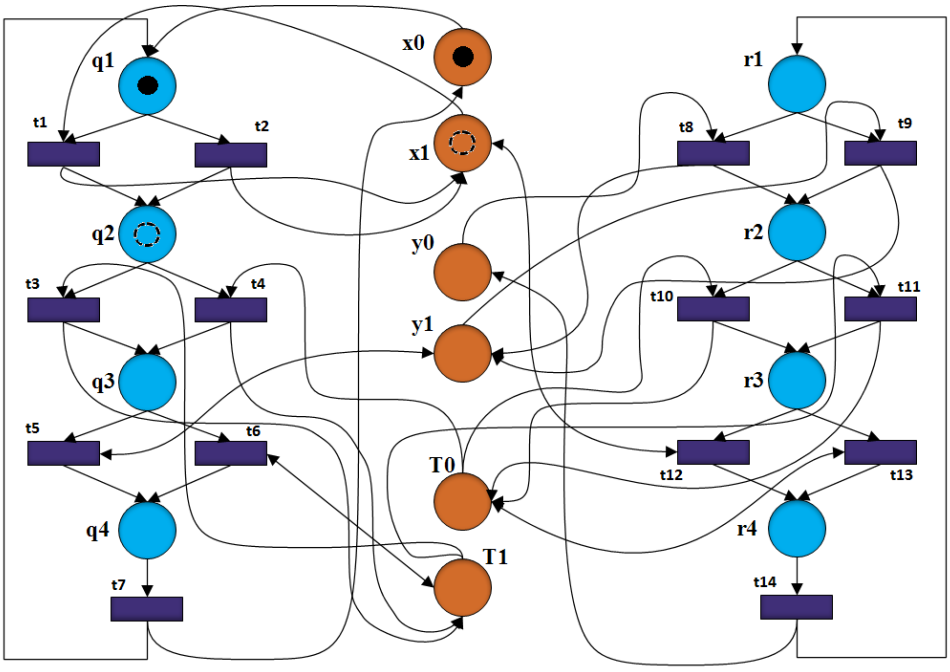
We use Petri net, a tuple $N = (S, T, W)$, to simulate and test behavior of systems with parallel running processes. Places and transitions of Petri net may have for different systems different interpretation. Places can represent required/freed resources, input/output data, input/output signals or buffers while transitions can represent tasks, computations, signal processing or processors.

As an example, we model the Peterson's mutual algorithm (Algorithm 3) as a Petri net. Configurations of the system are multisets over the alphabet $\{q_1, q_2, q_3, q_4, r_1, r_2, r_3, r_4, x_0, x_1, y_0, y_1, T_0, T_1\}$. In this chapter, we will denote a multiset as a word over the alphabet, and we will sometimes use the word terminology, for instance a subword for a submultiset. Each letter of the alphabet represents a one token in a place in the Petri net and corresponds to a literal in the formula (5.1).

If the first process changes its state from q_1 to q_2, it executes an instruction $x = 1$. It corresponds to firing the transition t_1: the token from the place q_1 moves to the place q_2 and the token from the place x_1 moves to the place x_0, or if the token is at the place x_1 it remains there. The Petri net modeling Peterson's algorithm is shown in Figure 6.1[1].

The aim is to check that the protocol guarantees exclusive access to a critical section regardless of the number of processes. A violation of mutual exclusion is considered a word with simultaneous occurrence of letters q_4 and r_4 that are represented in Petri net by a occurrence of tokens at places q_4 and r_4. The configurations that violate the property are given by $bad = \{q_4 r_4\}\uparrow$. Note that bad is an UCS: whenever a configuration contains two processes in the critical section then any larger configuration will contain (at least) two processes in the critical section as well.

[1] We know that studying of the PN is not necessary for understanding the rest of the thesis (but it is an intriguing exercise.)



Obrázek 6.1: Peterson's mutual exclusion algorithm modeled as a Petri net

6.2 Abstraction

In chapter 3.1, we introduces Petri net as an instance of *WSTS*. Analysis of Petri nets has to tackle the problem of the state space explosion, and, for instance when Petri nets are used to model systems with unbounded parallelism, also infinite state spaces. Hence, it is necessary to use abstraction. The abstraction $\alpha[D](c)$ defined in the equation (4.1) returns all subwords of the configuration c contained in the parameter D of the abstraction domain.

Let us consider the Peterson's mutual exclusion protocol and the following parameter $D = \{q_1, q_2, q_3, q_4, r_1, r_2, r_3, r_4, x_0, x_1, y_0, y_1, T_0, T_1, q_4r_4\}$. Given a configuration $c = q_4r_4T_0x_0$, after applying $\alpha[D]$ on c, we receive the abstract set $\alpha[D](c) = \{q_4r_4, q_4, r_4, T_0, x_0\}$.

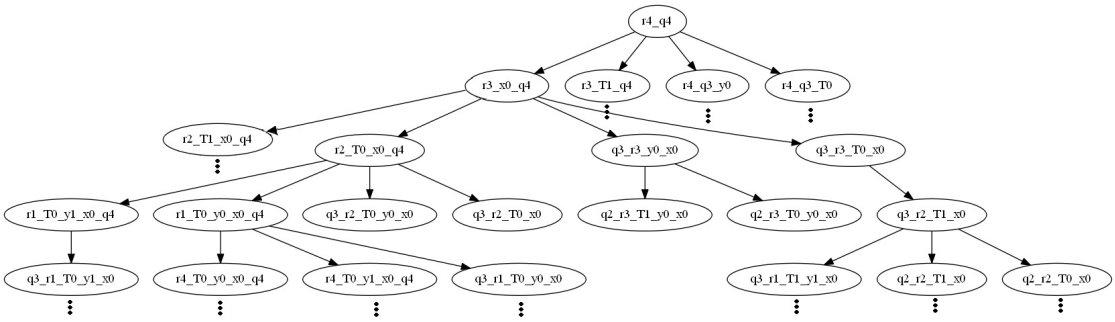
6.3 Forward search

The algorithm in Section 4.1 is presented in terms of *WSTS*. Nevertheless, it can be instantiated to Petri nets. Given a Petri net $N = (S, T, W)$ and an *UCS bad*, we check whether we can fire a sequence of transitions starting from an initial marking x_0 and hit the set *bad*. Checking the safety property equals to deciding reachability of the *UCS bad*.

Algorithm 4 *Forward reachability*

Input:

- $N = (S, T, W)$: Petri net
- *bad*: *UCS* of incorrect markings



Obrázek 6.2: Backward analysis GRB

Output:
Is *bad* reachable?

$P_i = \alpha[D](x_0)$
path = ""
while $P_i \neq P_{i-1}$ **do**
  for t *in* T **do**
    if t.isEnabled(P_i) **then**
      path = path . t
      $P_{i+1} = P_i \cup post^{t\sharp}[D](P_i)$
      $i = i + 1$
    end if
  end for
end while
if $P_i \cap bad = \emptyset$ **then**
  return UNREACHABLE
else if $\exists x_0, \ldots, x_k \in D : x_0 \to \cdots \to x_k \in bad$ **then**
  return REACHABLE
end if

The algorithm considers the abstraction domain parametrized by D and the abstract function $\alpha[D]$ (4.1). The function t.isEnabled() returns $True$ if there is a marking in $\gamma[D](P_i)$ such that the transition t can be fired from the marking, and $False$ otherwise. The function $post^{t\sharp}[D](P_i)$ represents a set of abstract successors of P_i under the transition t. It is computed in a symbolic manner analogical to an abstract post operator $post^{\sharp}$ (4.5) with the difference that it is restricted to the transition t that is:

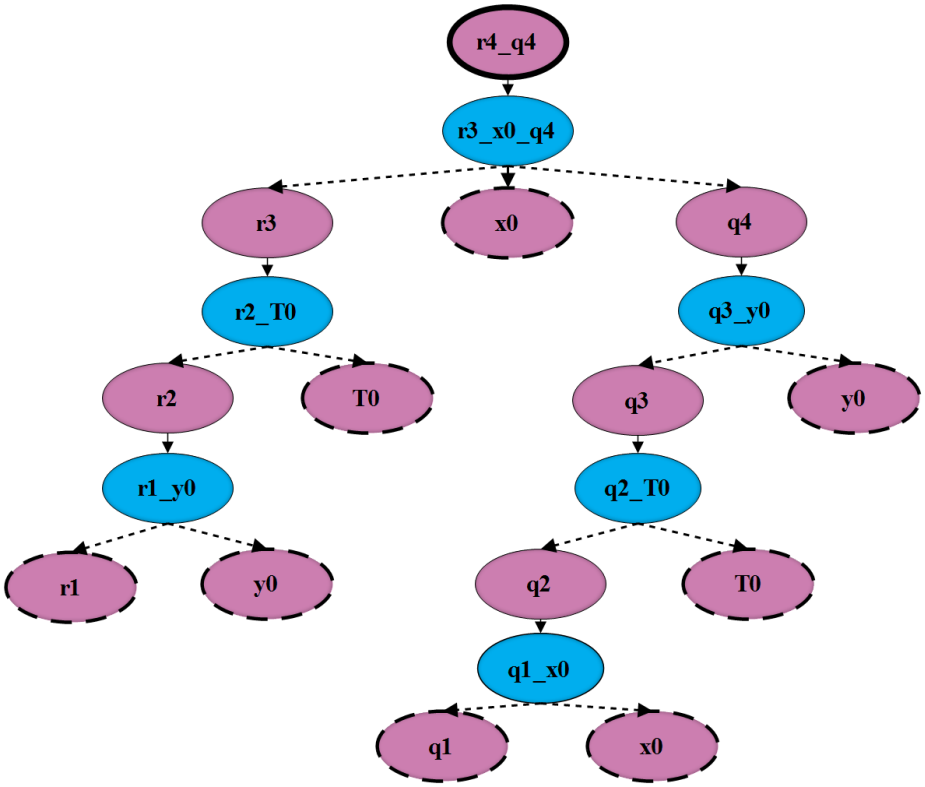
$$x \in post^{t\sharp}(P) \Leftrightarrow (x \in D \wedge \neg(pre^t(x\uparrow) \subseteq (D \backslash P)\uparrow)), \tag{6.1}$$

where

$$pre^t(X) = \bigcup_{x \in X} pre^t(x) \tag{6.2}$$

and

$$pre^t(x) = \{y \mid y \xrightarrow{t} x\}. \tag{6.3}$$



Obrázek 6.3: Backward analysis - our approach

The main idea of the algorithm is that if there is a sequence of transitions such that *bad* is reachable then we found a counterexample. If it is the case, we have to check whether it is a spurious counterexample that was received due to the low precision of the abstraction. Hence, we refine the abstract domain by enlarging the parameter D. If P_i does not intersect with *bad* then we excluded all spurious counterexamples and the precision of the abstraction is enough to prove the system safe.

The forward search is analogical to the forward search represented in the article [14]. On top of that, we store a sequence of transitions the analysis was going through during one particular forward run, and sets P_i, for $i < n$, representing the abstract states. They will be used in the backward analysis for examination of counterexample runs.

6.4 Backward analysis

Since the forward run is performed within the abstraction domain, the spurious counterexamples can be discovered. Hence, it is necessary to find a way how to increase the precision of the abstract domain enough to solve the coverability problem. The refinement technique is based on the analysis of the configurations from which the set of incorrect configurations *bad* is reachable.

Consider the backward analysis described in Algorithm 1. The algorithm is given an initial state x_0, the set of incorrect configurations *bad*, and a PN N. The backward analysis explores the state space using an operation *minpre* (4.10), searching for configurations from which the set *bad* is reachable. Nevertheless, not only one counterexample path is

inspected during the backward analysis but we follow all transitions whose postconditions the inspected configuration fulfills. Thus, several transitions can be fired backward from a single configuration. This technique is called breadth-first search. Hence, in each iteration of CEGAR, a covering set of the set *bad* (the current approximation of an invariant) is enlarged with a large number of states (see 6.2).

Our modification of the backward analysis outlined in Section 4.1 is based on a global view of the run. It has two steps. First, we traverse the state space of a system in a depth-first manner, rather than breath-first, and construct a minimal abstract run as we discuss in Section 5.1, and then we analyze the run to determine a minimal reason for that the run is spurious.

To analyze the run, we use the sets of abstract states P_i, for $i < n$, and *path* received in a forward analysis (Algorithm 4). It is a sequence of transitions $path = t_1 t_2 \dots t_n$ where n is a length of the run. The sequence represents a run in an abstract domain parametrized by D from an initial set I to the set *bad*. We have to analyze whether the run corresponding to *path* is feasible in a concrete domain as well or whether the analysis has hit a spurious counterexample due to the abstraction.

To do that, we construct a graph which records how exactly were the elements of P_i, $i < n$ (including an abstraction of *bad*), generated by the forward abstract run. An element e is generated in the i-th step of the run if it results of firing the transition t_i from a concretization $\gamma[D](P_{i-1})$ of P_{i-1}. It means that P_{i-1} contains the abstraction $\alpha[D](^\bullet t_i)$ of the precondition of t_i and the element is covered by the postcondition $t_i^\bullet$ of t_i, i.e $t_i^\bullet \succeq e$. To reflect this, the graph will have an edge from the element to the precondition and from the precondition to each element of the abstraction of the precondition. The construction of the graph is described in the following pseudocode (Algorithm 5).

Algorithm 5 *Construction of the graph*

Input:

- $G = (V, E)$ where $V = \alpha[D](bad)$ and $E = \emptyset$

for $i = n$ *to* 1 **do**
  if $\alpha[D](^\bullet t_i)$ *in* P_{i-1} **then**
    for v *in* V **do**
      if $t^\bullet \succeq v$ **then**
        $V = V \cup \{^\bullet t_i\} \cup \alpha[D](^\bullet t_i)$
        $E = E \cup \{(v, \alpha[D](^\bullet t_i))\} \cup \{(v, x) | x \in \alpha[D](^\bullet t_i)\}$
      end if
    end for
  end if
end for

As discussed in Section 5.1, we focus on an analysis of a *minimal run* which corresponds to a way how *bad* can be reached from the initial set. It is a minimal subgraph of G such that it is a DAG, its root is an abstraction of *bad*, and all its leaves are in an abstraction of the initial set (see Figure 6.3). Our aim is to refine the abstraction domain so that we will not generate the run in the future. This can be done by preventing the abstraction from generating some of the preconditions of the transitions used in the run (in the graph

they appear as the nodes ${}^\bullet t_i$ inserted on **line 5** of the Algorithm 5). The refinement can be achieved by including all the preconditions into the parameter D of the abstract domain.

Nevertheless, it is not necessary to include all the preconditions into the parameter since usually only a certain small set of subwords of the preconditions is necessary for the exclusion of the run. Finding the smallest set of such subwords efficiently is an interesting and difficult task that will be the subject of our future research. Our aim in this work is only to confirm that succinct invariants can be indeed discovered based on an analysis of minimal runs.

For this purpose, we implemented a simple and naive method for the analysis of minimal runs. From each minimal run, we select a subset of subwords of the preconditions on random bases. We use several selection strategies which will be described in detail in Section 8.

Kapitola 7

Implementation

To prove the existence of more succinct invariant we have implemented a tool based on the method presented in Section 6. The tool serves as a prototype. In this chapter, an implementation of Petri net is discussed followed by a description of the implementation of the forward and backward analysis.

The tool is implemented in Python. Python is a scripting and a highly readable language which offers simply and general syntax. Moreover, it provides the abstraction capabilities needed to express the Petri net model. Since variables are neither declared in the program, nor generating by the compiler, their types need to be checked at run time which has a negative impact on performance for data intensive computation.

7.1 Requirements

We require from the tool that fulfills the following requirements in order to perform the computation required to prove the existence of more succinct invariants. The tool is necessary to be able:

- to load parameters of Petri net from an input file in format of mist2[2],
- to create a Petri net based on a given set of places, transitions and rules,
- to abstract a configuration based on the equation (4.1),
- to perform forward simulation within an abstract domain based on the equation (4.4),
- to perform a depth-first search analysis of counterexample runs,
- to determine a minimal reason which refutes the examined spurious counterexample run, and
- to generate a file in a format required by *Graphviz*[3].

[2] http://www.cprover.org/bfc/
[3] http://www.graphviz.org/

7.2 Processing the input file

Since the requirement was to implement a tool with an ability to verify a set of benchmarks we decided to create an interface for loading input files in format of *mist2*.

For the inputs in the experiments, we use the collection of Petri net examples from the MIST toolkit in format *mist2*. The format *mist2* is defined as:

```
 # comments
S ::= vars
     set_of_vars
     rules
     set_of_rules
     init
     init_states
     target
     target_states
     invariants
     invariant_set
set_of_vars::= variable , set_of_vars
                 | variable
set_of_rules::=  guard -> effect;  set_of_rules
                 | epsilon
guard ::= guard_atom , guard
        | guard_atom
guard_atom ::= variable >= integer
             | variable =integer
             | variable in [integer, integer]
effect ::= effect_atom , effect
            | effect_atom
effect_atom ::= variable' = assignment_var
assignment_var ::= variable + assignment_var
                 | integer
init_states ::= guard init_states
              | guard
target_states ::= guard target_states
                | guard

invariant_set ::= invariant EOL invariant_set
                | epsilon
invariant ::= invariant_atom , invariant
                | invariant_atom
invariant_atom::=variable = integer
```

Let us remark that *EOL* is a sign for the end of line. As $effect\_atom$ we consider a rule in form $x' = x + c$, $x' = x - c$ or $x' = c$ specifying the invariant where c is the coefficient of variable x. The rules correspond to Petri net transitions or to Petri net extension transitions [13].

The function $loadFile$ loads input files, processes them and creates corresponding

structures required by the tool. The function defines regular expressions for each type of the input information:

- **variables** - represent places of PN (states of the system),
- **rules** - transitions between places (state changes of the system),
- **initial marking** - an initial marking of PN (an initial configuration of the system),
- **targets** - incorrect markings of PN (incorrect configurations of the system), and
- **invariants**.

The function is based on an automaton. It parses an input file and if one of the key words is observed then the following lines are parsed using the relevant regular expression corresponding to the type of the information. In this way, an initial state, targets, variables and invariants are loaded.

Different approach is used in case of rules since they have more complex format. First, the relevant lines are selected and split into single rules. Subsequently, they are split again into a left and right part. Each part of the rule is processed separately since the left part of the rule indicates the required pre-conditions of the rule while the right part of the rule determines how the marking of PN is changed.

The retrieved parameters of PN from the input file are stored in structures **variables** (states), **bad** (a set of incorrect configurations), **x0** (an initial configuration), **invariants** and **transitions**.

7.3 Petri net

Each PetriNet object is an instance of the class PetriNet. PetriNet instance contains a list of names (states of the system), a list of transition names, a list of Transition objects, and the current labeling. States of the system are represented by their names as strings ("$x0$", "$q1$", "$T0$", ...). The labeling is a dictionary from a name of the state to a token count.

As an example, let us consider the incorrect configuration in Peterson's algorithm - $q_4 r_4$. The corresponding labeling of PN is the following:

```
bad = {"q1":0.0,"q2":0.0,"q3":0.0,"q4":1.0,"r1":0.0,"r2":0.0,"r3":0.0,
"r4":1.0,"x0":0.0,"x1":0.0,"y0":0.0,"y1":0.0,"T0":0.0,"T1":0.0}.
```

Each Transition object contains a transtion name, an input and an output dictionary. The input/output dictionary map the name of a state to the number of tokens that the transition takes as input/give as output. The class Transition contains two methods - *IsEnabledBack* and *FireBack*.

The method *IsEnabledBack* takes a labeling of PN as its input and return *true* if the transition can be fired or *false* otherwise. It compares a number of tokens in the labeling to a number of tokens requiring as post-condition of the transition. The method *FireBack* takes a labeling of PN as its input and modifies a number of tokens to simulate firing of the transition.

PetriNet also inherits methods from a class PetriNetBase consisting of names of the transitions and two methods - *BuildTransitions* which creates the object for each defined transition and *SetDegree* which sets post-condition and pre-condition labeling required for each transition.

The PetriNet class includes besides others the following methods:

- **StringToLabel** - takes a string (represented the configuration of a state) and creates a dictionary with corresponding labeling of PN,
- **LabelToString** - takes a dictionary as an input and converts it into a string according to the configuration that the dictionary represents. Considering the previous dictionary representing the state *bad* as an input the function returns a string $q_4\_r_4$.
- **LabelToList** - takes a dictionary as an input and coverts it into a list of all letters included. Considering the previous dictionary representing the state *bad* as an input the function returns a list $[q_4, r_4]$.

7.4 Abstraction

The abstraction is implemented based on the equation (4.1). We iterate over the parameter D of the abstract domain and test whether the configurations from D are in the DCS of the upward-closure of the configuration c. The abstraction of the initial state x_0 is implemented as follows:

```
this.invariant = x0
for item in dSet:
   if(this.CheckInvariant(item)):
      pSet += [item]
```

where a procedure *CheckInvariant* returns *True* if *item* is in the UCS of the upward-closure of x_0, and *False* otherwise. The procedure *CheckInvariant* is implemented as follows:

```
def CheckInvariant(this, label):
   for index in label:
      if label[index] > this.invariant[index]:
         return False # invariant invalid
      return True  # invariant valid
```

Given x_0 representing by a word q_4r_4 and D representing by a set of words $\{q_3, q_4, r_3, r_4, q_4r_3\}$ as an input the abstraction procedure returns a subset $\{q_4, r_4, q_4r_4\}$.

7.5 Forward search

In Section 6.3, we have introduced the algorithm of the forward search. The forward search is equivalent to the forward search presented in Section 4.1. The only modification consists in storing the sequence of fired transitions during the search in *path*.

A function *Forward* is implemented based on the equation (4.5). The forward search can be divided into several parts:

[Initial abstraction] Function takes the initial configuration x_0 as its input and returns a subset of D. The abstraction is described in detail in Section 7.4. The whole subset is included to the set P.

[Complement recalculation] The next step is creating a complement $D\backslash P$. According to the equation (4.5), the valid predecessors of x are the configurations outside of the complement.

[Forward step] To determine a post-image of P, we iterate over the parameter D and test which of the configurations are able to reach the set P in one step backward. For each configuration x from D we fire all transitions using a procedure $FireBack$. If the predecessor is valid (we check this property using a procedure *CheckInvariant*) and it is not included yet in D then x is a valid successor of P and is added into P.

7.6 Backward analysis

In Section 6.4, we presented the optimized modification of the backward analysis. Since we have implemented both algorithms we introduce first the backward analysis of the original version.

Raskin tool We have implemented the backward analysis based on the equation (4.10). We start with the whole set *bad* that is stored in a list $dBad$. The list stores predecessors of the incorrect configurations from *bad*. i-th item of the list represents a set of predecessors of the configurations from the previous level.

In each iteration of CEGAR, the backward analysis takes one step backward using a procedure $FireBack$. The predecessor is proved valid if:

- is not already included in D,
- is not in a complement $D \backslash P$, and
- is not in an DCS of the down-ward closure of the initial state x_0 (last two properties are checked by the procedure $CheckInvariant$).

If the initial set is reached then the system is proved unsafe. This property is checked by testing whether the inspected configuration is in the DCS of $x0$ by a procedure $CheckInvariant$.

Our prototype The modified version of the backward analysis was presented in Section 6.4. The backward analysis is divided as well as the forward search into several parts:

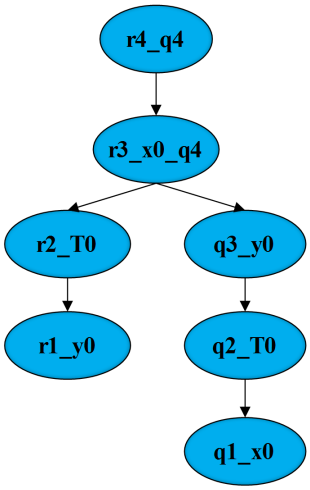
Initial abstraction To check whether the initial set is reached, we apply abstraction to $x0$ within the set P as we presented in Section 7.4. The result of the abstraction is stored in $initAbs$.

Initialization The backward analysis starts from a state *badState* representing the abstraction of the incorrect configuration from the set D which was hit during the forward search. We select the relevant transition from the sequence of transition *path*. Both the *badState* and its predecessor are stored in a list $treePre$.

AbsConfiguration Since the backward analysis starts from an abstract set the procedure *AbsConfiguration* is called. It returns a set of configurations from D that represents the abstraction of the predecessors.

Step backward For each configuration from the current level of the analysis we fire a corresponding transition selected from the sequence *path*. Since the discovered configurations are configurations from the abstract set P we can easily check whether the discovered configuration is in the abstract set of the initial set $initAbs$.

Creating the set of possible configurations The result of the backward analysis is the list $treePre$ containing new-found configurations. We create a list of all these configurations and their sub-configurations. The sub-configurations are created using a procedure *AllSubwords*.



Obrázek 7.1: Small graph

If they are not included already in D they will be a part of the set *newSet*. One of the configuration of the set *newSet* is randomly selected and added to D.

7.7 Creating graphs

During the backward analysis, the configurations and their predecessors are stored in order to be written to an output file in a format required by *Graphviz*. *Graphviz* is open source graph visualization software which represents structural information as as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains. he Graphviz layout programs take descriptions of graphs in a simple text language, and make diagrams in useful formats, such as images and SVG for web pages; PDF or Postscript for inclusion in other documents [6]. Figure 7.1 illustrates the example of the minimal run using our approach.

Kapitola 8

Evaluation

Based on our method from Section 6, we have implemented a prototype in Python to find invariants for checking safety properties for parameterized programs communicating via shared variables and mutexes. We have implemented also the original version of the Algorithm 2 (GRB) and compared its performance to our implementation in experiments on a set of Petri net benchmarks [2]. The input files are in format of *mist2* (see Section 7.2).

8.1 Results

The goal of the evaluation was to compare a size of the inductive invariant. We measured a number of iterations of CEGAR as well. The experiments were based on Petri net examples from the *mist2* distribution. They contain Petri nets modeling for instance a variation of classical example of concurrent readers and writers (*rw*), a mutual exclusion protocol such as Peterson's exclusion algorithm (*Peterson*), or a PN model of asynchronous programs (*pingpong*).

We implemented several strategies of refinement of the parameter D. They are all based on choosing a subset of subwords of preconditions of a minimal run (Section 6.4). We were choosing the refinement of D as the set of:

1. all preconditions,
2. all subwords of preconditions with the maximum length 2, 3 or 4,
3. one randomly selected subword from the set of preconditions, or
4. a subword of a precondition very close to the beginning or end of the minimal run.

Using completely random method (method 3) leads to finding the most succinct invariant in most of the cases. The disadvantage of the random method is that it needs many iterations of CEGAR and has unpredictable results. In the Table 8.1, we show the results of the most successful strategies.

Table 8.1 presents results for both algorithms. The column on the left shows the benchmark while the columns on the right show the size of invariant (a number of elements) and the number of CEGAR iterations. The last column shows the percentage reduction in the size of the invariant compared to GRB. A positive value shows a reduction while a negative value an increase in the size of the inductive invariant.

Benchmark	GRB		Our Approach		Reduction
	Size of invariant	Iterations	Size of invariant	Iterations	
basicME	45	5	22	19	44%
rw	331	8	36	35	90%
Peterson	135	8	107	21	21%
newrtp	9	9	56	53	-20%
pingpong	10	10	28	27	65%

Tabulka 8.1: Comparison between GRB and our approach

The set of Petri net benchmarks of *mist2* contain more examples than presented but our method using the random strategy did not terminate in the given time limit and the other strategies did not give results comparable to GRB.

8.2 Discussion

Our results confirm that method of analyzing minimal runs can be used for finding a more succinct invariant than which can be found using the method GRB. The difference can be seen at example of Peterson's mutual exclusion algorithm. While the method GRB found invariant of size 135 our approach discovered that there can be even smaller one of size 107. However, the price for a smaller invariant was an increase in the number of iterations required to find it. In Section 5.1, we presented even smaller invariant of size 9. This indicate that there is still much space for improvement which we hope to harvest by using more sophisticated methods of analyzing minimal runs.

Kapitola 9

Conclusion

In the first part of this thesis, we have introduced state-of-the-art methods [9, 14, 3] for automatic verification of systems with parallel running processes. The presented methods are all based on the effort to find an inductive invariant of the system. Based on our study, we conjecture that they would benefit from focusing on finding the smallest possible inductive invariants.

We have chosen the methods presented in [14] as the most suitable for future research and testing our ideas. The method discovers an inductive invariant using a variant of counterexample-guided abstract refinement. The original version of the algorithm uses breath-first search to analyze counterexamples. This causes that a huge number of elements are added into the approximation of invariant, making it unnecessary verbose. Our modification is based on analyzing counterexamples in a death-first manner, searching for so called minimal counterexample runs.

We implemented the original method as well as our modified version. Even though we use a naive method for analysis of minimal runs, our experimental results confirm that our approach can indeed give smaller invariants. This result will motivate our future research which will focus on a development of a truly efficient method for analyzing minimal runs. We believe that this will lead to a design of a method for analyzing parallel systems much better than the state-of-the-art.

This work participated in the student conference Excel@FIT 2015 and was awarded with the first place in the category “Scientific contribution” and the second place in the category “Technical level”.

Literatura

[1] Aaron R. Brandley: SAT-Based Model Checking without Unrolling. *Verification, Model Checking, and Abstract Interpretation*, ročník 6538, 2011.

[2] Alexander Kaiser, Daniel Kroening, Thomas Wahl: Bfc - A Widening Approach to Multi-Threaded Program Verification. [online]. `Available on: http://www.cprover.org/bfc/`.

[3] Alexander Kaiser, Daniel Kroening, Thomas Wahl: Efficient Coverability Analysis by Proof Minimization. *CONCUR 2012 - Concurrency Theory*, ročník 7454, 2012.

[4] Daniel J. Sorin, Mark D. Hill, David A. Wood: Dynamic verification of end-to-end multiprocessor invariants. *Proceedings International Conference on Dependable Systems and Networks*, 2003.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith: Counterexample-guided abstraction refinement for symbolic model checking. *2003 International Conference on Dependable Systems and Networks*, ročník 50, č. 5, 2003.

[6] Envisioning connections: Grapviz - Graph Visualization Software. [online]. `Available on: http://www.graphviz.org/`.

[7] Jan Holeček: Petriho sítě. [online]. `Available on: http://www.fi.muni.cz/usr/kucera/teaching/pn/pnets.pdf`.

[8] Javier Esparza, Rupak Majumdar, Filip Niksic, Ruzica Piskac: Decidability and complexity of Petri net problems - an introduction. *Lectures on Petri Nets I: Basic Models*, ročník 1491, 1998.

[9] Johannes Kloos, Rupak Majumdar, Filip Niksic, Ruzica Piskac: Incremental, Inductive Coverability. *Lecture Notes in Computer Science*, ročník 8044, 2013.

[10] Marco Sgroi: Petri nets. [online]. `Available on: http://www.zemris.fer.hr/predmeti/fpors/pred/Notes/Petri`.

[11] Parosh Aziz Abdulla: Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, ročník 16, č. 4, 2010.

[12] Parosh Aziz Abdulla, Frederic Haziza, Lukáš Holík: All for the Price of Few. *Verification, Model Checking, and Abstract Interpretation*, 2013.

[13] Pierre Ganty: Coverability Checkers included in mist. [online]. `Available on: https://github.com/pierreganty/mist/wiki`.

[14] Pierre Ganty, Jean-Francios Ganty, Lurent Van Begin: A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. *Verification, Model Checking, and Abstract Interpretation*, ročník 3855, 2006.

[15] Ronald Beyer, Stefan Lowe: Explicit-State Software Model Checking Based on CEGAR and Interpolation. *Fundamental Approaches to Software Engineering*, ročník 7793, 2013.

[16] Stanley Gill: Parallel Programming. *The Computer Journal*, ročník 1, č. 1, 1985: s. 2–10.

[17] Tomáš Vojnar: Formal Analysis and Verification. [online]. `Available on: http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-01.pdf`.

Příloha A

Storage Medium

A storage medium containging an electronic version of the thesis and source code of the implemented tools.