Concurrency theory

Lecture notes

December 4, 2018

Roland Meyer

Sebastian Muskalla

Prakash Saivasan

TU Braunschweig

Winter term 2017/2018

Contents

1 Introduction **4**

I Petri nets and well-structured transition systems **9**

2 Petri nets **10**
- Syntax and semantics of Petri nets 10
- Algorithmic problems 16

3 Petri net coverability **22**

4 Rackoff's algorithm for coverability **26**

5 Lipton's hardness result **36**

6 Petri net reachability **52**
- Generalized Markings 53
- Covering graphs 54
- Precovering graphs 59
- Marked graph transition sequences 67
- Decomposing MGTS 72

II Weak memory models **81**

7 Total store ordering **82**

8 TSO reachability **89**

9 TSO reachability in a bounded number of rounds **101**

10 Robustness against TSO **109**
- Traces and trace-based robustness 110
- Minimal violations and locality 116
- Instrumentation 123

References **127**

Preface

These are the lecture notes accompanying the course "Concurrency theory" taught at TU Braunschweig in the winter term of 2017/2018.

Unfortunately, we cannot guarantee the correctness of these notes. In case you spot a bug, please send a mail to us: `s.muskalla@tu-bs.de`.

Roland Meyer, Sebastian Muskalla, Prakash Saivasan

Braunschweig, December 4, 2018

Literature

The content of this lecture overlaps with the contents of past iterations of "Concurrency theory". The lecture notes for parts of the lecture are based on Roland Meyer's notes, in particular on his texed lecture notes from 2011:

tcs.cs.tu-bs.de/documents/lecturenotes/conctheo2011.pdf

For the rest of the lecture, we mostly use the original papers as sources. The beginning of each section will contain information on the material on which the content is based.

1. Introduction

The overall topic of this lecture is the verification of concurrent systems. We will approach this by considering types of automata that can model the behavior of such systems and solving their algorithmic problems.

Concurrent systems

A **concurrent system** is a collection of components (e.g. threads)

- running asynchronously or synchronously,
- running concurrently, e.g. interleaved on one core, on several cores of one CPU, on several CPUs of the same machine or one multiple machines (distribution), and
- communicating in some way, e.g. via shared memory or messages.

Verification

Verification is one of the biggest active research areas within Theoretical Computer Science. Its most basic problem is the verification problem: Given a system *Sys* and a specification *Spec*, does the behavior of the system satisfy the specification, $Sys \models Spec$? The difficulty of this problem arises from the fact that one usually only has a syntactic description of the system (e.g. the source code of a program), but the specification talks about the runtime behavior of the system. Even in the simple case where the system is a sequential Java program and the specification is given by a designated error location in the source code that should not be reached, the problem is undecidable. (It corresponds to the control state reachability problem for Turing machines, a variant of the undecidable halting problem.)

Research in verification tackles this problem in various ways.

There are *semi-decision procedures* for verification that do not always terminate, but if they terminate, their result is correct. For example, to solve the above problem, one can enumerate possible computations of the program. If one finds a valid computation reaching the error location, the answer to the verification problem is negative.

Approximation techniques replace a complex system by a system from a simpler class for which the verification problem is decidable. For example, one can model a recursive program, more precisely its control flow, by a pushdown system, for which control state reachability can be checked in polynomial time. There are two types of approximations:

- An *overapproximation* is a system whose behavior subsumes the behavior of the original system. If one is able to prove an overapproximation to be sound, i.e. all possible behaviors satisfy the specification, this also holds true for the original system. If the overapproximation has a bad computation, it may not be clear whether this computation is also a computation of the original system.
- An *underapproximation* is a system whose behavior is a subset of the behavior of the original system. If one is able to find a bad computation of an underapproximation, then also the original system has a bad computation.

All these techniques may be combined. Some semi-decision procedures iteratively refine approximations of the systems until they can find a bad computation or prove the system correct. Still, it may happen that procedure does not terminate since none of the two cases applies within a finite number of steps.

For approximation techniques, it is crucial that efficient techniques are available for the algorithmic problems of the class of systems that is used to approximate. In this lecture, we will therefore consider types of automata that can be used for modeling concurrent systems and discuss their algorithmic problems.

Models for concurrent systems

(1) Automata on a distributed alphabet. Assume the simple case in which each component of the system is modeled by a finite automaton. If all components are synchronized by some external clock, the system can be modeled by the product automaton. This automaton can be explicitly constructed and algorithms for finite automata can be applied.

If we assume that the automaton run asynchronously, interesting behavior can occur. Consider for example an automaton over the alphabet $\{a, b\}$ generating ab and an automaton over the alphabet $\{c, d\}$ generating cd. If we give their parallel composition an **interleaving semantics**, we obtain that it can generate all possible interleavings of ab, cd, i.e. the set of words $\{abcd, acbd, acdb, cdab, cadb, cabd\}$. Now assume that there is some action s that is in the alphabet of both automata on which the automata synchronize. If the automata generate asb resp. csd, the set of possible interleavings is reduced to $\{acsbd, acsdb, casbd, casdb\}$. Research in this area studies the structure of languages generated by such automata over **distributed alphabets** that run asynchronously, but synchronize on some actions.

(2) Multi-pushdown systems. A multi-pushdown system is a system with several stacks that can be used independently. Such systems occur if we assume that each

component of a concurrent system is modeled by a pushdown system, and we then construct the product. A multi-pushdown system is much more powerful than a regular pushdown system; Unfortunately, it is already too powerful. The tape of a Turing machine can be modeled by two stacks, the one containing the part of the tape that is on the left of the read-head, the other containing the part on the right. Moving the head can be implemented by popping from one stack and pushing onto the other. Consequently, multi-pushdown systems are Turing-complete and all interesting problems, e.g. control state reachability, are undecidable.

To overcome this problem, one may only look at a restricted set of computations. For example, one may consider **bounded context switching**, i.e. one only considers computations that can be split into k phases (where k is some fixed number) such that in each phase, only one of the stacks is used. This corresponds to restricting the communication between the components of the concurrent system. Research in this area focuses on understanding which restrictions lead to verification problems becoming decidable.

(3) Petri nets. Petri nets are an automata model in which a concurrent system can be modeled natively (i.e. without taking the product of several systems). Petri nets are the first topic that we will consider in the lecture. We refer the reader to Section 2 for a detailed explanation and an example.

(4) Well-structured transition systems. Well-structured transition systems (WSTS) are a general class of systems to which some algorithmic techniques for Petri nets can be extended. The idea is to order the state space such that larger states have a richer behavior. An important example of WSTSs that are not Petri net-like are **lossy channel systems**.

A (perfect) channel system is a system that uses LIFO-queues (last in, first out) as storage. The transitions of the system can perform enqueue and dequeue actions. Unfortunately, perfect channel system are Turing-complete. In a lossy channel system, we assume that at any point in time, the channel may lose some of its content.

Network protocols can be modeled as a lossy channel system, because for any network communication (e.g. TCP/IP), one has to assume that packages can be lost. A protocol should be correct even if package losses occur.

(5) Weak memory models. Consider the following parallel program, a simplification of the so-called **Dekker mutex**.

$x = y = 0.$

$x = 1;$	$y = 1;$
`if`$(y == 0)$ {	`if`$(x == 0)$ {
//critical section	*//critical section*
}	}

If we assume that the program is executed under an **interleaving semantics**, **mu**tual **ex**clusion holds. At most one thread can enter the critical section: As soon as one thread signals that it wants to enter the critical section by setting x resp. y to 1, the other thread will see this and cannot enter anymore.

Such an interleaving semantics corresponds to a **strong memory model** like **sequential consistency (SC)** in which all writes made by one thread are instantly visibly to all other threads. This is not feasibly in practice, as it would essentially slow down the speed of execution to the speed of the communication between the threads. Even when we assume that both threads runs on two CPUs of the same machine, this may lead to a 90% decrease of performance (e.g. the Intel Xeon E3-1285V6 processor has up to 4.5 GHz, but DDR4-3200 SDRAM has only 400 MHz memory clock).

This problem is solved by introducing buffers: A write will not be directly written to the main memory, but it will be buffered. At some later point, the buffer content will be batch-processed into the memory and then become visible to the other threads. Theoreticians model this by introducing **weak memory models**, e.g. the **total store ordering (TSO)** as memory model for the x86 architecture. A write command is split into the issue-event and the store-event, where the latter marks the point in time when the write has landed in main memory.

Under a weak memory model, programs that are correct under SC might become incorrect. In the above example, mutual exclusion does not hold anymore. Consider the following execution:

1. Left thread issues the write $x = 1$
2. Right thread issues the write $y = 1$
3. Left thread reads $y = 0$ from the main memory and enters the critical section
4. Right thread reads $x = 0$ from the main memory and enters the critical section
5. The write $x = 1$ is stored in the main memory

6. The write $y = 1$ is stored in the main memory

To rule out this unwanted behavior, some sort of synchronization has to be enforced. x86-Assembly provides a memory fence command (`MFENCE`) that makes the execution of a thread stop until all its writes have been stored in the main memory. Inserting a memory fence after the write of each thread fixes the example.

This leads to two interesting questions for researchers in theory:

- Can the behavior of programs executed under a weak memory model (e.g. with delayed stores and memory fences) still be verified? How does the complexity of the verification problem change when going from strong to weak memory models?
- Understanding the behavior of a parallel program under interleaving semantics/a strong memory model is already difficult. Can one prove that if the program satisfies some conditions that are easy to understand by a programmer, its behavior under a weak memory model is the same as the behavior under a strong memory model?

The latter question is of particular interest because (1) a programmer cannot be expected to know neither the internals of the implementation of the architecture nor the theory on memory models and (2) verifications tools – even if they exist – are usually to slow to be applicable to large-scale software systems.

Part I.
Petri nets and well-structured transition systems

2. Petri nets

We introduce the syntax and semantics of Petri nets and some algorithmic problems that we want to solve in the next sections.

Sources
This content of this section can be found in any standard textbook on Petri nets, e.g. [Rei85]. The presentation of Petri nets chosen here differs a bit from the one that is commonly used in the literature, see below.

Syntax and semantics of Petri nets

2.1 Definition: Petri nets
A **Petri net** is a tuple $N = (P, T, in, out)$ where

- P is a finite set of **places**,
- T is a finite set of **transitions** with $P \cap T = \emptyset$, and
- the functions

  $$in, out \colon T \to P \to \mathbb{N}$$

  assign to each transition $t \in T$ a vector $in(t)$ resp. $out(t) \in \mathbb{N}^P$ of **incoming resp. outgoing multiplicities**. For a transition $t \in T$ and a place $p \in P$ the incoming multiplicity $in(t, p)$ is the multiplicity of the arc from p to t. Similarly, the outgoing multiplicity $out(t, p)$ is the multiplicity of the arc from t to p.

2.2 Remark
A function of type $P \to \mathbb{N}$ that assigns each place a number can be seen as a vector in $\mathbb{N}^P$. A function of type $T \to P \to \mathbb{N}$, which is shorthand for $T \to (P \to \mathbb{N}$ can be equivalently seen as a function of type $T \times P \to \mathbb{N}$ or as a function of type $T \to \mathbb{N}^P$. We may also see it as matrix in $\mathbb{N}^{T \times P}$ having one entry for each transition and place.

We obtain a graphic representation of a Petri net as follows: We draw places as circles and transitions as boxes. If $in(t, p) \neq 0$ for some t, p, we draw an arc from p to t, and label this arc by $in(t, p)$. Similarly, we draw an arc from t to p if $out(t, p) \neq 0$, and we label it with this number if $out(t, p)$. If the multiplicity is 1, we often omit the label of the arc.

2.3 Example

Consider the Petri net $N = (P, T, in, out)$ with

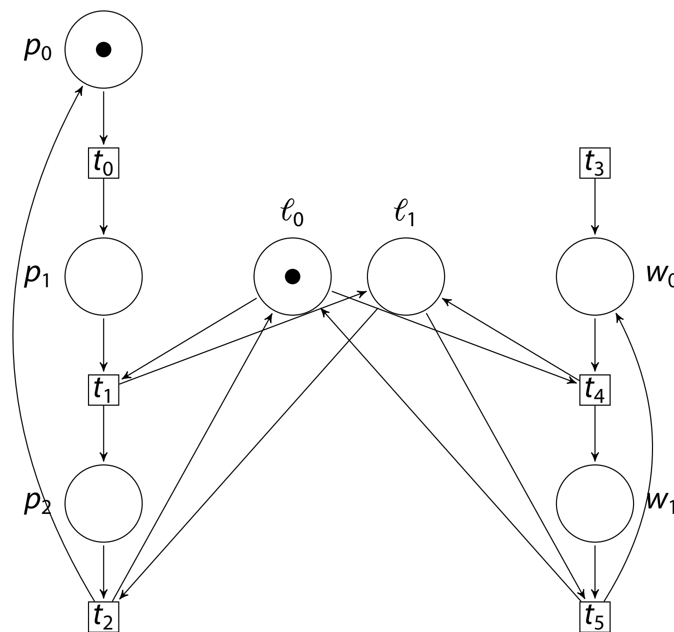
$$P = \{p_0, p_1, p_2, \ell_0, \ell_1, w_0, w_1\},$$
$$T = \{t_0, t_1, t_2, t_3, t_4, t_5\},$$

and in, out specified by the following tables.

in	t_0	t_1	t_2	t_3	t_4	t_5
p_0	1	0	0	0	0	0
p_1	0	1	0	0	0	0
p_2	0	0	1	0	0	0
ℓ_0	0	1	0	0	1	0
ℓ_1	0	0	1	0	0	1
w_0	0	0	0	0	1	0
w_1	0	0	0	0	0	1

out	t_0	t_1	t_2	t_3	t_4	t_5
p_0	0	0	1	0	0	0
p_1	1	0	0	0	0	0
p_2	0	1	0	0	0	0
ℓ_0	0	0	1	0	0	1
ℓ_1	0	1	0	0	1	0
w_0	0	0	0	1	0	1
w_1	0	0	0	0	1	0

The following figure gives a graphical representation of this Petri net.



This Petri net actually represents a concurrent system, we will explain this later in Example 2.5.

2.4 Remark

The multiplicities are also called **weights** in the literature.

In the literature, usually *in* and *out* are combined into a single **flow matrix**

$$F\colon (T \cup P) \times (T \cup P) \to \mathbb{N}$$

such that $F(p,t) = in(t,p)$ is the incoming weight, and $F(t,p) = out(t,p)$ is the outgoing weight. With this view, a Petri net is a triple (P,T,F). We can easily convert one representation into the other and will use them interchangeably during the lecture.

2.5 Example

The flow matrix for the Petri net from Example 2.3 is the following.

F	t_0	t_1	t_2	t_3	t_4	t_5	p_0	p_1	p_2	ℓ_0	ℓ_1	w_0	w_1
t_0								1					
t_1									1		1		
t_2							1			1			
t_3												1	
t_4											1		1
t_5										1		1	
p_0	1												
p_1		1											
p_2			1										
ℓ_0		1			1								
ℓ_1			1			1							
w_0					1								
w_1						1							

Here, the entry of the cell in row x and column y contains $F(x,y)$. If the cell is empty, the corresponding value is zero.

We are now able to define the semantics of Petri nets. This includes defining the possible configurations of a Petri net and their computational behavior, i.e. how computations may lead from one configuration to another. We start by defining the configurations.

2.6 Definition: Marking

A **marking** of a Petri net $N = (P,T,in,out)$ is a vector $M\colon P \to \mathbb{N}$ that assigns each place $p \in P$ a number $M(p)$ of **tokens**.

Petri nets differ from other automata models that you may know (finite automata, Push-down automata, Turing machines) in that their configurations do not consist of a control state. This reflects the fact that they were designed to model concurrent systems: A Petri net does usually not represent a single program, but a collection of interacting components. Each component is in some state, which can be represented by considering markings in which several places carry a token. We can even represent multiple instances of the same component in the same state by assigning more than one token to a place. We will come back to this when discussing the meaning of Example 2.3.

2.7 Definition: Firing relation
Let $N = (P, T, in, out)$ be a Petri net and let $M \in \mathbb{N}^P$ be a marking for N.

For a transition $t \in T$, we say that t is **enabled** in M if $M \geqslant in(t)$, i.e. for all $p \in P$, we have $M(p) \geqslant in(t, p)$. We write $M\,[t\rangle$ in this case.

An enabled transition can be **fired** leading to the new marking

$$M' = M - in(t) + out(t)\;,$$

i.e. the marking M' with $M'(p) = M(p) - in(t, p) + out(t, p)$. We write $M\,[t\rangle\,M'$ in this case.

Intuitively, firing transition t first consumes $in(t, p)$ many tokens from each place p. The transition being enabled guarantees that every place carries the number of tokens needed. Then, $out(t, p)$ many tokens are produced on each place p.

2.8 Definition: Firing sequence
We extend the notion of firing to sequences: For a sequence $\sigma \in T^*$ of transitions, we write $M\,[\sigma\rangle\,M'$ if firing the transitions in σ successively leads from marking M to marking M'. This implies that for every decomposition $\sigma = \sigma_1.t.\sigma_2$, we have that t is enabled in the marking M_1 with $M\,[\sigma_1\rangle\,M_1$. We call such a σ a **(valid) firing sequence**.

A **computation** of a Petri net is a sequence

$$M_0\,[t_0\rangle\,M_1\,[t_1\rangle\,M_1\,[t_2\rangle\,\dots\,[t_n\rangle\,M_{n-1}$$

of markings and transitions.

We call the vector $e(t) = out(t) - in(t)$ (i.e. the vector $e \in \mathbb{N}^P$ with $e(t)_p = in(t)_p - out(t)_p$) the **effect** of transition t The effect of a transition sequence σ is the sum of the effects of the transitions occurring in σ, $e(\sigma) = \sum_{i=0}^{|\sigma|-1} e(\sigma_i)$ is the effect of σ.

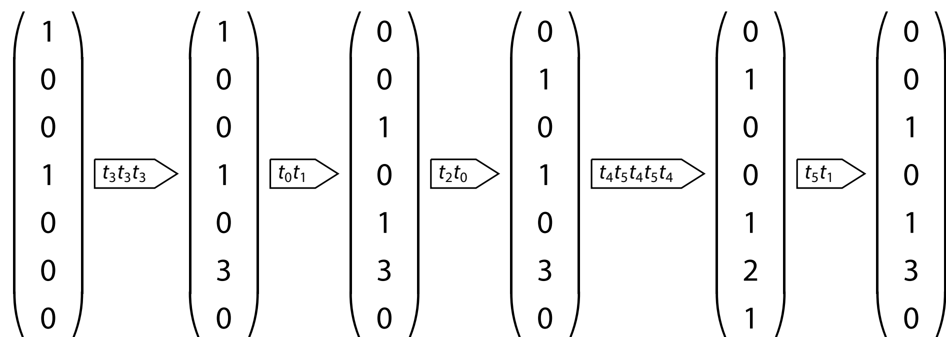
Note that an initial marking together with a valid firing sequence uniquely specifies a computation. Similarly, a sequence of marking specifies a computation if the differences between the markings are the effects of enabled transitions that exist in the net. This will allow us to sometimes to see a computation just as a sequence of transitions or markings instead of a sequence of both.

2.9 Example
We equip the Petri net from Example 2.3 with the marking $M_0 = (1, 0, 0, 1, 0, 0, 0)$, i.e. the marking that assigns one token to p_0 and ℓ_0 and no tokens elsewhere.

Note that the Petri net represents a simple concurrent system:

- The places p_0, p_1, p_2 represent a controller thread. The places w_0, w_1 represent worker threads. The places ℓ_0, ℓ_1 form a **semaphore (lock)**.
- Initially, the lock is not held, as $M_0(\ell_0) = 1, M_0(\ell_1) = 0$. Initially, the controller thread is in state p_0. Initially, there is no worker thread.
- During the net, worker threads can be created by firing transition t_3. This will spawn a new worker thread in state w_0
- The controller may freely move to state p_1 by firing t_0.
- The places p_2 and w_1 are **critical sections** of their respective thread. Only one thread can be in one of those states at a time (i.e. we have at most one token assigned to them). This is ensured because the transitions t_1, t_4 need to take the lock by *moving* the token from ℓ_0 to ℓ_1. The lock is released when the threads leave the critical section by transition t_2, resp. t_5.

In the following computation, we spawn 3 worker threads, let the controller enter the critical section, let each worker thread enter the critical section, and let the controller enter the critical section again.



$$\begin{pmatrix}1\\0\\0\\1\\0\\0\\0\end{pmatrix} \xrightarrow{t_3t_3t_3} \begin{pmatrix}1\\0\\0\\1\\0\\3\\0\end{pmatrix} \xrightarrow{t_0t_1} \begin{pmatrix}0\\0\\1\\0\\1\\3\\0\end{pmatrix} \xrightarrow{t_2t_0} \begin{pmatrix}0\\1\\0\\1\\0\\3\\0\end{pmatrix} \xrightarrow{t_4t_5t_4t_5t_4} \begin{pmatrix}0\\1\\0\\0\\1\\2\\1\end{pmatrix} \xrightarrow{t_5t_1} \begin{pmatrix}0\\0\\1\\0\\1\\3\\0\end{pmatrix}$$

When we discuss algorithms, we will analyze their complexity, i.e. their worst-case memory and time consumption. This analysis will always be in terms of the input size. As in input is formed by Petri nets and markings, we will need to assign a size to these objects.

2.10 Definition

Let N be a Petri net and let M be a marking for N. The size $|M|$ of the marking M is the size of the numbers occurring in M encoded in binary. We assume that each entries needs at least one bit.

$$|M| = \sum_{p \in P} \lceil \log M(p) \rceil + 1 \, .$$

Similarly, the size $|N|$ of the Petri net N is the encoding of the ingoing and outgoing multiplicities in binary.

$$|N| = \sum_{t \in T} \sum_{p \in P} \lceil \log in(t,p) \rceil + \lceil \log out(t,p) \rceil + 2 \, .$$

2.11 Remark

- We assume a *dense* encoding here, as we represent each entry using at least one bit. In a *sparse* encoding, we would only measure non-zero entries.
- For a marking M, let m be its maximal entry, $m = \max_{p \in P} M(p)$. We have

  $$|M| \in \mathcal{O}(|P| \cdot \lceil \log m \rceil + 1) \, .$$

  Similarly, let m' be the maximal multiplicity of any arc in the Petri net N,

  $$m' = \max_{t \in T, p \in P} \max\{i(t,p), o(t,p)\} \, .$$

  We have

  $$|N| \in \mathcal{O}\big(|P| \cdot |T| \cdot \lceil \log m' \rceil + 1\big) \, .$$

- A Petri net of polynomial size can have exponential multiplicities, as 2^n can be encoded in binary using n bits. This will play an important role when we analyze algorithms.

  If one considers an **unary encoding** of markings and multipliticies, i.e. we would define $|M| = \sum_{p \in P} M(p) + 1$, we would get different complexity results.

Algorithmic problems

The most basic algorithmic problem is **reachability**.

2.12 Definition

Petri net reachability (PNREACH)

Decide: Petri net N, initial marking M_0, final marking M_f

Decide: Is there a firing sequence $\sigma \in T^*$ such that $M_0 \xrightarrow{\sigma} M_f$?

Petri net reachability is known to be decidable, but there is no algorithm that is known to have primitive-recursive complexity. This means that all known algorithms need unimaginable running times in the worst case, even for tiny examples. Please read Remark 6.1 for a more detailed discussion.

2.13 Remark

Usually, a Petri net is considered in conjunction with a fixed initial marking, and sometimes also with a fixed final marking that should be reached. This is for example the case in the input for PNREACH.

In the following, when we write that (N, M_0) or (N, M_0, M_f) is a Petri net, we mean that N is a Petri net and M_0, M_f are markings for N, where we consider M_0 as the initial and M_f as the final marking.

2.14 Definition

We say that marking M_f is **reachable** from marking M_0 in the Petri net N if there is a valid firing sequence σ with $M_0 \xrightarrow{\sigma} M_f$. This is the case if and only if (N, M_0, M_f) is a YES-instance of the Petri net reachability problem.

We use $R(N, M_0)$ to denote all markings reachable from M_0,

$$R(N, M_0) = \left\{ M \in \mathbb{N}^P \,\middle|\, \exists \sigma \in T^* \colon M_0 \xrightarrow{\sigma} M \right\}.$$

We furthermore define the **reachability graph** $RG(N, M_0)$, a directed graph whose set of vertices is $R(N, M_0)$ and in which we have an arc $M \to M'$ (for $M, M' \in R(N, M_0)$) if there is a transition t such that $M \xrightarrow{t} M'$.

Obviously, M_f is reachable from M_0 if and only if $M_f \in R(N, M_0)$. Note that $R(N, M_0)$ may be infinite.

In case $R(N, M_0)$ is a finite set, we can explicitly construct $RG(N, M_0)$: We initially add M_0 as a vertex, and then for each vertex M not yet considered do the following: For each transition t, check whether t is enabled in M. If so, compute M' with $M [t\rangle M'$. If M' is not yet a vertex, add it. Draw an arc from M to M'.

If $R(N, M_0)$ is finite, at some point, no new vertices will be added anymore (all transitions are either not enabled or lead to vertices that are already present). If $R(N, M_0)$ is infinite, the algorithm will not terminate.

This makes it interesting for us to consider the **finiteness problem for Petri nets**.

2.15 Definition

Petri net finiteness	
Decide:	Petri net N, initial marking M_0
Decide:	Is $R(N, M_0)$ finite?

The problem is also called the **boundedness problem** due to the following definition and lemma.

2.16 Definition
Let $k \in \mathbb{N}$ be a natural number. A Petri net (N, M_0) is called k-**bounded** or k-**safe** if each component of every marking $M \in R(N, M_0)$ is bounded by k,

$$R(N, M_0) \subseteq \left\{ M \in \mathbb{N}^P \;\middle|\; \forall p \in P\colon M(p) \leqslant k \right\} = \{0, \ldots, k\}^P .$$

2.17 Lemma
Let (N, M_0) be a Petri net. $R(N, M_0)$ is finite if and only if there is a $k \in \mathbb{N}$ such that (N, M_0) is k-bounded.

We will later see an algorithm that decides finiteness.

2.18 Remark
A k-bounded Petri net is actually a finite state system. Still, seeing it as Petri net provides a compact representation. For example, an 1-safe Petri net of polynomial size can represent a finite state system of exponential size.

Even if finiteness is decidable, we are actually very much interested in Petri nets for which $R(N, M_0)$ is infinite. One of the key features of Petri nets is that we can model an unbounded number of threads (as we did in Example 2.3). We will consider Petri net

reachability much later in this lecture. Furthermore, we will consider coverability in the next section, a weaker variant of reachability, for which efficient algorithms are known.

The difficulty of the reachability problem has sparked interest in necessary conditions for reachability that are easy to check. If they are violated, we are sure that the marking under consideration is not reachable. Even if they are hold, it might be non-reachable. There is plethora of research on this, we will just consider one very simple example, the **marking equation**.

Assume that $M_0 \xrightarrow{\sigma} M_f$ for some transition sequence $\sigma \in T^*$. We then need to have

$$M_0 + e(\sigma) = M_f\,.$$

Recall that $e(\sigma) = \sum_{i=0}^{|\sigma|-1} e(\sigma_i) = \sum_{i=0}^{|\sigma|-1} o(\sigma_i) - i(\sigma_i)$. In particular, the order of the transitions in σ does not matter, only the number of their occurrences is important. For each transition t_T, let c_t denote the number of occurrences of t in σ. We then have $e(\sigma) = \sum_{t\in T} c_t \cdot e(t)$.

Consequently, if M_f is reachable from M_0,

- then there is a sequence σ such that $M_0 \xrightarrow{\sigma} M_f$,
- then there is a sequence σ such that $M_0 + e(\sigma) = M_f$,
- then for each $t \in T$, there is a number c_t such that $M_0 + \sum_{t\in T} c_t \cdot e(t) = M_f$.

The last property can be phrased as a problem of linear algebra by introducing some notations. We can use the functions $i, o \in T \to P \to \mathbb{N}$ to defined matrices:

2.19 Definition
The **forward matrix** $\mathbb{F} \in \mathbb{N}^{P\times T}$ is the matrix with $\mathbb{F}_{p,t} = in(t,p)$. The **backward matrix** $\mathbb{B} \in \mathbb{N}^{P\times T}$ is the matrix with $\mathbb{B}_{p,t} = out(t,p)$. The **connectivity matrix** $\mathbb{C} \in \mathbb{Z}^{P\times T}$ is their difference, $\mathbb{C} = \mathbb{B} - \mathbb{F}$.

2.20 Lemma
Let σ be a transition sequence, and let (as above) be c_t be the number of occurrences of transition $t \in T$ in σ. Let us see these numbers as a vector $c \in \mathbb{N}^T$. Then the effect of σ is

$$e(\sigma) = \mathbb{C} \cdot c\,.$$

As a consequence, we can formulate a necessary condition for reachability.

2.21 Lemma
Let (N, M_0, M_f) be a Petri net. Any sequence $\sigma \in T^*$ with $M_0 \xrightarrow{\sigma} M_f$ satisfies the **marking equation**

$$M_0 + \mathbb{C} \times c = M_f$$

where the vector c is defined as above.

The contraposition of this lemma is used to provide a sufficient condition for non-reachability.

2.22 Corollary
Let (N, M_0, M_f) be a Petri net. If the marking equation

$$\mathbb{C} \cdot c = M_f - M_0$$

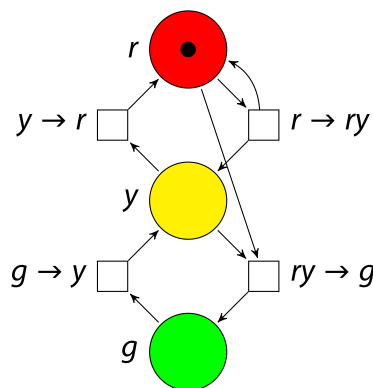
has no solution c, then M_f is not reachable.

Whether the above system of equations has a solution can be easily checked using techniques from linear algebra.

Note that one can quite easily construct examples such that the marking equation has a solution, but M_f is still not reachable.

Exercises

2.23 Exercise: Traffic lights and Petri nets
Consider the Petri net given by the following graphic representation.



a) Write down the net as a tuple $N = (P, T, in, out)$.

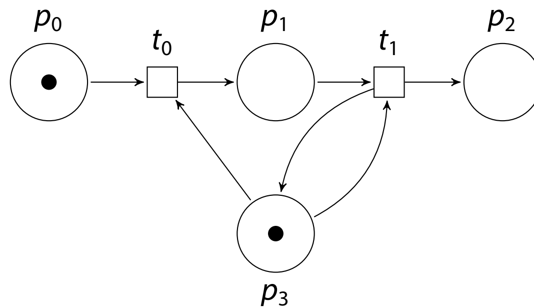
b) The net should model a traffic light, but it contains a bug and exhibits unwanted behavior. Show a valid firing sequence (from the initial marking indicated in the graphic representation) reaching a bad marking.

   Modify the net to fix the problem. The resulting net should be 1-safe.

c) Model two traffic lights handling a road crossing by using two such Petri nets.

2.24 Exercise: The marking equation

Consider the following Petri net.



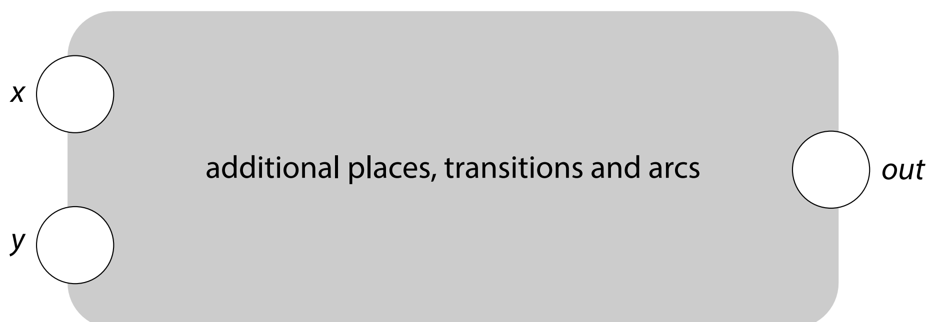
a) Write down the connectivity matrix $\mathbb{C}$ of the Petri net.

b) Argue that the marking $M_f = (0, 0, 1, 0)$ that has one token in p_3 is not reachable from the initial marking $M_0 = (1, 0, 0, 1)$.

c) Prove that the marking equation $M_f - M_0 = \mathbb{C} \cdot c$ has a solution (i.e. there is a vector $c \in \mathbb{N}^T$ satisfying the equation).

2.25 Exercise: Addition and multiplication

Consider the (incomplete) Petri net containing places x, y and *out* depicted below.



a) Add places and transitions to the net such that any computation of the net starting in
   $M_0(x) = m, M_0(y) = n, M_0(out) = 0$ terminates in a marking M_f with $M_f(out) = m + n$.

   (*Terminating* means that no transition is enabled anymore.)

b) Add places and transitions to the net such that any computation of the net starting in $M_0(x) = m, M_0(y) = n, M_0(out) = 0$ terminates in a marking M_f with $M_f(out) \in \{0, \ldots, m \cdot n\}$.

In each part of this exercise, argue briefly that your construction is correct.

2.26 Exercise: VASS

There are other automata models that are equivalent to Petri nets, but they are less useful to model concurrent systems.

A **vector addition system with states (VASS)** of dimension $d \in \mathbb{N}$ is a tuple $A = (Q, \Delta, q_0, v_0)$ where Q is a finite set of control states, $\Delta \subseteq Q \times \mathbb{Z}^d \times Q$ is a set of transitions, $q_0 \in Q$ is the initial state and $v_0 \in \mathbb{N}^d$ is the initial counter assignment. We write transitions $(q, a, q') \in \Delta$ as $q \xrightarrow{a} q'$. A configuration of a VASS is a tuple (q, v) consisting of a control state $q \in Q$ and a counter assignment, a vector $v \in \mathbb{N}^d$. The initial configuration of interest is (q_0, v_0). A transition (q, a, q') is enabled in some configuration (q'', v) if $q'' = q$ and $(v + a) \in \mathbb{N}^d$ (i.e. $(v + a)_i \geqslant 0$ for all $i \in \{1, \ldots, d\}$). In this case, it can be fired, leading to the configuration $(q', v + a)$. Reachability is defined as expected.

a) Let (N, M_0, M_f) be a Petri net. Show how to construct a VASS A and a configuration (q_f, v_f) such that (q_f, v_f) is reachable from (q_0, v_0) in A if and only if M_f is reachable from M_0 in N.

b) Let A be a VASS and (q_f, v_f) a configuration. Show how to construct a Petri net (N, M_0, M_f) such that (q_f, v_f) is reachable from (q_0, v_0) in A if and only if M_f is reachable from M_0 in N.

c) (Bonus exercise, not graded.) A **vector addition system (VAS)** is a VASS with a single state, i.e. $Q = \{q_0\}$. Show that VAS-reachability is interreducible with VASS reachability (or Petri net reachability).

3. Petri net coverability

Instead of considering Petri net reachability, we will study Petri net coverability.

Let us motivate this by an example: A typical application of concurrency theory is the verification of mutual exclusion protocols. For this problem, the goal is to verify that only one thread can access a critical section at a time. If we model this as a Petri net, this means that we have to check that there is no reachable marking in which the amount of tokens in some place *cs* modeling the critical section is 2 or larger. We do not care about the precise amount of tokens in *cs*, and we do not care about the assignment of tokens to other places. This means we are interested in checking whether a marking M with $M(cs) \geq 2$ is reachable. Phrased differently, we are interested in checking whether a marking M that is larger or equal to M_f in every component is reachable, with $M_f(cs) = 2$ and $M_f(p) = 0$ for all $p \neq cs$.

3.1 Definition

Petri net coverability

Decide: Petri net N, initial marking M_0, final marking M_f

Decide: Is there a firing sequence $\sigma \in T^*$ and a marking $M \in \mathbb{N}^P$ such that $M_0 \xrightarrow{\sigma} M$ and $M \geq M_f$?

As usual in this lecture, by $M \geq M_f$ we mean that $M(p) \geq M_f(p)$ for all $p \in P$.

We call a computation $M_0 \xrightarrow{\sigma} M$ with $M \geq M_f$ a **covering computation**.

Another reason for coverability being interesting is the following monotonicity property of Petri nets.

3.2 Lemma

Let N be a Petri net and M_1, M_2 be markings and t a transition. If $M_1 \xrightarrow{t} M_1'$ and $M_2 \geq M_1$, then $M_2 \xrightarrow{t} M_2'$ with $M_2' \geq M_1'$. If we had $M_2 > M_1$ (i.e. in addition to $M_2 \geq M_1$, there is at least one component p with $M_2(p) > M_1(p)$), then also $M_2' > M_1'$.

Proof:

Transition t is enabled in M_2 since $M_2 \geq M_1 \geq in(t)$. Furthermore,

$$M_2' = M_2 + e(t) \geq M_1 + e(t) = M_1' .$$

If $M_2 > M_1$, we have $M_2' = M_2 + e(t) > M_1 + e(t) = M_1'$. □

We also say that larger markings are able to simulate the transition of smaller markings. This fact can be represented by the following diagramm.

$$\begin{array}{ccc} M_2 & \xrightarrow{t} & M_2' \\ \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} \\ M_1 & \xrightarrow{t} & M_1' \end{array}$$

In the following, we want to prove that coverability is an EXPSPACE-complete problem.

- In Section 4, we consider an Algorithm due to Rackoff that solves coverability using exponential space.
- In Section 5, we present Lipton's famous proof for the EXPSPACE-hardness of coverability and reachability.

Exercises

3.3 Exercise: Petri net constructions

a) Let (N, M_0, M_f) be a Petri net. Explain how to construct a Petri net (N', M_0', M_f') with $M_0'(p) = 0$ for all places but a single place p' with $M_0'(p') = 1$ and $M_f'(p) = 0$ for all places such that
$M_f \in R(N, M_0)$ iff $M_f' \in R(N', M_0')$.

b) Let (N, M_0, M_f) be a Petri net. Explain how to construct a Petri net (N', M_0', M_f') such that
M_f is coverable from M_0 in N iff M_f' is reachable from M_0' in N'.

c) Construct a Petri net N with only 3 places, a marking M_0 and markings $M_{c\wedge r}, M_{\neg c\wedge\neg r}$ and $M_{c\wedge\neg r}$ such that

  - $M_{c\wedge r}$ is reachable and coverable from M_0,
  - $M_{\neg c\wedge\neg r}$ is neither reachable nor coverable, and
  - $M_{c\wedge\neg r}$ is coverable, but not reachable.

In each part of this exercise, argue briefly that your construction is correct.

3.4 Exercise: The Ackermann function

a) The three-argument Ackermann function φ is defined recursively as follows.

$$
\begin{aligned}
&\varphi\colon \mathbb{N}^3 \to \mathbb{N} && \\
&\varphi(m,n,0) = m+n && \\
&\varphi(m,0,1) = 0 && \\
&\varphi(m,0,2) = 1 && \\
&\varphi(m,0,x) = m && \text{for } x > 2 \\
&\varphi(m,n,x) = \varphi(m,\varphi(m,n-1,x),x-1) && \text{for } n > 0 \text{ and } x > 0
\end{aligned}
$$

Formally prove the following equalities (e.g. using induction):

$$\varphi(m,n,0) = m+n, \qquad \varphi(m,n,1) = m \cdot n, \qquad \varphi(m,n,2) = m^n.$$

b) Nowadays, one usually considers the following two-parameter variant.

$$
\begin{aligned}
&A\colon \mathbb{N}^2 \to \mathbb{N} && \\
&A(0,n) = n+1 && \\
&A(m,0) = A(m-1,1) && \text{for } m > 0 \\
&A(m,n) = A(m-1,A(m,n-1)) && \text{for } m > 0 \text{ and } n > 0
\end{aligned}
$$

For example, we have

$$A(1,2) = A(0,A(1,1)) = A(0,A(0,A(1,0))) = A(0,A(0,A(0,1))) = A(0,A(0,2)) = A(0,3) = 4\,.$$

Similar to this computation, write down a full evaluation of $A(2,3)$.

3.5 Exercise

3.6 Exercise: Communication-free Petri nets and SAT

A **communication-free Petri net** (or **BPP net**) is a Petri net in which each transition consumes at most one token, i.e. we have $\forall t \in T\colon \sum_{p \in P} in(t,p) \in \{0,1\}$.

Show that the coverability problem for communication-free Petri nets is NP-hard by reducing SAT.

To this end, show how to construct in polynomial time from a given Boolean formula φ in conjunctive normal form communication-free Petri net (N, M_0, M_f) such that M_f is coverable if and only if φ is satisfiable.

Hint: Introduce places for the parts of the formula. A computation of the net should first define a variable assignment, and then evaluate the formula under the assignment.

Remark: In fact, reachability and coverability for communication-free Petri nets are NP-complete.

3.7 Exercise: 1-safe Petri nets and Boolean programs

Recall that a Petri net (N, M_0) is **1-safe** if we have $M \in \{0,1\}^P$ for all $M \in R(N, M_0)$.

Consider **Boolean programs**, sequences of labeled commands over a fixed number of Boolean variables. For simplicity, we restrict ourselves to the following types of commands:

$$z \leftarrow x \wedge y \qquad z \leftarrow x \vee y \qquad z \leftarrow \neg x$$

$$\texttt{if}\ x\ \texttt{then goto}\ \ell_t\ \texttt{else goto}\ \ell_f \qquad \texttt{goto}\ \ell \qquad \texttt{halt}$$

Here, x, y, z are variables and ℓ, ℓ_t, ℓ_f are labeles. The semantics of the commands are expected.

Assume that the initial variable assignment is given by $x = \mathit{false}$ for all variables x.

Assume that a Boolean program is given. Explain how to construct an equivalent 1-safe Petri net. Equivalent means that the unique execution of the Boolean program is halting if and only if a certain marking is coverable.

Remark: This proves that coverability for 1-safe Petri nets is PSPACE-hard. In fact, coverability and reachability for 1-safe Petri nets are PSPACE-complete.

4. Rackoff's algorithm for coverability

We prove the following result.

4.1 Theorem: Rackoff 1978 [Rac78]
The Petri net coverability problem can be solving using exponential space in terms of the input.

Sources
This subsection is based on the original paper [Rac78] and on Roland Meyer's notes on the topic:

tcs.cs.tu-bs.de/documents/ConcurrencyTheory_WS_20162017/rackoff.pdf

The algorithm that we construct to prove Theorem 4.1 is a brute force enumeration of all computations up to a certain length. To be precise, we proceed as follows:

1. We show that if a covering computation exists, then there is one of doubly exponential length.
2. We show that all such computations can be enumerated and tested using exponential space.

Proving the first step is the tricky part. To do so, we relax the enabledness-condition of Petri nets. Instead of considering markings in $\mathbb{N}^P$, we consider **pseudo markings** in $\mathbb{Z}^P$ in which only the first i components need to stay non-negative, where $0 \leqslant i \leqslant |P|$. We then prove a variant of the theorem for each i by induction, i.e. we iteratively increase the number of components that are treated properly. The case in which i is the number of places yields the desired result.

Throughout this section, (M, M_0, M_f) is the fixed Petri net of interest. We will assume that the places are ordered, i.e. $P = \{1, \dots, \ell\}$ for some number $\ell \in \mathbb{N}$. This can be enforced by an appropriate renaming. We furthermore us $n = |N| + |M_f| + |M_0|$ to denote the size of the encoding of the input net.

4.2 Definition
Let $i \in \{0, \dots, \ell\}$ be a number.

A **pseudo marking** is a vector $M \in \mathbb{Z}^P$. It is called ***i*-non-negative** if we have $M(p) \geqslant 0$ for all $p \in \{1, \dots, i\}$. It is called ***i*-covering** if we have $M(p) \geqslant M_f(p)$ for all $p \in \{1, \dots, i\}$.

In an i-non-negative marking M, a transition t is i-**enabled** if we have $M(p) \geq in(p)$ for all $p \in \{1, \ldots, i\}$. In this case, we can **fire** it, yielding the new marking $M' = M + e(t)$ as usual. We also write $M \xrightarrow{t} M'$ as it will be clear from the context which i we are considering.

An i**-non-negative,** i**-covering computation** is a sequence of markings and transitions

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \ldots \xrightarrow{t_m} M_m$$

such that each marking M_i is i-non-negative and M_m is i-covering.

4.3 Remark

Note that every pseudo marking is 0-non-negative and 0-covering. Every sequence of markings and transitions as above is a 0-non-negative, 0-covering computation.

An ℓ-non-negative pseudo marking is a normal marking, an ℓ-covering marking is covering, and an ℓ-non-negative, ℓ-covering computation is a covering computation.

4.4 Definition

For some marking M and $i \in \{0, \ldots, \ell\}$, we define

$$m(i, M) = \min\left\{|\sigma| + 1 \;\middle|\; M \xrightarrow{\sigma} M' \text{ is a } i\text{-non-negative, } i\text{-covering computation}\right\}.$$

We define $m(i, M) = 0$ if no such computation exists.

Our goal is to obtain a bound for $m(\ell, M_0)$, i.e. the case where we treat all places properly and consider the initial marking of interest. In the proof of the bound, we will need to consider a different marking as initial. Therefore, we quantify over all initial markings.

4.5 Definition

For $i \in \{0, \ldots, \ell\}$, we define

$$f(i) = \max\left\{m(i, M) \;\middle|\; M \in \mathbb{Z}^P\right\}.$$

First note that it is not at all clear that $f(i)$ is a well-defined natural number, as $m(i, M)$ could grow unboundedly for different values of M. If we prove a bound on $f(i)$, we will not only show that it is well-defined, but we will also obtain a bound for $m(i, M_0) \leq f(i)$. This is provided by the following technical lemma and proposition.

4.6 Lemma

$f(0) = 1$.

Proof:
For each $M \in \mathbb{Z}^P$, the empty computation $M \xrightarrow{\varepsilon} M$ is 0-non-negative and 0-covering. We have $m(0, M) = |\varepsilon| + 1 = 1$ and thus $f(0) = 1$. □

4.7 Proposition
For all $i \in \{0, \ldots, \ell - 1\}$, we have

$$f(i+1) \leqslant \left(2^n \cdot f(i)\right)^{i+1} + f(i) .$$

Proof:
Recall that $f(i + 1) = \max_{M \in \mathbb{Z}^P} m(i + 1, M)$. Hence, if we prove that $m(i + 1, M) \leqslant \left(2^n \cdot f(i)\right)^{i+1} + f(i)$ for all pseudo markings $M \in \mathbb{Z}^P$, we are done. Let $M \in \mathbb{Z}^P$ be an arbitrary pseudo marking.

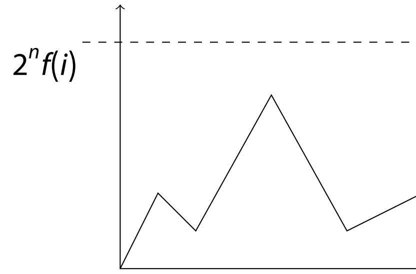
If there is no $(i + 1)$-non-negative and $(i + 1)$-covering computation, then we have $m(i + 1, M) = 0$, which obviously satisfies the desired bound. Let us therefore assume that such a computation exists, i.e. we have

$$M = M^{(0)} \xrightarrow{t_1} M^{(1)} \xrightarrow{t_2} M^{(2)} \xrightarrow{t_3} \ldots \xrightarrow{t_m} M^{(m)}$$

such that all transitions are $(i+1)$-enabled when they are fired, all markings M_i are $(i+1)$-non-negative and $M^{(m)}$ covers M_f in the first $i + 1$ components.

Our goal is to transform this computation such that it remains $(i + 1)$-non-negative and $(i + 1)$-covering, but satisfies the bound on the length. We distinguish two cases.

Case 1: In all occurring markings, the number of tokens in the first $(i + 1)$ places is bounded by $2^n \cdot f(i) - 1$.



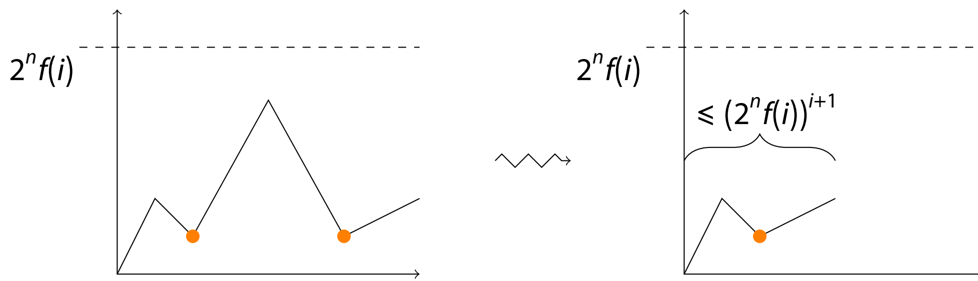
This means $\forall j \in \{0, \ldots, m\}, \forall p \in \{1, \ldots, i + 1\}, M^{(j)}(p) < 2^n \cdot f(i)$. Furthermore, we have $M^{(j)}(p) \geqslant 0$ as the computation was $(i + 1)$-non-negative.

Assume that the computation above contains markings $M^{(j)}$ and $M^{(j')}$ for $j < j'$ that coincide on the first $i+1$ components. Then, we can delete the transitions $t_{j+1}, \ldots, t_{j'}$, obtaining the new computation

$$M = M^{(0)} \xrightarrow{t_1} \ldots \xrightarrow{t_j} M^{(j)} \xrightarrow{t_{j'+1}} \hat{M}^{(j'+1)} \xrightarrow{t_{j'+2}} \ldots \hat{M}^{(m)} .$$

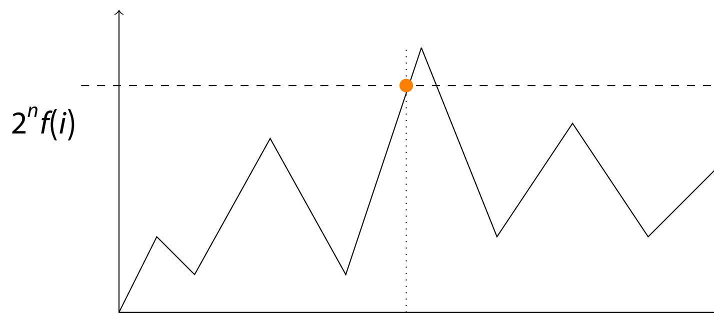
Note that since $M^{(j)}$ and $M^{(j')}$ coincide in the first $(i+1)$ components, transition $t_{j'+1}$ is $(i+1)$-enabled in $M^{(j)}$. Furthermore, for each k, $M^{(j'+k)}$ and $\hat{M}^{(j'+k)}$ coincide on the first $(i+1)$ components.

Consequently, the newly constructed computation is $(i+1)$-non-negative and $(i+1)$-covering. It still satisfies in all markings that the number of tokens on the first $(i+1)$ places is bounded by $2^n \cdot f(i) - 1$.



We may iteratively apply the step above to shorten the initial computation until we arrive at a computation in which there are no two markings that coincide on the first $(i+1)$ places. We have $\left|\{0, \ldots, 2^n \cdot f(i) - 1\}^{i+1}\right| = (2^n \cdot f(i))^{i+1}$, i.e. there are only $(2^n \cdot f(i))^{i+1}$ many possibilities for the first $(i+1)$ components. Consequently, a repetition-free computation consists of at most $(2^n \cdot f(i))^{i+1}$ many markings. Its number of transition is thus $(2^n \cdot f(i))^{i+1} - 1$, which is smaller than the bound for $f(i+1)$ that we wanted to show.

Case 2: There is a marking in which some of the first $(i+1)$ places exceeds the bound of $2^n \cdot f(i) - 1$ many tokens.



Consider the first marking $M^{(j)}$ (i.e. with j minimal) in which some place p contains at least $2^n \cdot f(i)$ many tokens. By reordering the places appropriately, we may assume without loss of generality that this happens for the place $i+1$, i.e. we have

$$M^{(j)}(i+1) \geq 2^n \cdot f(i)\,.$$

(There might be other places that also exceed the bound in $M^{(j)}$, which will not influence the correctness of our proof.) Let $\sigma = t_1 \ldots t_m$ be the sequence of the transitions used in the computation. We split it into $\sigma = \sigma_1.t_j.\sigma_2$ such that $\sigma_1 = t_1 \ldots t_{j-1}$ and $\sigma_2 = t_{j+1} \ldots t_m$. We can write our original computation is

$$M \xrightarrow{\sigma_1} M^{(j-1)} \xrightarrow{t_j} M^{(j)} \xrightarrow{\sigma_2} M^{(m)}\,.$$

Note that in the computation $M \xrightarrow{\sigma_1} M^{(j-1)}$, all markings admit the bound. Therefore, we may treat it as in Case 1 and can assume that it has length at most $\left(2^n \cdot f(i)\right)^{i+1} - 1$.

Now consider $M^{(j)}$. We know that there is a i-non-negative, i-covering computation from $M^{(j)}$ on, namely $M^{(j)} \xrightarrow{\sigma_2} M^{(m)}$. We thus have $m(i, M^{(j)}) \neq 0$. We furthermore have $m(i, M^{(j)}) \leq f(i)$. By the definition of $f(i)$, there is an i-non-negative, i-covering computation

$$M^{(j)} \xrightarrow{\sigma_2'} \hat{M}$$

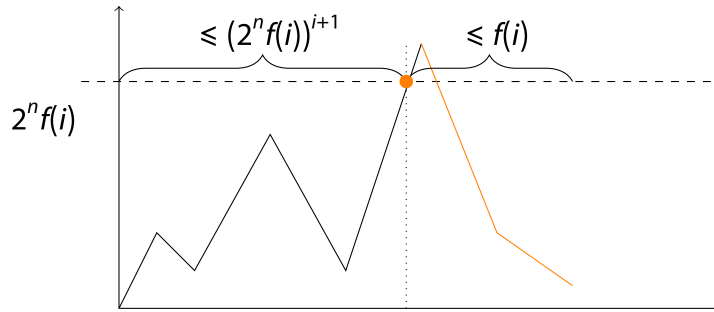
with $|\sigma_2'| \leq f(i) - 1$.

We claim that

$$M \xrightarrow{\sigma_1} M^{(j-1)} \xrightarrow{t_j} M^{(j)} \xrightarrow{\sigma_2'} \hat{M}\,.$$

is the desired $(i+1)$-non-negative, $(i+1)$-covering computation with

$$\left|\sigma_1.t_j.\sigma_2'\right| \leq |\sigma_1| + 1 + |\sigma_2'| \leq \left(\left(2^n \cdot f(i)\right)^{i+1} - 1\right) + 1 + (f(i) - 1) = \left(2^n \cdot f(i)\right)^{i+1} + f(i) - 1\,.$$



That the bound on the length holds is clear from the inequalities above. It remains to argue that all transitions in σ_2' are $(i+1)$-enabled when they are fired and that the computation is indeed $(i+1)$-non-negative and $(i+1)$-covering.

Since $M^{(j)} \xrightarrow{\sigma_2'} \hat{M}$ was an i-non-negative and i-covering computation, we do not have to care about the first i places; their value in each occurring marking coincides with the value in the corresponding marking of $M^{(j)} \xrightarrow{\sigma_2'} \hat{M}$. It remains to consider to consider place $i + 1$.

Recall that $n = |N| + |M_f| + |M_0|$. By the definition of the size of the encoding of a marking resp. Petri net, the maximal multiplicity of any arc in the net is 2^n. This means in the worst case, each transition contained in σ_2' consumes 2^n tokens on place $i + 1$. Note that σ_2' has length at most $f(i) - 1$.

We obtain that firing σ_2' consumes at most $2^n \cdot (f(i) - 1)$ tokens from $(i + 1)$. Recall that we assumed that $M^{(j)}(i + 1)$ is at least $2^n \cdot f(i)$. Therefore, we have that the number of tokens on place $i + 1$ on $M^{(j)}, \hat{M}$ and any marking occurring in between is at least

$$\left(2^n \cdot f(i)\right) - \left(2^n \cdot (f(i) - 1)\right) \geqslant 2^n .$$

We conclude that all transitions are $(i + 1)$-enabled whenever they are fired and the computation is indeed $(i + 1)$-covering. We have that $\hat{M}$ is greater or equal to M_f in the first i components. Since $n = |M_f| + |N| + |M_0|$, we have $M_f(i + 1) \leqslant 2^n \leqslant \hat{M}(i + 1)$, so the computation is also $i + 1$-covering. This finishes the proof. □

The proposition gives us a recursively defined bound for $f(i)$. We will combine it with Lemma 4.7 to obtain a non-recursive bound. We start by giving a simpler recursive bound.

4.8 Lemma
Define $g(0) = 2^{3n}$ and $g(i + 1) = (g(i))^{3n}$. We have $f(i) \leqslant g(i)$ for all i.

Proof:
Before we can prove the main statement of the lemma, we have to show that $2^{n \cdot (i+1)} \leqslant g(i)$ for all i. We proceed by induction on i.

Base case, $i = 0$.

We have $2^{n \cdot 1} = 2^n \leqslant g(0) = 2^{3n}$.

Inductive step, $i \to i + 1$.

We have

$$\begin{aligned}2^{n\cdot((i+1)+1)} &= 2^{n\cdot(i+1)} \cdot 2^n \leqslant g(i) \cdot 2^n \\ &\leqslant g(i) \cdot g(0) \leqslant g(i) \cdot g(i) \\ &= (g(i))^2 \leqslant (g(i))^{3n} \\ &= g(i+1)\,.\end{aligned}$$

We can now prove the statement of the lemma by induction.

Base case, $i = 0$**.** We have $f(i) = 1 < 8 \leqslant 2^{3n} = g(0)$ using Lemma 4.6.

Inductive step, $i \to i+1$**.**

We have

$$f(i+1) \leqslant \left(2^n \cdot f(i)\right)^{i+1} + f(i)$$

by Proposition 4.7. Note that this expression is monotonous in $f(i)$, so we may use the induction hypothesis to obtain

$$\begin{aligned}f(i+1) &\leqslant \left(2^n \cdot g(i)\right)^{i+1} + g(i) \\ &= \left(2^n\right)^{i+1} \cdot g(i)^{i+1} + g(i) \\ &= \left(2^{n\cdot(i+1)}\right) \cdot g(i)^{i+1} + g(i)\,.\end{aligned}$$

Using the statement that we have proven above, we finally obtain

$$\begin{aligned}f(i+1) &\leqslant \left(2^{n\cdot(i+1)}\right) \cdot g(i)^{i+1} + g(i) \\ &\leqslant g(i) \cdot g(i)^{i+1} + g(i) \\ &\leqslant 2 \cdot g(i) \cdot g(i)^{i+1} \\ &\leqslant 2 \cdot g(i) \cdot g(i)^n = 2g(i)^{n+1} \\ &\leqslant g(i)^{n+2} \leqslant g(i)^{3n} \\ &= g(i+1)\,.\end{aligned}$$

Here, we have used that $i+1$ is at most the number of places ℓ, and $n \geqslant |N| \geqslant \ell$. □

We can now use this lemma to obtain a non-recursive bound.

4.9 Lemma
We have

a) $g(\ell) \leqslant 2^{(3n)^n}$

b) $(3n)^n \leqslant 2^{c \cdot n \log n}$, where c is a constant independent of n.

Proof:

a) By definition, we have
$$g(\ell) = \left(\dots\left(2^{3n}\right)^{3n}\dots\right)^{3n},$$
where we have $\ell + 1$ powers of $3n$. We iteratively us the power law $(a^b)^c = a^{bc}$ to obtain
$$g(\ell) = 2^{(3n)^{(\ell+1)}} \leqslant 2^{(3n)^n}.$$

b) We have
$$\begin{aligned}(3n)^n &= \left(3 \cdot 2^{\log n}\right)^n \leqslant \left(2^2 \cdot 2^{\log n}\right)^n \\ &= \left(2^{\log n + 2}\right)^n \leqslant \left(2^{4 \log n}\right)^n \\ &= 2^{4n \log n}.\end{aligned}$$

□

We combine all results to obtain the following proposition.

4.10 Proposition

If M_f is coverable from M_0, then there is a covering computation of length at most $2^{2^{c \cdot n \log n}}$, where c is a constant not dependent on the size of the input.

Proof:

The definition of $m(\ell, M_0)$, $m(\ell, M_0) \leqslant f(\ell)$, Lemma 4.8 and Lemma 4.9. □

We have finally obtained that if there is a covering computation $M \xrightarrow{\sigma} M$ (with $M \geqslant M_f$), then there is one with $|\sigma| \leqslant 2^{c \cdot n \log n}$. We have to construct an algorithm that uses this fact to decide coverability.

Proof of Rackoff's theorem, Theorem 4.1:

We have to prove that there is a deterministic algorithm using exponential space checking whether a covering computation exists.

We first construct a non-deterministic algorithm that does this. The algorithm keeps track of a marking M and a counter c

1: $M \leftarrow M_0$
2: $c \leftarrow 0$

3: **while** $c \leqslant 2^{2^{c \cdot n \log n}}$ **do**
4: $c \leftarrow c + 1$
5: Guess a transition t.
6: Verify that it is enabled in M
7: Compute the marking M' with $M \xrightarrow{t} M'$
8: $M \leftarrow M'$
9: **if** $M \geqslant M_f$ **then**
10: **return true**
11: **end if**
12: **end while**
13: **return false**

If the verification that t is enabled fails or the algorithm reaches $c > 2^{2^{c \cdot n \log n}}$, it returns **false**.

Using Proposition 4.10, it is clear that the algorithm has a computation returning **true** if and only if M_f is coverable from M_0.

We still need to argue that the algorithm can be implemented using exponential space. The algorithm needs to store c, which can be done via a binary encoding using $\log 2^{2^{c \cdot n \log n}} = 2^{c \cdot n \log n}$ many bits.

We furthermore need to store the marking M. Note that any marking M that occurs assigns to each place at most

$$\begin{aligned} M_0 + 2^{2^{c \cdot n \log n}} \cdot 2^n &\leqslant 2^{2^{c \cdot n \log n}} \cdot 2^{n+1} \\ &\leqslant 2^{2^{c \cdot n \log n} + n + 1} \end{aligned}$$

many tokens. (Here, we have used $|M_0| \leqslant n$.) This number can be represented in binary using at most $2^{c \cdot n \log n} + n + 1$ many bits.

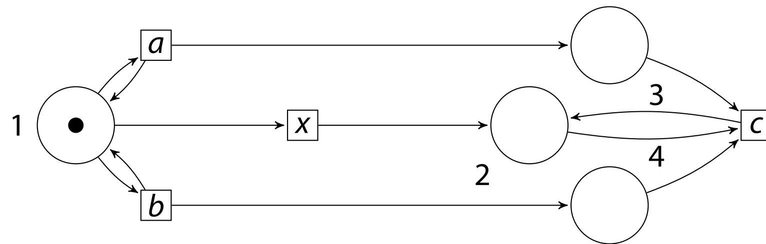
To finish the proof, we need to convert the non-deterministic algorithm to a deterministic one. Savitch's theorem proves that NEXPSPACE = EXPSPACE, yielding a deterministic algorithm for coverability using only exponential space. □

Exercises

4.11 Exercise: Rackoff's bound

Consider the Petri net $N = (\{1, 2, 3, 4\}, \{a, b, c, x\}, in, out)$ with multiplicities as depicted

below. The initial marking of interest is $M_0 = (1, 0, 0, 0)^T$ and the final marking is $M_f = (1, 0, 10, 100)^T$.



Compute the values $m(3, M_0)$ and $f(3)$ and argue why they are correct.

5. Lipton's hardness result

We prove that all interesting properties of general Petri nets are EXPSPACE-hard. More precisely, they require at least $2^{\mathcal{O}(\sqrt{n})}$ space, where n is the size of the encoding of the input. Interesting properties include reachability and coverability. By this, we obtain that Rackoff's algorithm for coverability is in the optimal complexity class.

Sources
The result was proven by Lipton in 1976 [Lip76]. Our presentation is based on Roland Meyer's handwritten notes on the topic:

tcs.cs.tu-bs.de/documents/ConcurrencyTheory_SS_2015/lipton_part_1_week_3.pdf

tcs.cs.tu-bs.de/documents/ConcurrencyTheory_SS_2015/lipton_part_2_week_3.pdf

These notes are based on a survey paper by Javier Esparza [Esp98].

The result follows from the following theorem.

5.1 Theorem: Lipton 1976 [Lip76]
A deterministic Turing machine of size n with exponential space consumption (in n) can be simulated by a Petri net of size $\mathcal{O}(n^2)$. This Petri net can be constructed in polynomial time.

Here, we assume that that the Turing machine is running on the empty input. This means we reduce the following problem that is known to be EXPSPACE-hard.

Turing machine acceptance on empty input with exponential space bound	
Decide:	Turing machine $\mathcal{M}$ of size n with space consumption bounded by 2^n
Decide:	Does $\mathcal{M}$ accept the empty word?

Unfortunately, it is technically challenging to encode a Turing machine into a Petri net. The Petri net is essentially a type of (concurrent) counter machine, while the Turing machine uses an ordered tape as storage. It is not clear how to represent a tape cell (that has one of finitely many symbols as content) by a place (that carries an unbounded number of tokens). To overcome this, we will need some intermediary steps.

Our approach:

$$\text{Turing machine} \longrightarrow \text{counter program} \longrightarrow \text{PN program} \longrightarrow \text{Petri net}$$

From Turing machines to counter programs:

A **counter program** is a goto-program that manipulates a fixed number of non-negative counters, variables that store a natural number. (A more formal definition will be given later.) In a **bounded** counter program, the variables are not incremented beyond some fixed bound.

5.2 Theorem
A deterministic Turing machine $\mathcal{M}$ of size n can be simulated by a counter program $c_{\mathcal{M}}$ consisting of $\mathcal{O}(n)$ commands such that $\mathcal{M}$ halts on the empty tape if and only if $c_{\mathcal{M}}$ halts.

If $\mathcal{M}$ uses at most 2^n cells, then the counters in $c_{\mathcal{M}}$ are bounded by 2^{2^n}.

The program $c_{\mathcal{M}}$ can be constructed in polynomial time.

Proof sketch:
We assume without loss of generality that the Turing-Machine uses $\{0, 1\}$ as tape alphabet.

We represent the tape content of the Turing machine by two stacks. Assume the Turing machine is in configuration $w\, q\, v$, i.e. $w \in \{0, 1\}^*$ is the tape content to the left of the head, $v \in \{0, 1\}^*$ is the rest of the stack, and the first letter of v is the content of the cell to which the head is pointing. Then the first stack contains w, where the first symbol is stored at the bottom and the last symbol is stored at the top. The second stack contains v where the first symbol is stored at the top and the last symbol is stored at the bottom. Moving the head can now be realized by popping from one stack and pushing onto the other.

A stack over $\{0, 1\}$ can be simulated by two counters. One counter holds the natural number represented by the stack content, where the least significant bit represents the topmost entry of the stack. Operations on the stack can be simulated my manipulating this number. For example, pushing 1 onto the stack representing value $v \in \mathbb{N}$ is implemented by setting the stack value to $2v + 1$. The second counter is needed as auxiliary storage to be able to implement the operations.

Combining these two insights, we obtain that the tape of a Turing machine can be simulated by four counters.

If the tape size is at most 2^n, then so is the size of each of the stacks. The natural numbers that occur as counter values are obtained by seeing the stacks as binary numbers. Consequently, the numbers may be exponential in the size of the stack. We obtain that if the tape has at most size 2^n, then the counters are bounded by 2^{2^n}. $\square$

5.3 Remark

- In fact, one could further reduce the numbers of counters needed to simulate a tape from 4 to 2 by using an encoding in terms of prime numbers [Min67]. Using this technique, one would obtain that if the tape contains at most 2^n cells, the counter values are bounded by $2^{2^{2^n}}$, which is triply exponential.
- The first part of the theorem holds independently of the second one: A counter program can simulate a Turing machine even if its space consumption is not bounded. In fact, counter programs over two counters are Turing complete and all interesting properties (like halting) are undecidable.

By the above theorem, it is clear that the following problem is EXPSPACE-hard, so it is sufficient to reduce it to Petri net coverability.

Halting problem for counter programs with doubly-exponentially bounded counters	
Decide:	A counter program c with counter values bounded by 2^{2^n}
Decide:	Does c halt?

We proceed to give a formal definition of counter programs.

5.4 Definition: Counter program
A **counter program** over a set of counter variables $x_0, \ldots, x_m$ consists of a sequence of labeled commands

$$\begin{aligned} \ell_0 &: \ \mathtt{cmd}_0; \\ \ell_1 &: \ \mathtt{cmd}_1; \\ &\ \vdots \\ \ell_n &: \ \mathtt{cmd}_n; \end{aligned}$$

Each $\mathtt{cmd}_i$ is of one of the following types:

- **Increment:**

$$\mathtt{cmd}_i = x_j{+}{+}$$

  where x_j is a counter.

- **Decrement:**

$$\mathtt{cmd}_i = x_j{-}{-}$$

  where x_j is a counter.

- **Unconditional jump:**

$$\texttt{cmd}_i = \texttt{goto}\ \ell_k$$

where ℓ_k is a label.

- **Conditional jump / Zero test:**

$$\texttt{cmd}_i = \texttt{if}\ x_j = 0\ \texttt{then goto}\ \ell_z\ \texttt{else goto}\ \ell_{nz}$$

where x_j is a counter and ℓ_z, ℓ_{nz} are labels.

- **Halt:**

$$\texttt{cmd}_i = \texttt{halt}\ .$$

We require that each command has a distinct label. When writing down programs, we sometimes omit the labels of commands that do not occur as the target location of a jump. We will later assume without loss of generality that the last labeled command is ℓ_n: $\texttt{halt}$;

The semantics is as expected:

- A configuration of the program is a label ℓ together with an assignment $M \in \{0, \ldots, b\}^k$ of the variables.
- If the command for the current label ℓ_k is x_j++, then the value of x_j is incremented and the program goes to the next label ℓ_{k+1}.
- If the command for the current label ℓ_k is x_j−−, then the value of x_j is decremented and the program goes to ℓ_{k+1}.

  This can only happen if the current value of x_j is non-zero. If it is zero, the execution gets stuck.
- If the command for the current label is $\texttt{goto}\ \ell_k$, then the program goes to ℓ_k without changing the variable assignment.
- If the command for the current label is $\texttt{if}\ x_j = 0\ \texttt{then goto}\ \ell_z\ \texttt{else goto}\ \ell_{nz}$, then the program goes to ℓ_z or ℓ_{nz}, depending on whether the value of x_j in the current variable assignment is 0.
- If the command for the current label is $\texttt{halt}$, the execution halts.

In the initial configuration, the program is at ℓ_0 and all variables have value 0.

Note that counter programs are deterministic: There is a unique computation of a program from the initial configuration. This computation might be infinite, get stuck because of a blocked decrement, or it might reach a `halt` command.

The **halting problem for counter programs** is, given a counter program, checking whether the unique execution of the program reaches a `halt`-command.

Simulating zero tests:

In the following, we will only consider counter programs in which the counter values never go above 2^{2^n}, where n is the number of commands. Our goal is to translate such a counter program into an equivalent Petri net.

When translating a counter program into a Petri net, increments, decrements, halting and unconditional jumps can be easily modeled. The problem are the zero tests: A transition of a Petri net can only check that a marking has at least a certain number of tokens in a place, but it cannot check that there is no token in a place.

The absence of zero tests is the crucial difference between Petri nets and counter programs. Any extension of Petri nets that allows for zero tests makes them Turing complete and thus the reachability problem becomes undecidable. (But there are "mild" extensions of Petri nets that do still have a decidable reachability problem.)

To be able to model zero tests, we can use that the counters are bounded. We represent the counter variable x_j by two places: The number of tokens on place x_j is the value of x_j, the number of tokens on place $\overline{x_j}$ is the bound minus this value. We will maintain the invariant

$$x_j + \overline{x_j} = 2^{2^n} .$$

Incrementing and decrementing x_j can now be done by moving tokens from $\overline{x_j}$ to x_j respectively the other way around. Checking that x_j is non-zero can be done by decrementing it and incrementing it again: If it was zero, the decrement blocks the execution. Instead of testing that x_j is zero, we can test that $M(\overline{x_j}) = 2^{2^n}$.

The problem is the initialization of the places: Since we assume that the initial values of the counters is 0, we need to have $\overline{x_j} = 2^{2^n}$ in the initial marking., We cannot define our initial marking like this, since $\log 2^{2^n} = 2^n$ is not polynomial, but exponential in n.

Lipton's famous trick is a procedure that allows a polynomially-sized Petri net to create exactly 2^{2^n} tokens on a place. Understanding this trick is the fundamental part of the proof of the following theorem.

5.5 Theorem: Lipton 1976 [Lip76]
A counter program with n commands and counters bounded by 2^{2^n} can be simulated by a Petri net of size $\mathcal{O}(n^2)$. This Petri net can be constructed in polynomial time.

PN programs and Petri nets:

Towards a proof of the theorem, we would need to construct for each command of the counter program an equivalent part of the Petri net. Doing this directly is possible, but messy. We instead opt for introducing PN programs and using them as another intermediary step.

5.6 Definition: PN program
A **PN program** over a set of counter variables $x_0, \ldots, x_m$ is a sequence of labeled commands, just as a counter program.

The halt command, and increment, decrement and unconditional jump are valid commands, just as for counter programs. We furthermore have the following types of commands

- **Nondeterministic branching:**

  $$\texttt{cmd}_\texttt{i} = \texttt{goto}\ \ell_e\ \texttt{or}\ \texttt{goto}\ \ell_o$$

  where ℓ_e, ℓ_o are labels.

  When executing this command, the execution will nondeterministically either continue at label ℓ_e or at label ℓ_o, without changing the variable assignment.

- **Subroutine call & return:**

  $$\texttt{cmd}_\texttt{i} = \texttt{call}\ \ell_{sr}$$

  $$\texttt{cmd}_{\texttt{i}'} = \texttt{return}$$

  When executing `call` ℓ_{sr}, the execution will continue at label ℓ_{sr}, but it will store the label ℓ_i from which the call was made.

  When `return` is executed inside this subroutine, then the execution will continue at label ℓ_{i+1}, i.e. the location at which the routine was called.

5.7 Remark
It might seem like the semantics of PN programs requires us to keep track of an unbounded call stack (to be able to return to the correct location). We will actually only consider **well-structured programs**, in which

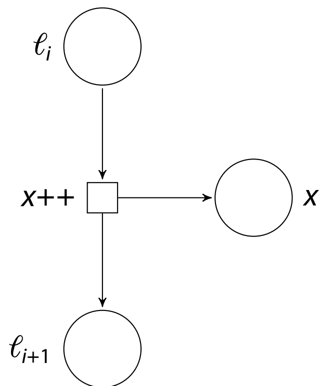
- unconditional jumps will only jump inside the current subroutine,
- there is an order on the routines, i.e. the program consists of a main routine, 1st level subroutines, 2nd level subroutines and so on. We guarantee that a level k subroutine only calls subroutines of level $k + 1$ and higher.

By those two conditions, the height of the call stack is bounded by some number that can be extracted from the syntax of the program.

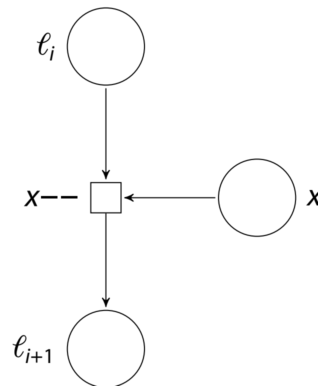
5.8 Remark
In contrast to counter programs, PN programs are non-deterministic. There is not a unique execution anymore. The halting problem is now checking whether an execution exists that reaches `halt`.

To a PN program, we can assign an equivalent Petri net. In the following, we show how to do this for all commands but subroutine calls and returns.

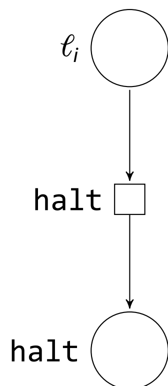
5.9 Definition: Petri net semantics of PN programs, Part 1
To a PN program, we can associate a Petri net with one place for each counter variable x one place for each label ℓ_i, and a special place for `halt`. The transitions are as follows.



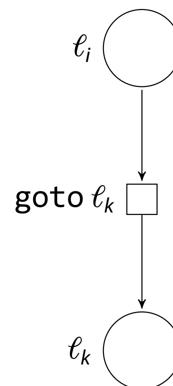
Encoding of ℓ_i: x++;



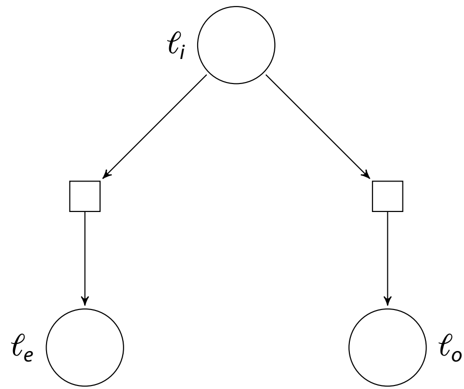
Encoding of ℓ_i: x−−;



Encoding of ℓ_i: `halt`;



Encoding of ℓ_i: `goto` ℓ_k;



Encoding of ℓ_i: goto ℓ_e or goto ℓ_o;

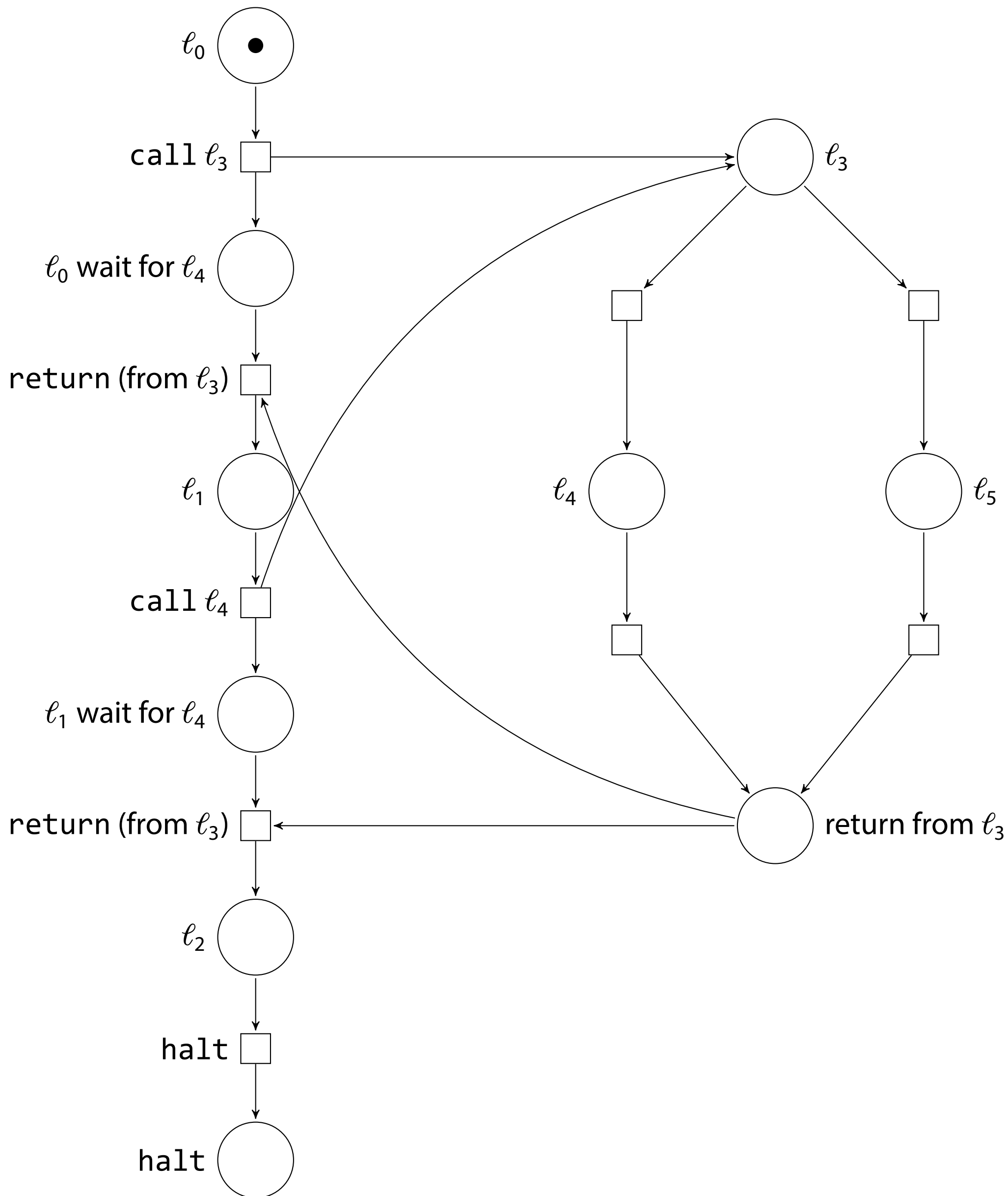
It remains to see how subroutine calls and returns are handled. Before defining this formally, we consider an example.

5.10 Example
Consider the following example program.

ℓ_0: call ℓ_3;
ℓ_1: call ℓ_3;
ℓ_2: halt
ℓ_3: goto ℓ_4 or goto ℓ_5;
ℓ_4: return;
ℓ_5: return;

The lines 0, 1, 2 form the main routine, the lines 3, 4, 5 a 1st level subroutine.

The following Petri net is the net associated to this program.



We can now define the construction in general. Note that we chose not to make concepts like *subroutines* formal.

5.11 Definition: Petri net semantics of PN programs, Part 2
The Petri net associated to a PN program has a return place for each subroutine. Any `return` inside the routine will be modeled as a transition moving the token to the return place.

A routine call is modeled using two transitions: When the procedure is called, two tokens are produced, one on the entry location of the routine, and one on a special *wait*

place. The return transition consumes a token from the return location of the procedure and one from the wait place.

Overall, we obtain that a PN program with at most n commands (and therefore also at most n counters) can be modeled as a Petri net of at most $\mathcal{O}(n)$ transitions and $\mathcal{O}(n)$ places. The size of this net is consequently in $\mathcal{O}\left(n^2\right)$.

Note that this representation is compact: Since the programs we consider are well-structured, we could unfold them to get rid of subroutines. In the example, we could create two copies of the subroutine $4, 5, 6$, one copy for the first call and one for the second.

5.12 Proposition
A PN program has a halting execution if and only if in the associated Petri net, the marking that requires one token on the `halt`-place is coverable.

Consequently, it is sufficient to show that we simulate a counter program by a PN program.

From counter programs to PN programs:

We will now prove the following: Given a counter program with n commands and counters bounded by 2^{2^n}, we can construct a PN program with $\mathcal{O}(n)$ commands in polynomial time such that the unique execution of the counter program halts if and only if the PN program has a halting execution.

Recall that for each variable x of the counter program, the PN program will have variables $x, \overline{x}$ and we will have the invariant

$$\overline{x} = 2^{2^n} - x\,.$$

Initially, all variables have value 0. The PN program np that we construct is of the shape

$$\mathrm{np} = \mathrm{np}_{init}; \mathrm{np}_{sim}\,.$$

In np_{init}, the variables $\overline{x}$ are set to 2^{2^n}. After executing it, the invariant will hold. The second part np_{sim} will simulate the given counter program, maintaining the invariant.

The construction of the simulation, $\mathbf{np}_{sim}$:

We will first discuss how to construct np_{sim}, assuming that the variables have been initialized correctly.

- Each command x++ in the counter program is replaced by x++; $\bar{x}$−− in the PN program.
- Each command x−− in the counter program is replaced by x−−; $\bar{x}$++ in the PN program.
- `halt` and `goto` ℓ commands remain unchanged.

We still have to show how a zero test

$$\ell\colon \texttt{if } x = 0 \texttt{ then goto } \ell_z \texttt{ else goto } \ell_{nz}$$

can be modeled. Do this end, we design a macro

$$\texttt{Test}_n(x, \ell_z, \ell_{nz})$$

that replaces each zero test. Its behavior is specified as follows.

5.13 Definition: Specification of the macro $\texttt{Test}_n(x, \ell_z, \ell_{nz})$

- If $\texttt{Test}_n(x, \ell_z, \ell_{nz})$ is executed starting from a variable assignment in which $x = 0$ holds, then *some* execution of the macro leads to ℓ_z and *no* execution leads to ℓ_{nz}.
- If $\texttt{Test}_n(x, \ell_z, \ell_{nz})$ is executed starting from a variable assignment in which $x > 0$ holds, then *some* execution of the macro leads to ℓ_{nz} and *no* execution leads to ℓ_z.
- There might be executions that do get stuck.
- The macro $\texttt{Test}_n$ has no side effects: In any execution reaching ℓ_z or ℓ_{nz}, the variable assignment will be unchanged.

To define $\texttt{Test}_n(x, \ell_z, \ell_{nz})$, we introduce another macro $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$. It is s easier to design, but it has a side effect: After an execution leading to ℓ_z, the values of x and $\bar{x}$ are swapped. Other than that, its specification coincides with the one of $\texttt{Test}_n$.

To cancel the swapping out, we swap twice.

5.14 Definition: Macro $\texttt{Test}_n(x, \ell_z, \ell_{nz})$

$$\begin{aligned} \ell\colon &\quad \texttt{Test}'_n(x, \ell_{cont}, \ell_{nz}); && \text{// Swaps } x \text{ and } \bar{x} \\ \ell_{cont}\colon &\quad \texttt{Test}'_n(\bar{x}, \ell_z, \ell_{nz}); && \text{// Undoes the swap} \end{aligned}$$

Note that if x is zero, after executing $\texttt{Test}'_n(x, \ell_{cont}, \ell_{nz})$, we will have $x = 2^{2^n}$ and $\bar{x} = 0$. Therefore, we have to test $\bar{x}$ for being zero in the next line.

The idea for the **construction of** $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$ is the following: If $x > 0$, this can be verified by incrementing and decrementing again. If $x = 0$, we have $\bar{x} = 2^{2^n}$, which can be verified by decrementing $\bar{x}$ by 2^{2^n}. We non-deterministically guess which is the case. Execution in which the wrong choice is picked block.

Assume that we had already constructed a subroutine $\texttt{Dec}_n x$ that decrements x by 2^{2^n}.

5.15 Definition: Specification of the subroutine $\texttt{Dec}_n$

- The routine uses an auxiliary variable s_n.
- If the initial value of s_n is strictly less then 2^{2^n}, any execution of $\texttt{Dec}_n$ will get stuck.
- If the value of s_n is at least 2^{2^n}, then all *executions of* $\texttt{Dec}_n$ *that reach a return command* have the effect

$$s_n \leftarrow s_n - 2^{2^n}, \quad \bar{s}_n \leftarrow \bar{s}_n + 2^{2^n},$$

  and there is at least one such execution.
- There are no other side effects.

Using this subroutine, we finally define $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$.

5.16 Definition: Macro $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$

	$\texttt{goto}\ \ell_{positive}$ or $\texttt{goto}\ \ell_{loop}$;	// guess nondeterministically
$\ell_{positive}$:	$x{-}{-}; x{+}{+};$	// verify $x > 0$
	$\texttt{goto}\ \ell_{nz}$;	// verified non-zero
ℓ_{loop}:	$\bar{x}{-}{-}; x{+}{+};$	// move $\bar{x}$ to s_n
	$s_n{+}{+}; \bar{s}_n{-}{-};$	
	$\texttt{goto}\ \ell_{exit}$ or $\texttt{goto}\ \ell_{loop}$;	// guess whether moving is finished
ℓ_{exit}:	$\texttt{call}\ \texttt{Dec}_n$;	// check whether $s_n = 2^{2^n}$
	$\texttt{goto}\ \ell_z$;	// verified zero

Note that even if $x = 0$ does hold, the execution of $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$ might get stuck if in the loop, the value on $\bar{x}$ is not completely moved to s_n.

It remains to construct the subroutine $\texttt{Dec}_n$. We will do this inductively, i.e. we will first define $\texttt{Dec}_0$ and then construct $\texttt{Dec}_{i+1}$, assuming that we have already defined $\texttt{Dec}_i$. The specification of each $\texttt{Dec}_i$ is similar to the specification of $\texttt{Dec}_n$, with 2^{2^n} replaced

by 2^{2^i}. In the definition of $\texttt{Dec}_{i+1}$, we will use $\texttt{Test}'_i(x, \ell_z, \ell_{nz})$, which is defined just like $\texttt{Test}'_n(x, \ell_z, \ell_{nz})$, but it calls $\texttt{Dec}_i$ instead of $\texttt{Dec}_n$.

In the base case, we need to decrement s_n by by $2^{2^0} = 2^1 = 2$. This is done by the following routine.

5.17 Definition: Subroutine $\texttt{Dec}_0$

$$
\begin{array}{l}
s_0\texttt{--};\\
s_0\texttt{--};\\
\bar{s}_0\texttt{++};\\
\bar{s}_0\texttt{++};\\
\texttt{return};
\end{array}
$$

Assume we have already constructed $\texttt{Dec}_i$, a program decrementing s_n by 2^{2^i}, and $\texttt{Test}'_i(x, \ell_z, \ell_{nz})$. We now show how to construct $\texttt{Dec}_{i+1}$, a program decrementing by $2^{2^{i+1}}$.

We use the following trick:

$$2^{2^{i+1}} = 2^{2 \cdot 2^i} = 2^{2^i + 2^i} = 2^{2^i} \cdot 2^{2^i}.$$

In other words: To decrement by $2^{2^{i+1}}$, we decrement 2^{2^i} times by 2^{2^i}.

We implement this using two nested loops. More precisely, we use loop variables y_i and z_i that are initially set to 2^{2^i}. Each execution of the loop body of the inner loop decrements z_i as well as s_n by one. As soon as z_i hits 0, one execution of the loop body of the outer loop is finished, and we decrease y_i by one. When y_i hits 0, we have executed the outer loop 2^{2^i} times and have successfully decremented s_n by $2^{2^i} \cdot 2^{2^i} = 2^{2^{i+1}}$.

5.18 Definition: Subroutine $\texttt{Dec}_{i+1}$
Assume that initially, we have $y_i = z_i = 2^{2^i}$ and $\bar{y}_i = \bar{z}_i = 0$. The initialization phase will initialize these variables accordingly.

ℓ_{outer}:	$y_i\texttt{--}; \bar{y}_i\texttt{++};$	// one execution of outer loop starts
ℓ_{inner}:	$z_i\texttt{--}; \bar{z}_i\texttt{++};$	// one execution of inner loop starts
	$s_{i+1}\texttt{--}; \bar{s}_{i+1}\texttt{++};$	// the crucial decrement
	$\texttt{Test}'_i(z_i, \ell_{innerdone}, \ell_{inner});$	// check whether inner loop if finished
$\ell_{innerdone}$:	$\texttt{Test}'_i(y_i, \ell_{outerdone}, \ell_{outer});$	// check whether outer loop if finished
$\ell_{outerdone}$:	$\texttt{return}$	// decremented by $2^{2^i} \cdot 2^{2^i}$

Executing subroutine $\texttt{Dec}_{i+1}$ is possible without getting stuck if we initially have $s_i = 2^{2^{i+1}}$ and $\bar{s}_i = 0$.

Note that after the inner loop has been finished, we have moved the tokens from z_i to $\bar{z}_i$, i.e. we have $\bar{z}_i = 2^{2^i}$ and $z_i = 0$. As discussed earlier, $\texttt{Test}'_i(z_i, \ell_{innerdone}, \ell_{inner})$ swaps the values of z_i and $\bar{z}_i$ so that the variables are prepared for the next iteration outer loop.

Similarly, after the outer loop has finished, y_i and $\bar{y}_i$ are swapped, which is undone by $\texttt{Test}'_i(y_i, \ell_{outerdone}, \ell_{outer})$ so that the variables can be reused in the next call of $\texttt{Dec}_{i+1}$.

We can finally combine everything and define np_{sim}.

5.19 Definition: Program np_{sim}
The program np_{sim} consists of the subroutines $\texttt{Dec}_0, \ldots, \texttt{Dec}_n$ and the given counter program, modified as follows:

- Each increment $x{+}{+}$ is replaced by $x{+}{+}; \bar{x}{-}{-}$.
- Each decrement $x{-}{-}$ is replaced by $x{-}{-}; \bar{x}{+}{+}$.
- Each zero test $\texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then goto}\ \ell_z\ \texttt{else goto}\ \ell_{nz}$ is replaced by the code of the macro $\texttt{Test}_n(x, \ell_z, \ell_{nz})$ as defined above.

The construction of the initialization, np_{init}:

The initialization has to set the variables to the values required by the simulation.

- $x_1, \ldots, x_k$ already have initial value 0.
- For each i, s_i, $\bar{y}_i$ and $\bar{z}_i$ already have initial value 0.
- $\bar{x}_1, \ldots, \bar{x}_k$ need to be initialized to 2^{2^n}.
- For each $i \in \{0, \ldots, n\}$, $\bar{s}_i$ needs to be initialized to 2^{2^i}.
- For each $i \in \{0, \ldots, n-1\}$, y_i and z_i need to be initialized to 2^{2^i}.

Note that in $\texttt{Dec}_n$, we only use y_{n-1} and z_{n-1}, so we do not need the counters y_n and z_n.

We will define for each i a macro $\texttt{Inc}_i(v_1, \ldots, v_m)$ that increments the values of $v_1, \ldots, v_m$ by 2^{2^i}. Assume we had done this. Then we can define the initialization program as follows.

5.20 Definition: Program np_{init}
The program np_{init} is

$$
\begin{aligned}
&\texttt{Inc}_0(\bar{s}_0, y_0, z_0);\\
&\texttt{Inc}_1(\bar{s}_1, y_1, z_1);\\
&\vdots\\
&\texttt{Inc}_{n-1}(\bar{s}_{n-1}, y_{n-1}, z_{n-1});\\
&\texttt{Inc}_n(\bar{s}_n, \bar{x}_1, \dots, \bar{x}_k);
\end{aligned}
$$

It remains to construct for each i the macro $\texttt{Inc}_i(v_1, \dots, v_m)$. We proceed similar to the definition of $\texttt{Dec}_i$.

5.21 Definition: Macro $\texttt{Inc}_i(v_1, \dots, v_m)$
The program $\texttt{Inc}_0(v_1, \dots, v_m)$ is

$$
\begin{aligned}
&v_1\texttt{++}; v_1\texttt{++};\\
&\vdots\\
&v_m\texttt{++}; v_m\texttt{++};
\end{aligned}
$$

It increments each v_j by $2 = 2^1 = 2^{2^0}$.

For each i, the program $\texttt{Inc}_{i+1}(v_1, \dots, v_m)$ is defined as follows.

$$
\begin{array}{rll}
\ell_{outer}: & y_i\texttt{--}; \bar{y}_i\texttt{++}; & \text{// one execution of outer loop starts}\\
\ell_{inner}: & z_i\texttt{--}; \bar{z}_i\texttt{++}; & \text{// one execution of inner loop starts}\\
 & v_1\texttt{++}; v_1\texttt{++}; & \text{// the crucial increments}\\
 & \vdots & \\
 & v_m\texttt{++}; v_m\texttt{++}; & \\
 & \texttt{Test}'_i(z_i, \ell_{innerdone}, \ell_{inner}); & \text{// check whether inner loop if finished}\\
\ell_{innerdone}: & \texttt{Test}'_i(y_i, \ell_{outerdone}, \ell_{outer}); & \text{// check whether outer loop if finished}\\
\ell_{outerdone}: & \text{// here, the next part of the program should continue} &
\end{array}
$$

Note that in $\texttt{Inc}_{i+1}(v_1, \dots, v_m)$, we use $\texttt{Test}'_i(z_i, \ell_{innerdone}, \ell_{inner})$. This requires that the variables $\bar{s}_j, y_j, z_j$ for $j \leq i$ are already initialized. This is the case, as when $\texttt{Inc}_{i+1}(v_1, \dots, v_m)$ is used in np_{init}, the $\texttt{Inc}_j(\bar{s}_j, y_j, z_j)$ that perform that initialization have already been executed.

Furthermore, $\texttt{Inc}_{i+1}(v_1, \dots, v_m)$ manipulates the variables y_i and z_i. Note that the calls of $\texttt{Test}'_i$ (respectively the subsequent calls of $\texttt{Dec}_i$ will only use y_j and z_j for $j \leq i - 1$, so this is not a problem.

Complexity analysis

It remains to consider the resulting PN program and show that its size is indeed in $\mathcal{O}(n)$. It consists of several parts:

- The program for the initialization phase uses $\texttt{Inc}_0, \ldots, \texttt{Inc}_n$.

  The $\texttt{Inc}_0, \ldots, \texttt{Inc}_{n-1}$ increment 3 variables each, so they are of size constant in n and their total size is in $\mathcal{O}(n)$. $\texttt{Inc}_n$ increments $k + 1$ variables, and $k \leqslant n$, so its size is in $\mathcal{O}(n)$.

- The program for the simulation phase is obtained by replacing each command of the counter program by a constant number of commands. Its total size is in in $\mathcal{O}(n)$.

- The code for the subroutines $\texttt{Dec}_0, \ldots, \texttt{Dec}_n$ is of constant size each. Their total size is in $\mathcal{O}(n)$.

Adding everything, we obtain that we can simulate a counter program of size n with counters bounded by 2^{2^n} by a PN program of size $\mathcal{O}(n)$. The size of the associated Petri net is in $\mathcal{O}\left(n^2\right)$. This finishes the proof of Theorem 5.5. Together with Theorem 5.2, we obtain the desired result Theorem 5.1.

We conclude that Petri net coverability is EXPSPACE-hard. Coverability can be easily reduced to reachability, so reachability is also EXPSPACE-hard, see Exercise 3.3.

6. Petri net reachability

In this section, we want to study the Petri net reachability problem, proving that it is decidable.

Recall Definition 2.12

Definition: Petri net reachability

Petri net reachability (PNREACH)
Decide: Petri net N, initial marking M_0, final marking M_f
Decide: Is there a firing sequence $\sigma \in T^*$ such that $M_0 \xrightarrow{\sigma} M_f$?

6.1 Remark: The history of the Petri net reachability problem
The history of the Petri net reachability problem is a long one and it does not yet have a happy end. Petri nets were introduced by Carl Adam Petri (PhD thesis "Kommunikation mit Automaten" 1962, some sources claim he invented Petri nets 1939 at the age of 13). For a long time, it was unclear whether the Petri net reachability problem is decidable, i.e. whether there is an algorithm to solve it.

When complexity theory arose in the 1960s, it became clear that Petri net reachability is at least PSPACE-hard. This means that any algorithm solving it requires at least a polynomial amount of space, and, unless P = PSPACE holds, a superpolynomial amount of time. In 1976, Lipton has proven that it is even EXPSPACE-hard [Lip76], i.e. any algorithm solving it requires at least an exponential amount of space, and, unless EXP = EXPSPACE, a superexponential amount of time. (These lecture notes contain a proof of Lipton's result based on the presentation in [Esp98], see Theorem 5.1.) At this time, it was still not clear whether such an algorithm actually exists. To quote Lipton himself: "My theorem would have been wiped out, if someone had been able to prove that the reachability problem was undecidable." [Lip09].

In 1977, Sacerdote and Tenney gave a partial proof of decidability [ST77]. In 1981, this proof was completed by Mayr [May81], finally proving that Petri net reachability is decidable. As the proof was highly complicated, simplified versions were later published by Kosaraju [Kos82] and Lambert [Lam92]. All these proofs rely on a decomposition of the reaching firing sequences, later dubbed Kosaraju-Lambert-Mayr-Sacerdote-Tenney (KLMST) decomposition by Leroux.

Recently, Leroux has done a lot of work on Petri net reachability. In 2009, he published a proof of decidability [Ler09; Ler10] that uses the techniques from the previous proofs (Mayr, Kosajaru, Lambert), but obtains a different algorithm. He shows that if the final marking is not reachable, then there is a forward-inductive invariant, a set of a special

shape containing all reachable markings but not the final marking. Forward-inductive invariants can be shown to have a finite representation, so if an invariant exists, it can be found by brute-force enumeration. This yields a semi-algorithm for unreachability which then can be combined with a semi-algorithm for reachability, e.g. one that enumerates all computations.

In 2011, he published a different proof [Ler11b; Ler11a] that results in the same algorithm, but obtains the fact that an forward-inductive invariant has to exist without relying on the KLMST decomposition. Later, he published a simplified version of this alternative proof [Ler12]. (See also a later article of him together with Finkel on the proofs using inductive invariants [FL14; FL15].)

Until 2015, the exact time complexity of Mayr's algorithm was unknown, but it was clear that the KLMST decompositionmay need non-primitive recursive time. In 2015, Leroux and Schmitz [LS15]. proved that the algorithm is what they call *cubic Ackermann*, i.e. roughly the Ackermann function applied to itself applied to itself applied to the size of the net

The fact that even 30 years after Mayr's proof, new proofs for a solved problem are published at the best conferences shows on the one hand how complicated the original proof is, and on the other hand that the interest in the topic is unbroken. Closing the huge gap between the EXPSPACE lower bound and the non-primitive recursive upper bound remains one of the biggest open problems of Theoretical Computer Science.

The goal of this section is to prove the following theorem.

6.2 Theorem: [May81; Kos82; Lam92; Ler09; Ler10; Ler11b; Ler11a; Ler12]
Petri net reachability is decidable.

Sources
The proof presented here is an adapted version of Lambert's proof [Lam92].

In the following let $N = (P, T, in, out)$ with initial marking M_0 and final marking M_f be the Petri net instance of interest.

Generalized Markings

6.3 Remark
A **generalized marking** for a net is an element of $\mathbb{N}_\omega^d$, where $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$. The natural

order $\leqslant$ on $\mathbb{N}$ is extended to $\mathbb{N}_\omega$ by setting $n \leqslant \omega$ for all $n \in \mathbb{N}_\omega$. The (strict) product order on $\mathbb{N}_\omega^d$ is as usual:

$$M \leqslant M' \quad \text{iff} \quad M(p) \leqslant M'(p) \forall p \in P\,,$$

$$\begin{aligned} M < M' \quad &\text{iff} \quad M \leqslant M' \text{ and } M \neq M' \\ &\text{iff} \quad \forall p \in P \colon M(p) \leqslant M'(p) \text{ and } \exists p' \in P \colon M(p) < M'(p)\,. \end{aligned}$$

We extend the firing relation to generalized markings. We have

$$M \xrightarrow{t} M' \quad \text{iff} \quad M' = M + e(t) = M - in(t) + out(t)\,.$$

Here, the operations plus and minus should be read component-wise, and they are extended to $\mathbb{N}_\omega^d$ by setting $\omega + n = \omega - n = \omega$ for all $n \in \mathbb{N}$. (The cases $\omega + \omega$ and $\omega - \omega$ can remain undefined as they will never occur.)

As we are interested in reachability and not in coverability, the product order $\leqslant$ on $\mathbb{N}_\omega^d$ is too imprecise. Instead, we define a new order $\leqq_\omega$ on $\mathbb{N}_\omega^d$ as follows.

6.4 Definition
For two generalized markings $M, M' \in \mathbb{N}_\omega^d$, we have

$$M \leqq_\omega M' \quad \text{iff} \quad \forall p \in P \text{ with } M'(p) < \omega \colon M(p) = M'(p)\,.$$

We say that M is **under** M'.

In words: Whenever a component of M' is not ω, it coincides with the corresponding component of M. For ω-components of M', the corresponding components of M may be arbitrary. This means that we may introduce new ω-componments along $\leqq_\omega$.

6.5 Lemma

a) $\leqq_\omega$ is a partial order, i.e. reflexive, antisymmetric and transitive.

b) $\leqq_\omega$ is **monotonic** in the following sense: Let $M \leqq_\omega M'$ and let $M'' \in \mathbb{N}_\omega^P$. Then we have $M + M'' \leqq_\omega M' + M''$.

Covering graphs

We will now introduce covering graphs, a standard tool to decide the coverability problem. Here, we will define the coverability graph along a graph that acts a finite control.

6.6 Remark

Recall that a **finite** T**-(arc)-labeled directed graph**, just called **graph** in the following, is a tuple

$$G = (V, R)$$

where V is finite set of vertices, $R \subseteq V \times T \times V$ is the set of labeled arcs.

We write $q \xrightarrow{t}_G q'$ if $r = (q, t, q') \in R$. If the graph is clear from the context, we omit the subscript G and write just $q \xrightarrow{t} q'$.

A **path** in G is a sequence of vertices and transitions

$$q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} q_n \,.$$

We call $\sigma = t_1 \ldots t_n$ the word of labels along the path.

We write $q_0 \xrightarrow{\sigma} q_n$ if there is a path from q_0 to q_n labeled by σ.

For some vertex q_i, the **trace language of** G **from** q_i is

$$\mathcal{L}(G, q_0) = \left\{\sigma \in T^* \;\middle|\; q_i \xrightarrow{\sigma} q_n \text{ for some } q_n \in V\right\},$$

the set of all sequences that occur as labels along paths from q_i, no matter where the path ends.

For some vertices $q_i, q_f \in V$ the **language of paths or reachability language from** q_i **to** q_f is

$$\mathcal{L}(G, q_i, q_f) = \left\{\sigma \in T^* \;\middle|\; q_0 \xrightarrow{\sigma} q_f\right\},$$

the set of all sequences that occur as labels along paths from q_i to q_f.

Similarly, for a Petri net N and markings M, M', we define the trace lanugage

$$\mathcal{L}(N, M) = \{\sigma \in T^* \mid M [\sigma\rangle M'' \text{ for some } M''\}$$

and the **reachability language**

$$\mathcal{L}(N, M, M') = \{\sigma \in T^* \mid M [\sigma\rangle M'\}$$

It is no coincidence that we have used T as the set of labels in the remark above. We will indeed by interest in graphs whose transitions are labeled by transitions of the Petri net.

6.7 Definition

Let N be a Petri net together with a generalized (!) initial marking M_i and let G be a T-labeled graph together with a vertex q_i. A **covering graph for** N **from** M_i **along** G **from** q_i is a directed, T-arc-labeled graph $\mathcal{G} = (V, R)$ that is obtained from an execution of Algorithm 6.8 below. Here, $V \subset Q \times \mathbb{N}_\omega^P$ is a finite set of vertices of the shape (q, M) and $R \subseteq V \times T \times V$ are arcs labeled by transitions of the Petri net.

6.8 Algorithm: Computing a covering graph

Input: N Petri net, M_i generalized marking, G graph, q_i vertex

Output: Graph $\mathcal{G}$

Initialize $\mathcal{G}$ as the empty tree // We first create a tree

Create an unmarked vertex labeled by (q_i, M_i).

while There is an unmarked vertex, say v labeled by (q, M) **do**

  Mark v

  for all $q \xrightarrow{t} q'$ in A (for some t, q') **do**

    if $M [t\rangle M_t$ (in particular, t is enabled in M) **then**

      Define $M' \in \mathbb{N}_\omega^P$ by

$$M'(p) = \begin{cases} \omega\,, & \text{if there is an ancestor of } v \text{ labebled by } (q', M_a) \\ & \text{with } M_a \leqslant M_t \text{ and } M_a(p) < M_a(p)\,, \\ M_t(p)\,, & \text{else}\ . \end{cases}$$

      Add a new vertex v' with label (q', M')

      Add an arc (v, v') labeled by t

      if v' has a (strict) ancestor with the same label **then**

        Mark v // Do not consider it again

      end if

    end if

  end for

end while

Merge vertices that have the same label // Convert the tree

return $\mathcal{G}$

Depending on the order in which we pick the vertices and transitions during the algorithm, we might end up with a different graph $\mathcal{G}$. Let $CG(N, M_i, G, q_i)$ denote the set of all possible covering graphs.

In the following, we will always rely on the following properties that are independent from the element of $CG(N, M_i, G, q_i)$ that we pick.

6.9 Proposition: Classical properties of covering graph
Let $\mathcal{G} \in CG(N, M_i, G, q_i)$.

a) We may compute one member of $CG(N, M_i, G, q_i)$.

b) Each graph in $CG(N, M_i, G, q_i)$ is finite.

c) For any arc $(q, M) \xrightarrow{t}_{\mathcal{G}} (q', M')$ in $\mathcal{G}$, we have $M \, [t\rangle \, M_t$ with $M_t \leqq_\omega M'$.

d) If $\sigma \in \mathcal{L}(N, M_i) \cap \mathcal{L}(G, q_i, q')$, then we have $\sigma \in \mathcal{L}(\mathcal{G}, (q_i, M_i), (q', M'))$ for some M' with $M_i + e(\sigma) \leqq_\omega M'$.

e) For each vertex (q, M) of $\mathcal{G}$ and each number $n \in \mathbb{N}$, we can compute $\sigma_n \in \mathcal{L}(N, M_i) \cap \mathcal{L}(G, q_i, q)$ such that $M_i \, [\sigma_n\rangle \, M_n$ for some M_n with

$$M(p) \neq \omega \implies M_n(p) = M(p) \,,$$
$$M(p) = \omega \implies M_n(p) \geq n \,.$$

Proof:
a) and b) are due to the fact that $\leq$ on $\mathbb{N}_\omega^d$ is a well-quasi ordering.

c) is by the construction of the arcs in $\mathcal{G}$, and d) is obtained from c) using induction.

e) Take a path from (q_i, M_i) to (q_i, M_i) in $\mathcal{G}$. By inserting pumps, we obtain a firing sequence that is enabled in M_i. Insert pumps appropriately to get the desired σ_n. □

Intuitively, c) and d) state that the arcs of the covering graph are an overapproximation of the behavior of the Petri net: For the non-ω components, the covering graph actually provides the correct behavior, but it may introduce ω-components.

In turn, e) states that whenever the covering graph introduces an ω in some component, there is actually a firing sequence in this Petri net that brings the component to an arbitrarily high value.

6.10 Definition & Proposition: Covering
Let $\mathcal{G} \in CG(N, M_i, G, q_i)$.

a) There is a vertex (q_i, M) in $\mathcal{G}$ such that M is the largest marking over M_i in q_i, meaning for any vertex (q_i, M') of $\mathcal{G}$, $M_i \leqq_\omega M'$ implies $M' \leqq_\omega M$.

b) We have $M_i \leqq_\omega M$.

c) M is independent of the choice of $\mathcal{G} \in CG(N, M_i, G, q_i)$.

We call M the the **covering** of (N, M_i, G, q_i), denoted by $C(N, M_i, G, q_i)$.

Proof:
We will show that given two markings M', M'' over M_i from arbitrary covering graphs, we can compute one in $\mathcal{G}$ that dominates both. Applying this fact inductively yields all statements of the proposition.

Pick M', M'' such that $(q_i, M'), (q_i, M'')$ are vertices of some $\mathcal{G}', \mathcal{G}'' \in CG(N, M_i, G, q_i)$ and $M_i \leqq_\omega M'$, $M_i \leqq_\omega M''$.

We prove that there is a vertex (q_i, M) of $\mathcal{G}$ such that $M' \leqq_\omega M, M'' \leqq_\omega M$.

By Part e) of Proposition 6.9, for any $n \in \mathbb{N}$, we may pick sequences $\sigma', \sigma'' \in \mathcal{L}(G, q_i, q_i)$ such that

- $M_i \xrightarrow{\sigma'} M_1$, where $M_1(p) = M_i(p)$ if $M'(p) = M_i(p)$, and $M_1(p) > M_i(p) + n$ else,
- $M_i \xrightarrow{\sigma''} M_2$, where $M_2(p) = M_i(p)$ if $M''(p) = M_i(p)$, and $M_2(p) > M_i(p) + n$ else,

Consider these sequences for some n that is larger than any finite (non-ω) number occurring in a vertex of $\mathcal{G}$. (Meaning it is larger than any number $\tilde{M}(p) \neq \omega$ for any vertex $(\tilde{q}, \tilde{M})$ in $\mathcal{G}$.)

Now consider the marking M_3 with $M_i \xrightarrow{\sigma'\sigma''} M_3$. By Part d) of Proposition 6.9, we have a path in $\mathcal{G}$ from (q_i, M_i) to some (q_i, M) with

$$M_3 = M_i + e(\sigma') + e(\sigma'') \leqq_\omega M .$$

We have that if $M'(p)$ or $M''(p)$ is ω, then $M_3(p) \geqslant n$. Since n is larger than any number occurring in $\mathcal{G}$, we need to have have $M(p) = \omega$.

If $M'(p)$ and $M''(p)$ are not ω, we have

$$M_i(p) = M'(p) = M_1(p) \quad \text{and} \quad M_i(p) = M''(p) = M_2(p)$$

and consequently $M_i(p) = M_3(p)$. We conclude that $M_i \leqq_\omega M$, $M' \leqq_\omega M$, and $M'' \leqq_\omega M$ as desired.

This allows us to show that there is a largest marking M over M_i in $\mathcal{G}$ by considering the finite set of markings over M_i in $\mathcal{G}$ and applying the proof inductively.

Now assume that there is some other $\mathcal{G}''$ for which this largest marking is different, say $\overline{M}$. We apply the proof above again to construct a marking $\overline{\overline{M}}$ in $\mathcal{G}$ that is even larger. This yields a contradiction to the construction of M unless $M = \overline{\overline{M}} = \overline{M}$. □

6.11 Definition: Covering sequences
Let N be a Petri net with a generalized initial marking M_i, and G a graph with a vertex q_i Let $M = C(N, M_i, G, q_i)$ be the covering of (N, M_i, G, q_i).

We call a sequence $\sigma \in \mathcal{L}(G, q_i, q_i) \cap \mathcal{L}(N, M_i)$ such that

$$\text{for all } p \text{ with } M_i(p) \neq \omega: \quad e(\sigma, p) \geqslant 0$$
$$\text{and} \quad e(\sigma, p) > 0 \text{ iff } M(p) = \omega$$

a **covering sequence**.

We denote by $CS(N, M_i, G, q_i)$ the set of all covering sequences.

Intuitively spoken, a covering sequence σ has

- arbitrary effect on the ω-components of M_i,
- strictly positive effect on the ω-components of M,
- zero effect on the remaining components.

6.12 Proposition
$CS(N, M_i, G, q_i)$ is non-empty and we may compute one of its elements.

Proof:
We apply Part e) of Proposition 6.9 to the vertex (q_i, M) and $n = (\max_{p \in P} M_i(p)) + 1$. We obtain that we can compute a firing sequence $\sigma \mathcal{L}(G, q_i, q)$ such that $M_i \boxed{\sigma\rangle} M'$ and for all p, $M(p) \neq \omega$ implies $M'(p) = M(p) = M_i(p)$ and $M(p) = \omega$ implies $M'(p) \geqslant n$.

Consider a component p such that $M(p) \neq \omega$. In this case, we also have $M_i(p) \neq \omega$ since $M_i \leqq_\omega M$. We have $M'(p) = M(p) = M_i(p)$ and conclude $e(\sigma, p) = 0$.

For p with $M(p) = \omega$, we have $M'(p) \geqslant n > M_i(p)$. This implies $e(\sigma, p) > 0$ as desired. □

Precovering graphs

6.13 Remark
Let $G = (V, R)$ be a directed graph.

The **strongly connected component (SCC)** of a vertex $q \in V$ is the subgraph induced by all vertices q' such that there is a path from q to q' and a path from q' to q. Note that q is one such a vertex, consequently, each SCC is non-empty.

The graph G is called **strongly connected** if the graph itself is a SCC of one (and then all) of its vertices.

6.14 Definition

Let N be a Petri net. A **precovering graph on** N is a strongly connected, finite, directed, T-arc-labeled graph $G = (V, R)$ with $V \subseteq \mathbb{N}_\omega^P$ if for all

$$m \xrightarrow{t}_G m' \text{ in } G, \qquad \text{we have } m\,[t\rangle\, m_t \text{ with } m_t \leqq_\omega m' \;.$$

In other words, the edges of G are an overapproximation of the firing relation that is precise on the non-ω components, but may introduce new ω components We will now see that actually, no new ω components can be introduced.

6.15 Definition & Proposition

Let G be a precovering graph for N. For each place p, $m(p)$ is either ω in all vertices of G, or in none of them.

We may define

$$\Omega(G) = \{p \in P \mid \forall m \in V\colon m(p) = \omega\} \;,$$

the set of **ω-components in** G.

Proof:

Assume there is a component p such that there are vertices m, m' of G with $m(p) = \omega \neq m'(p)$. Since G is strongly connected by definition, there is a path

$$m = m_{(0)} \rightarrow_G m_{(1)} \rightarrow_G \ldots \rightarrow_G m_{(k)} = m' \;.$$

By the property of the edges in G, we obtain that m' has more ω-components than m, a contradiction. □

6.16 Corollary

Let G be a precovering graph for N. For any edge $m \xrightarrow{t}_G m'$ in G, we have $m\,[t\rangle\, m'$.

For any two vertices m, m' and any $\sigma \in \mathcal{L}(G, m, m')$, we have $m\,[\sigma\rangle\, m'$.

The first decomposition result shows that subgraphs of precovering graphs are again precovering graphs.

6.17 Lemma

Any strongly-connected subgraph of a precovering graph is again a precovering graph.

Proof: Clear from the definition. □

The second decomposition result relates covering graphs and precovering graphs.

6.18 Proposition

Let G be a precovering graph for N and let m_i be a vertex. Let $M_i \in \mathbb{N}_\omega^P$ with $M_i \leqq_\omega m_i$. Consider a covering graph $\mathcal{G} \in CG(N, M_i, G, m_i)$ of N along G from m_i.

a) All vertices (m, M) of $\mathcal{G}$ satisfy $M \leqq_\omega m$.

b) The projection

$$\begin{array}{rrcl} \pi_2 & : \quad V(\mathcal{G}) & \rightarrow & \mathbb{N}_\omega^P \\ & (m, M) & \mapsto & M \end{array}$$

is injective.

c) Each SCC of the the graph $\pi_2(\mathcal{G})$ is a precovering graph for N.

The graph $\pi_2(\mathcal{G})$ is obtained by projecting all vertices to their second component and leaving the arcs unchanged. Since the projecting is injective, it cannot happen that $\pi_2(\mathcal{G})$ has multiple vertices with the same label.

Proof:

Let (m, M) be a vertex of $\mathcal{G}$.

By Part e) of Proposition 6.9, for any n, there is $\sigma_n \in \mathcal{L}(G, m_i, m)$ such that we have $M_i \xrightarrow{\sigma_n} M_n$ with $M(p) \neq \omega$ implies $M_n(p) = M(p)$ and $M(p) = \omega$ implies $M_n(p) \geqslant n$.

By Corollary 6.16, $\sigma_n \in \mathcal{L}(G, m_i, m)$ implies $m_i \xrightarrow{\sigma_n} m$. Thus,

$$M_n = M_i + e(\sigma_n) \leqq_\omega m_i + e(\sigma_n) = m \; .$$

Here, we have used that $\leqq_\omega$ is monotonous, Lemma 6.5.

The non-ω components of m_i coincide with the corresponding components of M_i since $M_i \leqq_\omega m_i$. Consequently, the non-omega components of m coincide with the corresponding components of all M_n, which in turn coincide with the corresponding components of M. We conclude $M \leqq_\omega m$.

Assume that the projecting is not injective, i.e. there are vertices (m, M) and (m', M) with $m \neq m'$. By part a), we have $M \leqq_\omega m$ and $M \leqq_\omega m'$. Because m and m' are vertices of a precovering graph, they have the same ω-components, Proposition 6.15. Additionally,

the non-ω components coincide since they coincide with the corresponding component of M each. We conclude $m = m'$, a contradiction.

A strongly connected component of $\pi_2(\mathcal{G})$ is finite, directed, strongly-connected and T-labeled. It remains to check the property of the arcs. Any arc $M \xrightarrow{t} M'$ in $\pi_2(\mathcal{G})$ is induced by some arc $(m, M) \xrightarrow{t} (m', M')$ in $\mathcal{G}$. By Part c) of Proposition 6.9, we have that $M \boxed{t}\!\!> M_t$ with $M_t \leqq_\omega M'$, which is exactly as desired. □

We will now be interested in **initiated precovering graphs (IPGs)**, tuples (G, m) where G is a precovering graph for N and m is a vertex.

Decomposing precovering graphs

6.19 Remark

Let (V, R) be a graph and $q_i, q_f \in V$. Any path from q_i to q_f can be obtained from a cycle-free path (i.e. a path in which no intermediary vertex is repeated) by inserting cycles, i.e. paths from q to q for some q in the appropriate places.

6.20 Proposition: First IPG decomposition

Let (G, m) be an IPG for N. Let M be a generalized marking with $M \leqq_\omega m$.

- If $m = C(N, M, G, m)$:
  For any $\sigma \in CS(N, M, G, m)$, $\tau \in \mathcal{L}(G, m, m)$, there are integers k_τ, k'_τ such that for any $k \in \mathbb{N}$

$$k \geqslant k_\tau \implies M \boxed{\sigma^k\tau}\!\!>$$
$$k \geqslant k'_\tau \implies \sigma^k\tau \in CS(N, M, G, m)\ .$$

- If $m \neq C(N, M, G, m)$:
  We can compute a finite subset $\mathcal{L} \subseteq T^*$ (possibly empty) and for each $s = s_1 \dots s_n \in \mathcal{L}$ a sequence of IPGs

$$(G_0^s, m_0^s), (G_1^s, m_1^s) \dots (G_n^s, m_n^s)$$

  such that

$$M = m_0^s$$
$$\forall i{:}\Omega(G_i^s) \subsetneq \Omega(G)$$
$$\forall i{:}m_i^s \boxed{s_{i+1}}\!\!> m_i^s + e(s_{i+1}) \leqq_\omega m_{i+1}^s\ .$$

Furthermore, for all $\tau \in \mathcal{L}(G, m, m)$ with $M\,[\tau\rangle$, there is an $s = s_1 \dots s_n \in \mathcal{L}$ such that

$$\tau = \tau_{(0)} s_1 \tau_{(1)} s_2 \dots s_n \tau_{(n)}$$

for suitable $\tau_{(i)} \in \mathcal{L}\big(G_i^s, m_i^s, m_i^s\big)$.

Proof:

- Assume $m = C(N, M, G, m)$:

  We first show that $M\,[\sigma^k\tau\rangle$ for $k \geqslant k_\tau$. Since $M \leqq_\omega m$, we only need to worry about the components that are ω in m, but not in M. Recall that a covering sequence for m has positive effect on the ω-components of m and non-negative effect on the other components. Iterating σ often enough will load to a marking high enough so that τ becomes fireable.

  To show that $\sigma^k\tau \in CS(N, M, G, m)$ for $k \geqslant k'_\tau$, we need to show that

  - $\sigma^k\tau \in \mathcal{L}(G, m, m)$, which is true by definition,
  - $M\,[\sigma^k\tau\rangle$, which is true if we pick $k'_\tau \geqslant k_\tau$,
  - that $\sigma^k\tau$ has zero effect on the non-ω components of m, which is true since it is contained in $\mathcal{L}(G, m, m)$,
  - that is has strictly positive effect on the ω components of m, which is true for k large enough, since σ was a covering sequence.

- Assume $m \neq C(N, M, G, m)$:
  Compute $\mathcal{G} \in CG(N, M, G, m)$.

  Let $\tau \in \mathcal{L}(G, m, m)$ such that $M\,[\tau\rangle$. By Part d) of Proposition 6.9, we have $\tau \in \mathcal{L}(\mathcal{G}, (m, M), (m, M'))$ for some M' with $M + e(\tau) \leqq_\omega M'$.

  Define $\mathcal{L}_\tau$ as the set of all cycle-free paths from (m, M) to (m, M'). Let

  $$\pi = (m, M) \xrightarrow{s_1} \dots \xrightarrow{s_n} (m, M')$$

  be one such path.

  We define $s = s_1 \dots s_n$ as the transitions used along this path. We furthermore define the sequence of the m_i^s as the sequence of the second components in the path, in particular

  $$m_0^s = M, \quad m_n^s = M' .$$

For each i, let G_i^s be the maximal SCC of $\pi_2(\mathcal{G})$ containing m_i^s. Using $M \leqq_\omega m$ and Proposition 6.18, each (G_i^s, m_i^s) is indeed an IPG.

We have

$$m_0^s \xrightarrow{s_1} m_0^s + e(s_1) \leqq_\omega m_1^s \dots \xrightarrow{s_n} m_{n-1}^s + e(s_n) \leqq_\omega m_n^s = M'$$

by using Part c) of Proposition 6.9.

By showing that $M' = m_n^s$ has strictly less ω-components than m, we may conclude that each m_i^s has strictly less ω-components than m, yielding $\Omega(G_i^s) \subseteq \Omega(G)$.

First note that we have $M' \leqq_\omega m$ by Part a) of Proposition 6.18 since (m, M') is a vertex of $\mathcal{G}$. It remains to show that $M' \neq m$.

If $M' = m$, then we would have $M \leqq_\omega m = M'$. Consequently, we have $m \leqq_\omega C(N, M, G, m)$, since $C(N, M, G, m)$ is the largest vertex over the initial vertex (m, M). Furthermore, we have $C(N, M, G, m) \leqq_\omega m$, since $(m, C(N, M, G, m))$ is a vertex of $\mathcal{G}$, again by Part a) of Proposition 6.18. We conclude $m = CS(N, M_i, A, m_i)$, a contradiction to the assumption.

Finally, consider the path in $\mathcal{G}$ from (m, M) to (m, M') induced by τ. We may write this path as some cycle-free path π' with some cycles inserted at the appropriate places. Consider the element s' induced by π' as above. Using the fact that G itself was a precovering graph, we obtain the desired property.

To finish the proof, let $\mathcal{L}$ be the collection of all s obtained as above for all computation $\tau \in \mathcal{L}(\mathcal{G}, (m, M), (m, M'))$ with $M \xrightarrow{\tau}$. Because $\mathcal{G}$ contains only finitely many vertices that may be used as (m, M'), and for each such vertex there are only finitely many cycle-free paths, $\mathcal{L}$ is finite and can be computed.

□

6.21 Definition

a) The reverse of a Petri net $N = (P, T, in, out)$ is the Petri net$N^{\text{rev}} = (P, T, out, in)$.

b) For a sequence $\sigma = t_1 \dots t_n \in T^*$, its reverse is $\sigma^{\text{rev}} = t_n \dots t_1$.

c) Let $G = (V, R)$ be a graph. Its reverse $G^{\text{rev}} = (V, R^{\text{rev}})$ is obtained by inverting all arcs,

$$R^{\text{rev}} = \left\{ q' \xrightarrow{t}_{G^{\text{rev}}} q \;\middle|\; q \xrightarrow{t}_G q' \in R \right\}.$$

6.22 Lemma

Let N be a Petri net.

a) If $M \xrightarrow{\sigma} M'$ in N, then $M' \xrightarrow{\sigma^{\text{rev}}} M$ in N^{rev}.

b) If G is a precovering graph for N, then G^{rev} is a precovering graph for N^{rev}.

Proof: Immediate from the definitions. □

6.23 Proposition: Second IPG decomposition
Let (G, m) be an IPG for N. Let M be a generalized marking with $M \leqq_\omega m$.

- If $m = C\big(N^{\text{rev}}, M, G^{\text{rev}}, m\big)$:
  For any $\sigma^{\text{rev}} \in CS\big(N^{\text{rev}}, M, G^{\text{rev}}, m\big)$, $\tau \in \mathcal{L}\big(G^{\text{rev}}, m, m\big)$, there are integers k_τ, k'_τ such that for any $k \in \mathbb{N}$

$$\begin{aligned} k \geqslant k_\tau &\implies M \xrightarrow{(\sigma^{\text{rev}})^k \tau^{\text{rev}}} \\ k \geqslant k'_\tau &\implies (\sigma^{\text{rev}})^k \tau^{\text{rev}} \in CS\big(N^{\text{rev}}, M, G^{\text{rev}}, m\big)\,. \end{aligned}$$

- If $m \neq C\big(N^{\text{rev}}, M, G^{\text{rev}}, m\big)$:
  We can compute a finite subset $\mathcal{L} \subseteq T^*$ (possibly empty) and for each $s = s_1 \ldots s_n \in \mathcal{L}$ a sequence of IPGs

$$(G_0^s, m_0^s), (G_1^s, m_1^s) \ldots (G_n^s, m_n^s)$$

  such that

$$\begin{aligned} M &= m_n^s \\ \forall i{:}\Omega(G_i^s) &\subsetneq \Omega(G) \\ \forall i{:}m_i^s &\xrightarrow{s_i} m_i^s - e(s_i) \leqq_\omega m_{i-1}^s\,. \end{aligned}$$

  Furthermore, for all $\tau \in \mathcal{L}(G, m, m)$ with $M \xrightarrow{\tau^{\text{rev}}}$, there is an $s = s_1 \ldots s_n \in \mathcal{L}$ such that

$$\tau = \tau_{(0)} s_1 \tau_{(1)} s_2 \ldots s_n \tau_{(n)}$$

  for suitable $\tau_{(i)} \in \mathcal{L}\big(G_i^s, m_i^s, m_i^s\big)$.

Proof: Combine Lemma 6.22 with Proposition 6.20. □

Let π be a path in $G = (V, R)$. We define the occurrence vector $\Psi(\pi) \in \mathbb{N}^R$ as the vector that counts how often each arc is used. For a subset $E \subset R$, we let $\Psi(\pi)\restriction_E$ denote the vector in $\mathbb{N}^E$ obtained from $\Psi(\pi)$ by omitting components corresponding to arcs not in E.

6.24 Proposition: Third IPG decomposition
Let (G, m) be an IPG for N. Let $E \subset R(G)$ be a non-empty strict subset of the arcs of G and let $F \subset \mathbb{N}^E$ be a finite set of vectors. We can compute a finite subset $\mathcal{L} \subseteq T^*$ (possibly empty) and for each $s = s_1 \dots s_n \in \mathcal{L}$ a sequence of IPGs

$$(G_0^s, m_0^s), (G_1^s, m_1^s) \dots (G_n^s, m_n^s)$$

such that

$$\begin{aligned} m = m_0^s &= m_n^s \\ \forall i{:}\Omega(G_i^s) &= \Omega(G) \\ \forall i{:}\left|R(G_i^s)\right| &< |R(G)| \\ \forall i{:}m_i^s &\xrightarrow{s_{i+1}} m_{i+1}^s \ . \end{aligned}$$

Furthermore, for all $\tau \in \mathcal{L}(G, m, m)$ that occur as some path π such that $\Psi(\pi)\restriction_E \in F$, there is an $s \in \mathcal{L}$ such that

$$\tau = \tau_{(0)} s_1 \tau_{(1)} s_2 \dots s_n \tau_{(n)}$$

for suitable $\tau_{(i)} \in \mathcal{L}\left(G_i^s, m_i^s, m_i^s\right)$.

Proof:
Let $G = (V, R)$. Define $G' = (V, R \setminus E)$.

Consider $\tau \in \mathcal{L}(G, m, m)$ such that for a corresponding path π, we have $\Psi(\pi)\restriction_E \in F$. We may write

$$\pi = \pi_{(0)} \sigma_1 \pi_{(1)} \sigma_2 \dots \sigma_n \pi_{(n)}$$

where $\sigma = \sigma_1 \dots \sigma_n$ such that $\Psi(s)\restriction_E = \Psi(\pi)\restriction_E \in F$ and $\Psi(s)\restriction_{R \setminus E} = \vec{0}$, i.e. σ contains exactly the arcs used in π in E.

We define the sequences of vertices $(q_i')_{i \in \{0,\dots,n-1\}}$ and $(q_i)_{i \in \{0,\dots,n\}}$.

$$\begin{aligned} q_0 &= m \\ q_n &= m \\ s_i &= (q_{i-1}', q_i) \end{aligned}$$

By construction, each $\pi_{(i)}$ is a path from q_i to q_i' in the modified graph G'. Let us apply remark 6.19, so we may write each $\pi_{(i)}$ as a cycle-free path $\pi_{(i)}'$ from q_i to q_i' with cycles inserted appropriately.

We define

$$\sigma' = \pi'_{(0)}.\sigma_1.\pi'_{(1)}.\sigma_2 \ldots \sigma_n \pi'_{(n)} ,$$

as the path from m to m in G in which all cycles inside the $\pi_{(i)}$ have been removed. Let $s \in T^*$ be the associated sequence of transitions.

We define the sequence of the m_i^s as the sequence of the associated markings of G and we let G_i^s by the SCC of m_i^s in G'.

We have to check some properties:

- $m = m_0^s = m_n^s$:
  Clear.
- $\Omega(G_i^s) = \Omega(G') = \Omega(G)$:
  By Proposition 6.15.
- $m_i^s \xrightarrow{s_{i+1}} m_{i+1}^s$:
  By Corollary 6.16.
- $|R(G_i^s)| < |R(G)|$:
  We have

  $$|R(G_i^s)| \leqslant |R(G')| = |R(G) \setminus E| < |R(G)|$$

  since E is non-empty.
- The decomposition of τ is by construction.

It remains to argue that if we do this construction for all possible τ, we obtain a finite set $\mathcal{L}$ of all possible s. To this end, note that there are only finitely many possible σ since F is finite. Consequently, there are only finitely many different sequences $(q'_i)_{i \in \{0,\ldots,n-1\}}$ and $(q_i)_{i \in \{0,\ldots,n\}}$. For each of the finitely many parts (q_i, q'_i), there are only finitely many cycle free paths from q_i to q'_i in G'. □

Marked graph transition sequences

6.25 Definition

Let N be a Petri net A **graph-transition sequence** is sequence

$$\mathcal{U} = (G_0, m_0), t_1, (G_1, m_1), t_2, \ldots, t_n, (G_n, m_n)$$

where each $t_i \in T$ is a transition of N, and each (G_i, m_i) is an IPG for N.

A **marked graph-transition sequence (MGTS)** $(\mathcal{U}, \varphi)$ is a graph-transition sequence as above together with a function

$$\begin{array}{rccl} \varphi : & \{0, \ldots, n\} & \to & \mathbb{N}_\omega^P \times \mathbb{N}_\omega^P \\ & (G_i, m_i) \text{ resp. } i & \mapsto & (M_i^{\text{in}}, M_i^{\text{out}}) \end{array}$$

such that

$$M_i^{\text{in}} \leqq_\omega m_i \quad \text{and} \quad M_i^{\text{out}} \leqq_\omega m_i$$

for all i.

We call M_i^{in} resp. M_i^{out} the **input resp. output marking** of (G_i, m_i). We call $M^{\text{in}}(\mathcal{U}, \varphi) = M_0^{\text{in}}$ the input, $M^{\text{out}}(\mathcal{U}, \varphi) = M_n^{\text{out}}$ the output marking of $(\mathcal{U}, \varphi)$.

The language of a MGTS is the set of transitions sequences that contain the sequence of the t_i and such that the parts in the between are paths in the corresponding G_i that respect the input- and output marking. This is formalized by the following definition.

6.26 Definition
Let $(\mathcal{U}, \varphi)$ be a MGTS for Petri net n. The language $\mathcal{L}(\mathcal{U}, \varphi)$ is the set set of all sequences

$$\tau = \tau_{(0)} t_1 \tau_{(1)} t_2 \ldots t_n \tau_{(n)}$$

such that $\tau_{(0)} \in \mathcal{L}(G_i, m_i, m_i)$ and there are sequences $(\mu_i^{\text{in}})_{i \in \{0,\ldots,n\}}, (\mu_i^{\text{out}})_{i \in \{0,\ldots,n\}}$ in $\mathbb{N}^P$.

$$\mu_0^{\text{in}} \xrightarrow{\tau_{(0)}} \mu_0^{\text{out}} \xrightarrow{t_1} \mu_1^{\text{in}} \xrightarrow{\tau_{(1)}} \mu_1^{\text{out}} \xrightarrow{t_2} \ldots \mu_{n-1}^{\text{out}} \xrightarrow{t_n} \mu_n^{\text{in}} \xrightarrow{\tau_{(n)}} \mu_n^{\text{out}}$$

with $\mu_i^{\text{in}} \leqq_\omega M_i^{\text{in}}$, $\mu_i^{\text{out}} \leqq_\omega M_i^{\text{out}}$ for all i.

6.27 Example
Let (N, M_0, M_f) be a Petri net instance.

We consider the MGTS $(\mathcal{U}_0, \varphi_0)$, where $\mathcal{U}_0$ consists of the single IPG (G_0, m_0) and $\varphi(0) = (M_0, M_f)$. Here, $G_0 = \left(\{\vec{\omega}\}, \left\{\vec{\omega} \xrightarrow{t} \vec{\omega} \,\middle|\, t \in T\right\}\right)$ and $m_0 = \vec{\omega}$.

In other words, $\mathcal{U}_0$ is the trivial approximation of the Petri net.

We have that $\mathcal{L}(\mathcal{U}_0, \varphi_0)$ is the set of all sequences τ such that $\tau \in \mathcal{L}(G_0, \vec{\omega}, \vec{\omega})$ such that there are $\mu^{\text{in}}, \mu^{\text{out}}$ with

$$\mu^{\text{in}} \xrightarrow{\tau} \mu^{\text{out}}$$

and $\mu^{\text{in}} \leqq_\omega M^{\text{in}}(\mathcal{U}, \varphi) = M_0$ and $\mu^{\text{out}} \leqq_\omega M^{\text{out}}(\mathcal{U}, \varphi) = M_f$.

Now note that $\tau \in \mathcal{L}(G_0, \vec{\omega}, \vec{\omega})$ is satisfied by all $\tau \in T^*$. Since M_0, M_f are in $\mathbb{N}^P$ and contain no ω-components, we have that the only only choices for μ^{in} and μ^{out} are M_0 and M_f themselves. Consequently, we have that

$$\mathcal{L}(\mathcal{U}_0, \varphi_0) = \left\{\tau \in T^* \mid M_0 \xrightarrow{\tau} M_f\right\} = \mathcal{L}(N, M_0, M_f) .$$

Our examples shows that it is easy to construct an MGTS that has the same language of the Petri net. This implies that handling languages of arbitrary MGTS is as hard as handling Petri net firing sequences. In the following, we want to find a condition on a MGTS that implies that its languages can be described using linear algebra.

6.28 Definition
Let N be a Petri net and let $(\mathcal{U}, \varphi)$ be a MGTS.

Let $\mathcal{R} = \biguplus_i R(G_i)$ be the disjoint union of all arcs occurring in any G_i. Define $\mathcal{C} = \{0, \ldots, i\} \times (P \sqcup P)$ be the set of components occurring in the collection of the M_i^{in} and M_i^{out}. In the following, we will consider vectors $x \in \mathbb{N}^{\mathcal{R} \cup \mathcal{C}}$ that have one entry $x_{r,i}$ for each arc r of some G_i and entries $x_i^{\text{in}}(p)$ and $x_i^{\text{out}}(pb)$ for each component p of some M_i^{in} resp. M_i^{out}.

Such a vector is associated to a transition sequence

$$\tau = \tau_{(0)} t_1 \tau_{(1)} t_2 \ldots t_n \tau_{(n)} \in \mathcal{L}(\mathcal{U}, \varphi)$$

if there are

- for each i a path $\pi_{(i)}$ in G_i from m_i to m_i such that $\tau_{(i)}$ is the sequence of transitions occurring along $\pi_{(i)}$, such that $x\restriction_{R(G_i)}$ is the occurrence vector $\Psi(\pi_{(i)})$ of $\pi_{(i)}$
- We have

  $$x_1^{\text{in}} \xrightarrow{\tau_{(0)}} x_1^{\text{out}} \xrightarrow{t_1} x_2^{\text{in}} \xrightarrow{\tau_{(1)}} x_2^{\text{out}} \xrightarrow{t_2} \ldots x_{n-1}^{\text{out}} \xrightarrow{t_n} x_n^{\text{in}} \xrightarrow{\tau_{(n)}} x_n^{\text{out}}$$

  and $x_i^{\text{in}} \leqq_\omega M_i^{\text{in}}, x_i^{\text{out}} \leqq_\omega M_i^{\text{out}}$.

It is easy to see that we indeed have at least one vector associated to any $\tau \in \mathcal{L}(\mathcal{U}, \varphi)$.

6.29 Definition
The **characteristic equation** of a MGTS $(\mathcal{U}, \varphi)$ is the following linear system of equa-

tions:
For all $i \in \{0, \ldots, i\}$ and $p \in P$

$$x_i^{\text{in}}(p) = M_i^{\text{in}}(p) \qquad \text{if } M_i^{\text{in}} \neq \omega \tag{1}$$

$$x_i^{\text{out}}(p) = M_i^{\text{out}}(p) \qquad \text{if } M_i^{\text{out}} \neq \omega \tag{2}$$

$$x_{i+1}^{\text{in}} - x_i^{\text{out}} = e(t_{i+1}) \qquad \text{if } i \neq n-1 \tag{3}$$

$$x_i^{\text{in}} - x_i^{\text{out}} = \sum_{r=(q,t,q')\in R(G_i)} x(r) \cdot e(t) \tag{4}$$

and for each $i \in \{0, \ldots, i\}$ and each vertex $m \in V(G_i)$

$$\sum_{\substack{r=(q,t,m) \\ \text{incoming in } m}} x(r) = \sum_{\substack{r=(m,t,q') \\ \text{outgoing from } m}} x(r) \tag{5}$$

.

6.30 Proposition
Any vector $x \in \mathbb{N}^{\mathcal{R}\cup\mathcal{C}}$ associated to some element of $u \in \mathcal{L}(\mathcal{U}, \varphi)$ indeed satisfies the characteristic equation.

Proof sketch:
This is clear by the definition of the characteristic equation.

(1) formalizes $x_i^{\text{in}} \leqq_\omega M_i^{\text{in}}$,

(2) formalizes $x_i^{\text{out}} \leqq_\omega M_i^{\text{out}}$,

(3) formalizes $x_i^{\text{out}} \xrightarrow{t_i}$

(5) formalizes that each $x{\restriction_{R(G_i)}}$ is indeed the occurrence vector of some path $\pi_{(i)}$ in G_i from m to m. Note that requiring that we enter each vertex as often as we leave it is sufficient and necessary.

(4) formalizes that the effect induced e any such path $\pi_{(i)}$ indeed satisfies

$$x_i^{\text{in}} + e = x_i^{\text{out}} .$$

□

6.31 Definition
Let N be a Petri net, $(\mathcal{U}, \varphi)$ be a MGTS and let $Ax = b$ be its characteristic equation. A MGTS $(\mathcal{U}, \varphi)$ is **perfect** iff for all $i \in \{0, \dots, n\}$

$$m_i = C\Big(N, M_i^{\text{in}}, G_i, m_i\Big) = C\Big(N^{\text{rev}}, M_i^{\text{out}}, G_i^{\text{rev}}, m_i\Big)$$

and $Ax = 0$ has a solution x such that

$$\begin{aligned} x\restriction_{\mathcal{R}} &\geq \vec{1} \\ x_i^{\text{in}}(p) &\geq 1 \quad \text{if } M_i^{\text{in}}(p) = \omega \\ x_i^{\text{out}}(p) &\geq 1 \quad \text{if } M_i^{\text{out}}(p) = \omega \end{aligned}$$

Excursion: Some linear algebra

6.32 Theorem
Let $Ax = b$ be a system of equations over $\mathbb{Z}$. It is decidable whether an integer solution x exists, and if it does, we may compute one.

6.33 Lemma
Let $Ax = 0$ be a homogeneous linear system of equations over $\mathbb{Z}$ where we consider solutions $x \in \mathbb{N}^C$.

Either there is strictly positive solution $x \in \mathbb{N}^C$ with $x \geq \vec{1}$, or the set

$$\mathcal{Z} = \Big\{ i \in C \;\Big|\; \forall x \in \mathbb{N}^C \colon Ax = 0 \implies x_i = 0 \Big\}\,,$$

of components that have to be 0 in any solution is non-empty.

Proof:
If $Ax = 0$ has no solution at all, there is nothing to show.

Let us assume that $\mathcal{Z}$ is empty, i.e. there is no component i such that $Ax = 0$ implies $x_i = 0$. In this case, we may pick for each component i a solution $x^{(i)} \in \mathbb{N}^P$ such that $x_i^{(i)} > 0$. Now consider $x = \sum_{i \in C} x^{((i))}$.

We have

$$Ax = \sum_{i \in C} Ax^{(i)} = \sum_{i \in C} 0 = 0$$

and x is strictly positive as

$$x_i = \sum_{j \in C} x_i^{(j)} \geq x_i^{(i)} > 0\,.$$

This proves the desired statement. □

6.34 Theorem
Let $Ax = i$ be a linear system of equations over $\mathbb{Z}$. The set

$$\mathcal{Z} = \left\{ i \in C \;\middle|\; \forall x \in \mathbb{N}^C \colon Ax = 0 \implies x_i = 0 \right\},$$

can be computed. If $\mathcal{Z} \neq \emptyset$, then for any $i \in \mathcal{Z}$, x_i can only take a finite number of possible values in any solution of $Ax = b$. We may compute the set

$$\mathcal{V} = \left\{ x\restriction_{\mathcal{Z}} \;\middle|\; x \in \mathbb{N}^P, Ax = b \right\}$$

of all (combinations of) such values.

We may compute a vector x_0 with $x_0\restriction_{\mathcal{Z}} \geq \vec{1}$ and $Ax_0 = 0$, called **maximal support solution** of $Ax = 0$.

– End of excursion –

Using the linear algebra and Lambert's pumping lemma, which we skip here, one can prove the following theorem.

6.35 Theorem
Let $(\mathcal{U}, \varphi)$ be a perfect MGTS. Then we have that $\mathcal{L}(\mathcal{U}, \varphi)$ is non-empty if and only if its characteristic equation $Ax = b$ has an integer solution and t_{i+1} is enabled in M_i^{out}.

By Theorem 6.32, we obtain that for perfect MGTS, language emptiness is decidable.

6.36 Corollary
For perfect MGTS, it is decidable whether $\mathcal{L}(\mathcal{U}, \varphi)$.

Decomposing MGTS

Assume we could prove the following theorem.

6.37 Theorem
We can compute a finite set of perfect MGTS Γ such that

$$\mathcal{L}(N, M_0, M_f) = \bigcup_{(\mathcal{U},\varphi)\in\Gamma} \mathcal{L}(\mathcal{U}, \varphi)$$

This would yield, the decidability of Petri net reachability, Theorem 6.2.

Proof of Theorem 6.2:
By Theorem 6.37, we can compute a finite of perfect MGTS Γ such that

$$\mathcal{L}(N, M_0, M_f) = \bigcup_{(\mathcal{U},\varphi)\in\Gamma} \mathcal{L}(\mathcal{U}, \varphi)$$

We have

$$\begin{array}{ll} & M_f \text{ is reachable from } M_0 \text{ in } N \\ \text{iff} & \mathcal{L}(N, M_0, M_f) \neq \emptyset \\ \text{iff} & \exists (\mathcal{U}, \varphi) \in \Gamma\colon \mathcal{L}(\mathcal{U}, \varphi)\emptyset\ . \end{array}$$

The latter property is decidable using the fact that Γ is finite and computable and Corollary 6.36. □

It remains to prove 6.37. To this end, we show that we can start with an arbitrary MGTS and decompose it further.

6.38 Definition
A decomposition of $(\mathcal{U}, \varphi)$ be a MGTS into a finite (possibly empty) set Γ means that

(C1) each $(\mathcal{U}', \varphi') \in \Gamma$ is obtained from $(\mathcal{U}, \varphi)$ by replacing each G_i by some MGTS,

(C2) $\mathcal{L}(\mathcal{U}, \varphi) = \bigcup_{(\mathcal{U},\varphi)\in\Gamma} \mathcal{L}(\mathcal{U}, \varphi)$

(C3) for each $(\mathcal{U}', \varphi') \in \Gamma$, we have

$$M^{\text{in}}(\mathcal{U}', \varphi') \leqq_\omega M^{\text{in}}(\mathcal{U}, \varphi) \quad \text{and} \quad M^{\text{out}}(\mathcal{U}', \varphi') \leqq_\omega M^{\text{out}}(\mathcal{U}, \varphi)\ .$$

6.39 Theorem: Decomposition theorem
For any MGTS, we can compute a perfect set of MGTS it decomposes into.

Proof:
Let $(\mathcal{U}, \varphi)$ be the given MGTS and let $Ax = b$ be the characteristic equation.

Decomposing non-perfect MGTS

If it is perfect, there is nothing to do, so assume it is not. We consider each possible reason for $(\mathcal{U}, \varphi)$ not being perfect, and show that each of them leads to a decomposition.

(1) Assume that there is an i and a place p such that $M_i^{\text{in}}(p) = \omega$, but any solution of $Ax = 0$ has $x_i^{\text{in}}(p) = 0$. Using Theorem 6.34, we can compute the finite set of values $\mathcal{V}$ that $x_i^{\text{in}}(p)$ can take in any solution of $Ax = b$. Consider the set Γ of MGTS obtained from $(\mathcal{U}, \varphi)$ that contains for each $v \in \mathcal{V}$ a MGTS $(\mathcal{U}, \varphi_v)$ obtained by setting $M_i^{\text{in}}(p)$ to v. This means Γ contains one MGTS for each possible value.

    We claim that Γ is a decomposition of $(\mathcal{U}, \varphi)$. Conditions C1 and C3, it remains to argue for language equivalence, C2,

$$\mathcal{L}(\mathcal{U}, \varphi) = \bigcup_{v \in \mathcal{V}} \mathcal{L}(\mathcal{U}, \varphi_v)\,.$$

    Since $M_i^{\text{in}}(p) = \omega$, which is no restriction, we obtain that the right-hand side is a subset of the left-hand side. For the other inclusion, take any $\tau \in \mathcal{L}(\mathcal{U}, \varphi)$, and let x be an associated vector. By Proposition 6.30, x satisfies the characteristic equation. This means we have $x_i^{\text{in}}(p) = v \in \mathcal{V}$. It is easy to check that $\tau \in \mathcal{L}(\mathcal{U}, \varphi_v)$ holds.

(2) As (1), but for output places.

(3) Assume that there is an arc r in some $R(G_i)$ such that $Ax = 0$ implies $x(r) = 0$. Using Theorem 6.34, we can compute the finite set of values $\mathcal{V}$ that $x(r)$ can take in any solution of $Ax = b$.

    Consider the third IPG decomposition, Proposition 6.24 for $E = \{e\}$ and $F = \mathcal{V}$. The propositions allows us to compute a set $\mathcal{L} \subseteq T^*$ and for each $s_1 \dots s_n$ a sequence of IPGs

$$(G_0^s, m_0^s), (G_1^s, m_1^s) \dots (G_n^s, m_n^s)$$

    such that

$$\begin{aligned} m_i &= m_0^s = m_n^s \\ \forall j{:}\,\Omega(G_j^s) &= \Omega(G) \\ \forall j{:}\,|R(G_j^s)| &< |R(G_i)| \\ \forall j{:}\,m_j^s &\xrightarrow{s_{j+1}} m_{j+1}^s\,. \end{aligned}$$

We define Γ to be the set of MGTS $\{(\mathcal{U}_s, \varphi_s) \mid s \in \mathcal{L}\}$, where $\mathcal{U}_s$ is obtained by replacing G_i by the graph transition sequence

$$(G_0^s, m_0^s)s_1(G_1^s, m_1^s)s_2 \dots s_n(G_n^s, m_n^s) .$$

It remains to define φ_s. To this end, consider some τ such that

$$\mu^{\text{in}} \xrightarrow{\tau} \mu^{\text{out}}$$

with $\mu^{\text{in}} \leqq_\omega M_i^{\text{in}}$, $\mu^{\text{out}} \leqq_\omega M_i^{\text{out}}$ and write

$$\tau = \tau_{(0)}s_1\tau_{(1)}s_2 \dots s_n\tau_{(n)} .$$

We have

$$\begin{aligned}
&\mu^{\text{in}} \xrightarrow{\tau_{(0)} \dots t_j} \mu_j^{\text{in}} \leqq_\omega m_j^s && \text{for each } j \text{ by Prop. 6.24} \\
\Longrightarrow\ &\mu^{\text{in}} \xrightarrow{\tau_{(0)} \dots t_j\tau_{(j)}} \mu_j^{\text{out}} \leqq_\omega m_j^s && \text{since } G_j^s \text{ is a precovering graph} \\
\Longrightarrow\ &\mu^{\text{in}} \xrightarrow{\tau_{(0)} \dots t_j\tau_{(j)}t_{j+1}} \mu_{j+1}^{\text{in}} \leqq_\omega m_{j+1}^s && \text{since } m_j^s \xrightarrow{t_{j+1}} m_{j+1}^s
\end{aligned}$$

for some sequences of markings $\mu_j^{\text{in}}, \mu_j^{\text{out}}$.

We define the φ_s by setting

$$\begin{aligned}
(C_0^s, m_0^s) &\mapsto (M_i^{\text{in}}, m_0^s) \\
(C_j^s, m_j^s) &\mapsto (m_j^s, m_j^s) && 0 < j < n \\
(C_n^s, m_n^s) &\mapsto (m_n^s, M_i^{\text{out}}) .
\end{aligned}$$

(The markings for the other IPGs from the original MGTS remain unchanged.) This is a valid marking since $M_i^{\text{in}} \leqq_\omega m_i = m_0^s$ and $M_i^{\text{out}} \leqq_\omega m_i = m_n^s$. For all other markings in between, we even have equality by our definition of φ_s.

Checking conditions C1 and C3 is easy. Language equivalence, C2, follows from Proposition 6.24 by choosing a suitable $s \in \mathcal{L}$ as discussed above.

(4) Assume there is an i such that $m_i \neq C\left(N, M_i^{\text{in}}, G_i, m_i\right)$,

We apply the first IPG decomposition, Proposition 6.20, to obtain a set $\mathcal{L} \subseteq S$ and for each $s = s_1 \dots s_n \in \mathcal{L}$ a sequence of IPGs

$$(G_0^s, m_0^s), (G_1^s, m_1^s) \dots (G_n^s, m_n^s)$$

such that

$$\begin{aligned} M_i^{\text{in}} &= m_0^s \\ \forall j{:}\Omega(G_j^s) &\subsetneq \Omega(G_i) \\ \forall j{:}m_j^s &\xrightarrow{s_{j+1}} m_j^s + e(s_j) \leqq_\omega m_{j+1}^s \ . \end{aligned}$$

We define Γ to be the set of MGTS $\{(\mathcal{U}_s, \varphi_s) \mid s \in \mathcal{L}\}$, where $\mathcal{U}_s$ is obtained by replacing G_i by the graph transition sequence

$$(G_0^s, m_0^s)s_1(G_1^s, m_1^s)s_2 \ldots s_n(G_n^s, m_n^s) \ .$$

It remains to define φ_s, we do this by setting

$$\begin{aligned} (G_0^s, m_0^s) &\mapsto (M_i^{\text{in}}, m_0^s) \\ (G_j^s, m_j^s) &\mapsto (m_j^s, m_j^s) && 0 < j < n \\ (G_n^s, m_n^s) &\mapsto (m_n^s, M_i'^{\text{out}}) \ . \end{aligned}$$

where $M_i'^{\text{out}}$ still has to be defined. The markings for the other IPGs from the original MGTS remain unchanged.

For φ_s to be a valid marking and for Condition (C3) to hold, $M_i'^{\text{out}}$ should satisfy

$$M_i'^{\text{out}} \leqq_\omega m_n^s \quad \text{and} \quad M_i'^{\text{out}} \leqq_\omega M_i^{\text{out}} \ .$$

This is possible if and only if m_n^s and M_i^{out} agree on their non-ω components, i.e. if and only if the following property holds:

$$\forall p{:}\ m_n^s(p) \neq \omega \text{ and } M_i^{\text{out}} \neq \omega \implies m_n^s(p) = M_i^{\text{out}}(p) \ .$$

Assume that the property holds. We can now define

$$M_i'^{\text{out}}(p) = \begin{cases} \omega & m_n^s(p) = \omega \text{ and } M_i^{\text{out}}(p) = \omega \ , \\ M_i^{\text{out}}(p) & m_n^s(p) = \omega \text{ and } M_i^{\text{out}}(p) \neq \omega \ , \\ m_n^s(p) & m_n^s(p) \neq \omega \text{ and } M_i^{\text{out}}(p) = \omega \ , \\ m_n^s(p) = M_i^{\text{out}}(p) & \text{else.} \end{cases}$$

In other words set $M_i'^{\text{out}}$ to be the component-wise minimum of M_i^{out} and m_n^s. We obtain that $M_i'^{\text{out}}$ is the largest marking that satisfies $M_i'^{\text{out}} \leqq_\omega m_n^s$ and $M_i'^{\text{out}} \leqq_\omega M_i^{\text{out}}$.

Unfortunately, the desired property, i.e. M_i^{out} and m_n^s agreeing on their non-ω components might be violated for some of the $s \in \mathcal{L}$. The solution is to remove the corresponding MGTS $(\mathcal{U}_s, \varphi_s)$ from Γ, i.e. we want Γ to only contain the MGTS $(\mathcal{U}_s, \varphi_s)$ for which m_n^s satisfies the above property.

Let us argue that this new set Γ is a valid decomposition. Condition (C1) is clearly satisfied, and we have chosen the new Γ such that (C3) also holds by construction.

It remains to check that the language is preserved, Condition (C2), which is non-trivial as we have removed some of the s.

To this end, we consider a firing sequence in $\mathcal{L}(\mathcal{U}, \varphi)$ and show that it is in the language of a $(\mathcal{U}_s, \varphi_s)$ for some s that we have not removed. It is sufficient to consider an infix of the firing sequence corresponding to the G_i that we replace.

Let $\tau \in T^*$ such that

$$\mu^{\text{in}} \xrightarrow{\tau} \mu^{\text{out}}$$

with $\mu^{\text{in}} \leqq_\omega M_i^{\text{in}}$, $\mu^{\text{out}} \leqq_\omega M_i^{\text{out}}$ and write

$$\tau = \tau_{(0)} s_1 \tau_{(1)} s_2 \ldots s_n \tau_{(n)}$$

for some $s \in \mathcal{L}$ with $\tau_{(j)} \in \mathcal{L}\left(G_j^s, m_j^s, m_j^s\right)$.

Since $\mu^{\text{in}} \leqq_\omega M_i^{\text{in}} = m_0^s$, and for all j, $m_j^s + e(s_j) \leqq_\omega m_{j+1}^s$ by Proposition 6.20, we have $\mu^{\text{out}} \leqq_\omega m_n^s$. Since we also have $\mu^{\text{out}} \leqq_\omega M_i^{\text{out}}$ by assumption, we have for each p such that $m_n^s(p) \neq \omega$ and $M_i^{\text{out}}(p) \neq \omega$

$$m_n^s(p) = \mu_i^{\text{out}}(p) = M_i^{\text{out}}(p)\ .$$

Consequently, m_n^s satisfies the property that we have required above and the corresponding $(\mathcal{U}_s, \varphi_s)$ is contained in the new Γ.

(5) If there is an i such that $m_i \neq C\left(N^{\text{rev}}, M_i^{\text{out}}, G_i^{\text{rev}}, m_i\right)$, we proceed as in the previous case using the second IPG decomposition, Proposition 6.23.

The algorithm

We now construct an algorithm that computes the decomposition of the initially given MGTS.

Let $\mathcal{T}$ be the empty tree
Construct a new root node $(\mathcal{U}, \varphi)$
while $\mathcal{T}$ has a leaf $(\mathcal{U}', \varphi')$ that is not perfect **do**

Compute Γ' using one of the cases (1) - (5) // at least one of the cases is applicable
for $(\mathcal{U}_s, \varphi_s) \in \Gamma'$ **do**
Construct a child $(\mathcal{U}_s, \varphi_s)$ of $(\mathcal{U}', \varphi')$
end for
end while
return $\mathcal{T}$

Soundness

Assume the algorithm terminates. In this case, define Γ to be the set of all leaves of the tree. Note that each leaf is perfect; Otherwise, the algorithm would not have terminated. To conclude that Γ is a decomposition of the original MGTS $(\mathcal{U}, \varphi)$, note that each branching in the tree corresponds to a decomposition that preserves the language. Consequently, the union of the languages of all leaves is still the language of the initially given MGTS.

Termination

Assume that the algorithm does not terminate. Because the computation of the Γ' according to (1) - (5) always terminates, this means $\mathcal{T}$ is becoming infinitely large. Because each Γ' is finite, the tree has finite out-degree.

By König's Lemma, an infinite tree with finite out-degree needs to contain an infinite path, say

$$(\mathcal{U}, \varphi) = (\mathcal{U}_0, \varphi_0), (\mathcal{U}_1, \varphi_1), (\mathcal{U}_2, \varphi_2), \ldots .$$

Consider two successive entries $(\mathcal{U}_i, \varphi_i), (\mathcal{U}_{i+1}, \varphi_{i+1})$ in the chain and note that since the latter is a child of the first in the tree, it is obtained by applying one of the cases (1) - (5).

In the cases (1) and (2), the number of ω-components int the input or output marking strictly decreases. In the remaining cases, we replace one IPG in $(\mathcal{U}_i, \varphi_i)$ by potentially multiple IPGs that all have either strictly less ω-components (case (4) and (5)) or strictly less arcs. This allows us to conclude that the chain cannot be infinite.

□

Recall Theorem 6.37, which directly implies the decidability of Petri net reachability 6.2.

6.40 Theorem
We can compute finite set of perfect MGTS Γ such that

$$\mathcal{L}(N, M_0, M_f) = \bigcup_{(\mathcal{U},\varphi)\in\Gamma} \mathcal{L}(\mathcal{U}, \varphi)$$

Proof:
Consider the MGTS $(\mathcal{U}_0, \varphi_0)$ from Examples 6.27. As noted there, we have

$$\mathcal{L}(N, M_0, M_f) = \mathcal{L}(\mathcal{U}_0, \varphi_0) \ .$$

We may now apply the decomposition theorem, Theorem 6.39, to compute a finite set Γ such that

$$\mathcal{L}(N, M_0, M_f) = \mathcal{L}(\mathcal{U}_0, \varphi_0) = \bigcup_{(\mathcal{U}, \varphi) \in \Gamma} \mathcal{L}(\mathcal{U}, \varphi)$$

as desired. □

Exercises

6.41 Exercise: Counter programs

You may use additional counter variables to solve these problems. In each part of this exercise, you may use the previous parts as subroutines.

Let n be some fixed number.

a) Present a counter program $\mathtt{Set}_n(x_j)$ that sets the value of counter variable x_j to n.

b) Present a counter program $\mathtt{Double}(x_j)$ that doubles the current value of counter variable x_j.

c) Present a counter program $\mathtt{Power}_n(x_j)$ that sets the value of counter variable x_j to 2^n.

d) Present a counter program $\mathtt{Square}(x_j)$ that squares the value of counter variable x_j, i.e. the new value is v^2, where v is the old value.

In each part of this exercise, argue briefly that your program is correct.

6.42 Exercise: Using a unary encoding

Assume that we measure the size of Petri nets and markings by taking the unary encoding of the numbers, i.e. we redefine $|M| = \sum_{p \in P}(1 + M(p))$ and $|N| = \sum_{t \in T, p \in P}(1 + in(o, t) + out(t, p))$.

a) Does the coverability problem get any easier using this assumption?

   Hint: Inspect the proof of Lipton's result.

b) Discuss whether Rackoff's bound can be improved, proving

$$f(i+1) \leqslant (n \cdot f(i))^{i+1} + f(i) .$$

Part II.
Weak memory models

7. Total store ordering

Two of the main components of a modern computer are the CPU, which can quickly perform arithmetic computations but has only a limited amount of local registers as storage, and the memory (including the CPU cache, the main memory, and hard disk drives) from resp. to which the register content can be loaded resp. stored. In parallel programming, one usually uses the shared memory as a means of communication between threads, i.e. via a lock, a memory location that is set to 1 by a thread to signal that it needs exclusive access to a certain part of the memory for the time being. The correctness of such mechanisms relies on the assumption of having an underlying **strong memory model**, meaning that any write done by one thread becomes immediately visible to the other threads.

Such a model would require the CPU to wait in front of each memory access until it can be sure that all operations by other threads have become visible. As the clock rate of a modern CPU is roughly ten times as high as the clock rate of the main memory, this would make parallel programming unusably slow. To solve the problem, the designers of the CPU architectures have devised several tricks.

Here, we want to consider the **x86 architecture** common in processors for desktop computers and servers. In this architecture, any store made to the main memory is first put into a buffer. At some points in time, the content of the buffer is batch-processed into the main memory. This uses the fact that writing several stores to the main memory at once is faster than doing it successively for each store. To make synchronization mechanisms like locks work, x86 assembly provides a special **memory fence** command that ensures that the buffer has been emptied and all writes done by the thread have become visible to other threads.

When programming in a high level language like C++, programmers do not have to worry about this, but the people writing the compiler that translate the code into assembly as well as people directly writing assembly code need to make sure that they use the synchronization mechanisms like memory fences in the appropriate places.

It is a challenge to verify parallel programs under the assumption that they are not executed under a strong memory model. Here, we want to abstract away implementation details like the size of the buffer, the frequency with which it is emptied, and so on. Although this data might be available, it may change between different CPUs with the x86 architecture, and the correctness of a program should not rely on them. Instead, we define a **weak memory model** that describes the behavior of the memory in the x86 architecture in principle and is valid for all CPUs with this architecture.

The memory model used for x86 is called **total store ordering (TSO)**. The name means that there is a total order on the points in time at which the stores to the memory become visible to all threads. (Other memory models may allow that a store operation first become visible to some threads.)

Before considering verification problems for x86 programs executed under TSO, we first define a simplified version of x86 assembly and its semantics under TSO.

7.1 Definition: Syntax of parallel programs
The parallel programs we consider are defined by the following grammar.

⟨*prog*⟩	::=	program ⟨*name*⟩ ⟨*thread*⟩*	// Name and finite list of threads
⟨*thread*⟩	::=	thread⟨*threadid*⟩	// Identifier
		regs ⟨*reg*⟩*	// List of local registers used by the thread
		init ⟨*label*⟩	// Label of the initial instruction
		begin ⟨*linst*⟩* end	// List of label instructions
		end	
⟨*linst*⟩	::=	⟨*label*⟩ : ⟨*inst*⟩; goto⟨*label*⟩;	
⟨*inst*⟩	::=	⟨*reg*⟩ ← mem[⟨*reg*⟩]	// Load
	|	mem[⟨*reg*⟩] ← ⟨*reg*⟩	// Store
	|	mfence	// Memory fence
	|	⟨*reg*⟩ ← ⟨*expr*⟩	// Local assignment
	|	assert ⟨*expr*⟩	// Assertion

Here, we assume the following:

- The threads identifiers ⟨*threadid*⟩ are distinct numbers,
- the registers ⟨*reg*⟩ are chosen from a finite set of names (later, we will use $x, y, r, \ldots$), and no register is shared between threads,
- the labels ⟨*label*⟩ are strings, and each command has a distinct label (later, we will use $\ell_0, \ell_1, \ldots$),
- the program comes with a **finite data domain** *DOM* whose elements can be used as register content as well as as memory addresses,
- *DOM* contains the value 0,
- expressions ⟨*expr*⟩ are build from register names and a finite set of functions from a **function domain** *DOM* of (multi-parameter) functions defined on *DOM*, and

- we implicitly require that each thread only accesses its own registers and only jumps to its own labeled instructions.

7.2 Remark
Our version of assembly lacks features usually present, e.g. conditional jumps with which conditionals (if-then-else) and loops can be realized. It would be easy to add such features to the language without adapting the theory that we will develop in the following. The only reason why we choose not do so is to keep the language simple and focus on the interaction with the memory.

It remains to define the semantics of a parallel program executed under TSO. Each thread has a **store buffer**. Stores made by a thread are buffered locally and later propagated to the main memory in a FIFO manner. As long as a store is in a buffer, it is not visible to other threads. The thread that issued the store can do a **early read** from its own buffer, i.e. instead of loading the value from the main memory, it loads the last value stored to the address by itself.

Before we formally define the transition relation, we consider an example.

7.3 Example: Dekker's mutex
Consider the following parallel program. Note that it uses a simplified notation (i.e. the threads are separated by two lines) and does not follow the grammar from the definition, but it can easily be transformed.

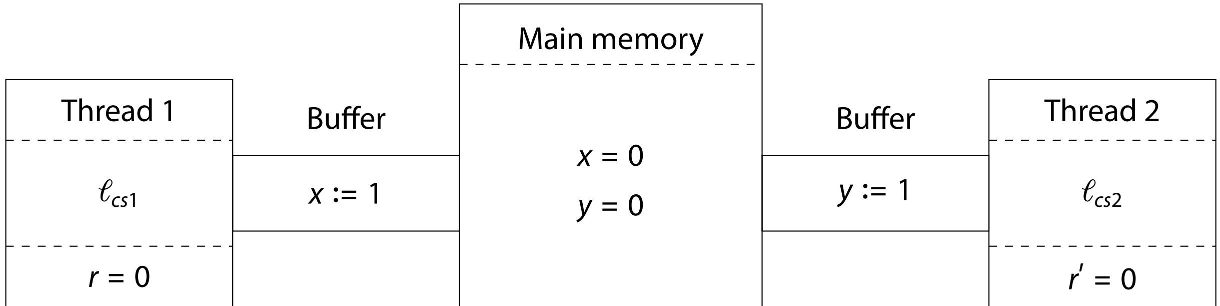
$$
\begin{array}{ll||ll}
\ell_0: & \text{mem}[x] \leftarrow 1; \text{goto}\ell_1; & \ell'_0: & \text{mem}[y] \leftarrow 1; \text{goto}\ell_1; \\
\ell_1: & r \leftarrow \text{mem}[y]; \text{goto}\ell_2; & \ell'_1: & r' \leftarrow \text{mem}[x]; \text{goto}\ell_2; \\
\ell_2: & \text{assert } r == 0; \text{goto}\ell_{cs1}; & \ell'_2: & \text{assert } r' == 0; \text{goto}\ell_{cs2}; \\
\ell_{cs1}: & // \text{ critical section} & \ell_{cs2}: & // \text{ critical section}
\end{array}
$$

Executed under a strong memory model, mutual exclusion holds, i.e. it is not possible to reach a configuration in which both threads are in the critical section. Either one thread executes the first two lines of its code before the other even starts running. In this case, the assert in this thread is successful and it can enter the critical section, while the other thread blocks as soon as it reaches the assert. If both threads execute the store in their first line before any of them loads the value written by the other, both threads block.

Executed under TSO, the program may exhibit unwanted behavior: Both threads may issue the store, which gets put in the local buffer. Both threads then load value 0 into their register, since this value is taken from the main memory that does not yet contain

the issued store. The assert is successful in both threads and they both enter the critical section.

This situation is depicted in the following graphic.



Let us now formally define the semantics of parallel programs under TSO. We first define the configurations, and then the transition relation between configurations.

7.4 Definition: Semantics of parallel programs under TSO: Configurations
Consider a program P with threads $t_1, \ldots, t_n$. Assume that i is the thread identifier of thread t_i, $\ell_{0,i}$ is its initial label and it declares the set of registers R_i.

Let $TID = \{1, \ldots, n\}$ denote the set of thread identifiers, LAB the set of all labels used by all threads and $VAR = DOM \cup \bigcup_{i=1,\ldots,n} R_i$ denote the set of all locations (addresses and registers).

A **configuration** of P is an element from the set

$$CF = (TID \to LAB) \times (VAR \to DOM) \times (TID \to (DOM \times DOM)^*)\ ,$$

i.e. a tuple of the shape

$$cf = (pc, val, buf)$$

where

- $pc\colon TID \to LAB$ is the **program counter**, assigning to each thread $i \in TID$ the label $pc(i) \in LAB$ of the instruction in its code that should be executed next,
- $val\colon (VAR \to DOM)$ is the **valuation**, assigning to registers r from some R_i their value $val(r)$ and to addresses a their value $val(a)$ in the main memory, and
- buf is the collection of **local buffers**, i.e. for each thread $i \in TID$, $buf(i)$ is a sequence of tuples $(a, v) \in DOM \times DOM$ that is currently buffered. We write such tuples as $a := v$, meaning that value v should be stored at address a. The left-hand

side of the sequence is the CPU side end of the buffer (i.e. a store $a := v$ that is issued is prepended), the right-hand side is the memory side (i.e. when a store gets propagated to the main memory, the rightmost element of the sequence is deleted).

The initial configuration is

$$cf_0 = (pc_0, val_0, buf_0)$$

where $pc_0(i) = \ell_{0,i}$ is the initial label for each threads i, the buffer is empty for all threads, $buf_0(i) = \varepsilon$, and all values are initialized to zero, $val_0(x) = 0$ for all $x \in VAR$.

7.5 Definition: Semantics of parallel programs under TSO: Transition relation
The transition relation $\rightarrow_{\mathrm{TSO}}$ is defined in an operational way. We provide calculus rules describing how the transition between configurations are induced by each part of the syntax.

Assume we are in configuration $cf = (pc, val, buf)$ with $pc(i) = \ell$ for some thread i, and we want to execute the labeled instruction $\ell : \langle inst \rangle ; \mathrm{goto} \ell'$. We define $pc' = pc[i := \ell']$ as the new program counter after executing this instruction.

The transition relation $\rightarrow_{\mathrm{TSO}}$ is the smallest relation $\rightarrow_{\mathrm{TSO}} \subseteq CF \times CF$ satisfying the following rules.

$$(\mathrm{EARLY})\ \frac{\text{<inst>} = r \leftarrow \mathrm{mem}[r'], a = val(r'), buf(i)\restriction_a = (a := v).\beta}{(pc, val, buf) \rightarrow_{\mathrm{TSO}} (pc', val[r := v], buf)}$$

The conditions on the top of the line are the **premise** of the rule, they need to be satisfied for the rule to be applicable. Here, $buf(i)\restriction_a$ is the restriction of the buffer $buf(i)$ to stores to address a. In other words, we require that $(a := v)$ is the most recent store issued by thread i to address a that has not yet been propagated to the main memory, where a is the value in register r'. In this case, we can perform an **early read** from the buffer instead of loading the value from the main memory.

$$(\mathrm{LOAD})\ \frac{\text{<inst>} = r \leftarrow \mathrm{mem}[r'], a = val(r'), buf(i)\restriction_a = \varepsilon, v = val(a)}{(pc, val, buf) \rightarrow_{\mathrm{TSO}} (pc', val[r := v], buf)}$$

If the buffer contains no store to address a, its value is loaded from the main memory.

$$(\mathrm{STORE})\ \frac{\text{<inst>} = \mathrm{mem}[r] \leftarrow r', a = val(r), v = val(r')}{(pc, val, buf) \rightarrow_{\mathrm{TSO}} (pc', val, buf[i := (a := v).buf(i)])}$$

Stores do not immediately land in the main memory, but are instead prepended to the buffer of the thread that issued them.

$$\text{(UPDATE)}\ \frac{buf(i') = \beta.(a := v)}{(pc, val, buf) \rightarrow_{\text{TSO}} (pc, val[a := v], buf[i' := \beta])}$$

At a later point in time, the earliest store in some buffer that has not yet been propagated to the main memory can be used to update *val*. Here, i' is an arbitrary thread with non-empty buffer. Note that this rule does not update the program counter as we did not execute any instruction.

$$\text{(MFENCE)}\ \frac{\text{<inst>} = \text{mfence}, buf(i) = \varepsilon}{(pc, val, buf) \rightarrow_{\text{TSO}} (pc', val, buf)}$$

An mfence command blocks the thread until its buffer content has been propagated to the main memory (via the update rule). It does not change the buffer or the valuation.

$$\text{(ASSERT)}\ \frac{\text{<inst>} = \text{assert}\ e, [\![e]\!] \neq 0}{(pc, val, buf) \rightarrow_{\text{TSO}} (pc', val, buf)}$$

An assertion can only be taken if the expression that is asserted is non-zero. Here, $[\![e]\!]$ should be the valuation of the expression *e* that is obtained by replacing register names *r* by their current value *val*(*r*), and names of functions by the corresponding functions from *FUN*.

$$\text{(ASSIGN)}\ \frac{\text{<inst>} = r \leftarrow e, [\![e]\!] = v}{(pc, val, buf) \rightarrow_{\text{TSO}} (pc', val[r := v], buf)}$$

Similarly, a local assignment changes the register content.

Exercises

7.6 Exercise: Sequential consistency

In the memory model **SC (sequential consistency)**, we assume that access to the main memory is atomic. More formally, the transition relation $\rightarrow_{\text{SC}}$ is defined similar to $\rightarrow_{\text{TSO}}$, but the rule (STORE) is replaced by the rule (SCSTORE).

$$\text{(SCSTORE)}\ \frac{\text{<inst>} = \text{mem}[r] \leftarrow r', a = val(r), v = val(r')}{(pc, val, buf) \rightarrow_{\text{SC}} (pc', val[a := v], buf)}$$

Note that the buffer will never be used, i.e. early reads and updates from the buffer never occur.

a) Explain the following statement and argue that it is true: There is a correspondence between all executions of a multi-threaded program under SC and the single execution of all single-threaded programs obtained by shuffling the source code of the threads.

b) Let *P* be a program. We define *fency*(*P*) as the program that we obtain from *P* by inserting an mfence instruction directly after every store operation (i.e. $\text{mem}[r] \leftarrow r'$).

   Argue whether the following statement is correct: The program *P* executed under SC has the same behavior as *fency*(*P*) does under TSO.

   Here, you may use control-state reachability (see below) as a suitable definition for "having the same behavior".

7.7 Exercise: SC reachability is in PSPACE

The (control-state) reachability problem for SC is defined as follows.

SC-Reachability	
Decide:	Program *P* over *DOM*, program counter *pc*
Decide:	Is there a computation $cf_0 \rightarrow^*_{SC} (pc, buf, val)$ for some *buf*, *val*?

a) Reduce SC-Reachability to Petri net coverability. Explain which places are needed by the net, and how each instruction in the program can be simulated by Petri net transitions.

b) Conclude that SC-Reachability can be solved in PSPACE. Here, you may assume that the size of DOM is encoded in unary.

8. TSO reachability

In this section, we will consider the TSO reachability problem.

8.1 Definition

TSO reachability

Decide: Program P, program counter pc

Decide: $pc \in \text{Reach}_{\text{TSO}}(P)$?,
i.e. is there a computation $cf_0 \to^*_{\text{TSO}} (pc, val, buf)$ for some val, buf?

We will prove that this problem is decidable. To be precise, we show how to construct a lossy channel system L_P that simulates P. In particular, for each program counter pc, we have a set of corresponding state in L_P such that pc is reachable by P if and only if one of the corresponding states is reachable in L_P. Since reachability in lossy channel systems can be decided using Abdulla's backwards search, TSO reachability is decidable.

We will construct lossy channel systems $L_P^0, L_P^1, L_P^2, L_P^3, L_P$ such that each of them more closely models TSO resp. fixes problems in earlier versions.

Modeling P as LCS L_P^0:

The fundamental idea behind modeling TSO programs as lossy channel systems is that shared memory communication is a lot like message passing in lossy channel system: A store might be overwritten before it is seen by another thread. Therefore, we can understand the TSO buffers as lossy channels.

Consequently, we may construct a lossy channel system L_P^0 whose control states are

$$(TID \to LAB) \times VAR \times DOM ,$$

meaning a control state is of the form (pc, val), storing for each thread i the next instruction $pc(i)$, the content of the local registers $val(r_i)$ and for each address a its value $val(a)$ in the main memory.

Furthermore, we have one channel per thread, each channel storing a sequence of symbols from $DOM \times DOM$, i.e. buffered stores of the shape $a := v$.

The transition relation between the control states is induced by the transition relation $\to_{\text{TSO}}$ between TSO configurations. (We will later provide a formal definition.)

Towards L_P^1:

The underlying well-quasi order for LCSs is Higman's subword ordering. This is not a simulation relation for TSO. Indeed, consider the following program.

$$
\begin{array}{ll|l}
\ell_0: & r \leftarrow \text{mem}[x]; \text{goto} \ell_1; & \ell_0': \quad \text{mem}[y] \leftarrow 1; \text{goto} \ell_1'; \\
\ell_1: & \text{assert } r == 1; \text{goto} \ell_2; & \ell_1': \quad \text{mem}[x] \leftarrow 1; \text{goto} \ell_2'; \\
\ell_2: & r \leftarrow \text{mem}[y]; \text{goto} \ell_3; & \ell_2': \quad \ldots \\
\ell_3: & \text{assert } r == 0; \text{goto} \ell_4; & \\
\ell_4: & \ldots &
\end{array}
$$

Consider the configuration

$$cf = \big((\ell_0, \ell_2', x = y = 0), (\varepsilon, x := 1 \,.\, y := 1)\big)$$

of L_P^0 directly corresponding to a TSO configuration. (For simplicity, we have not shown the content of the registers in the configuration here.) Compare it to the configuration

$$cf' = \big((\ell_0, \ell_2', x = y = 0), (\varepsilon, x := 1)\big)\,.$$

By the ordering of configurations for LCS that is induced by Higman's subword ordering on the channels, we have $cf' \geqslant cf$. Since it should be a simulation ordering, this means that for every state reachable from cf', there should be a larger state reachable from cf. Because the order requires equality of the control states, this in particular means that if a certain program counter is reachable from cf', it also has to be reachable from cf.

Now note that under TSO, we can reach program counter ℓ_4, ℓ_2' from cf' by letting the store $x := 1$ land in main memory and then executing the instructions in the left thread. This is not possible under TSO from cf: The store $x := 1$ needs to land so that ℓ_2 can be reached in the left thread. Since the buffer is FIFO, this means that the store $y := 1$ has also landed, this means that the assert in ℓ_3 will block and ℓ_4 cannot be reached.

Problem: Lossiness gives inconsistent memory configurations.

Fix: We fix this problem by modifying the LCS to obtain L_P^1. In L_P^1, the issuing of a store sends a whole **memory snapshot** to the channel. The snapshot contains the values for all memory addresses as currently seen by the thread.

For example, the configurations of L_P^1 corresponding to cf and cf' are

$$cf_1 = \left((\ell_0, \ell_2', x = y = 0), \left(\varepsilon, \begin{pmatrix} x & := 1 \\ y & := 1 \end{pmatrix} \cdot \begin{pmatrix} x & := 0 \\ y & := 1 \end{pmatrix}\right)\right)$$
$$cf_1' = \left((\ell_0, \ell_2', x = y = 0), \left(\varepsilon, \begin{pmatrix} x & := 1 \\ y & := 0 \end{pmatrix}\right)\right).$$

Note that they are incomparable.

The above problem has vanished.

Towards L_P^2:

Still, some behavior under L_P^1 is not possible under TSO. Consider the following program.

$$\begin{array}{ll||ll} \ell_0: & \text{mem}[y] \leftarrow 0; \text{goto} \ell_1; & \ell_0': & \text{mem}[x] \leftarrow 1; \text{goto} \ell_1'; \\ \ell_1: & \ldots & \ell_1': & r \leftarrow \text{mem}[x]; \text{goto} \ell_2'; \\ & & \ell_2': & \text{assert } r == 0 \text{goto} \ell_3'; \\ & & \ell_3': & \ldots \end{array}$$

Note that (ℓ_1, ℓ_3') is not TSO-reachable: TSO can only load from 1 from x (either via an early read or from main memory), since the store in ℓ_0' needs to have been performed. A configuration with program counter (ℓ_1, ℓ_3') is reachable in L_P^1, namely by the following sequence of transitions,

$$((\ell_0, \ell_0', x = y = 0), (\varepsilon, \varepsilon))$$

$$\rightarrow_{\text{TSO}}^2 \quad \left((\ell_1, \ell_1', x = y = 0), \left(\begin{pmatrix} x & := 0 \\ y & := 0 \end{pmatrix}, \begin{pmatrix} x & := 1 \\ y & := 0 \end{pmatrix}\right)\right) \quad \text{// Buffer both stores}$$

$$\rightarrow_{\text{TSO}} \quad \left((\ell_1, \ell_1', x = 1, y = 0), \left(\begin{pmatrix} x & := 0 \\ y & := 0 \end{pmatrix}, \varepsilon\right)\right) \quad \text{// Update main memory}$$

$$\rightarrow_{\text{TSO}} \quad ((\ell_1, \ell_1', x = y = 0), (\varepsilon, \varepsilon)) \quad \text{// Update main memory}$$

$$\rightarrow_{\text{TSO}}^2 \quad ((\ell_1, \ell_3', x = y = 0), (\varepsilon, \varepsilon)) \quad \text{// Execute load and assert}$$

Problem: Threads do not synchronize on memory updates and may use values that are no longer in memory.

Fix: Instead of having one buffer per thread, we let all threads share the same buffer.

In our example, we could e.g. have the configuration

$$\left((\ell_1, \ell_1', x = y = 0), \begin{pmatrix} x & := 1 \\ y & := 0 \end{pmatrix} \begin{pmatrix} x & := 1 \\ y & := 0 \end{pmatrix} \right).$$

Towards L_P^3:

Now we get the opposite problem: Some TSO behavior is not possible in L_P^2. Consider the following example program.

Thread 1		Thread 2		Thread 3		Thread 4	
$\text{mem}[x] \leftarrow 1;$	(1) (2)	$r_2 \leftarrow \text{mem}[y];$	(12)	$\text{mem}[y] \leftarrow 1;$	(3) (14)	$r_4 \leftarrow \text{mem}[x];$	(8)
$\text{mem}[x] \leftarrow 2;$	(6) (7)	assert $r_2 == 2;$	(13)	$r_3 \leftarrow \text{mem}[x];$	(4)	assert $r_4 == 2;$	(9)
$\vdots$		$r_2 \leftarrow \text{mem}[y];$	(15)	assert $r_3 == 1;$	(5)	$\text{mem}[y] \leftarrow 2;$	(10) (11)
		assert $r_2 == 1;$	(16)	$\vdots$		$\vdots$	
		$\vdots$					

Here, we have omitted the labels and the gotos to save space. Each instruction jumps to the next instruction in the same thread. The numbers after each instruction denote their order in a certain execution, see below.

Under TSO, it is possible to execute the final instruction in each thread, namely by the execution described as follows:

(1) First thread issues store $x := 1$.

(2) This store lands in main memory.

(3) Third thread issues store $y := 1$.

(4) Third thread loads $x = 1$ from the main memory.

(5) Third thread takes the assert.

(6) First thread issues store $x := 2$.

(7) This store lands in main memory.

(8) Fourth thread loads $x = 2$ from the main memory.

(9) Fourth thread takes the assert.

(10) Fourth thread issues store $y := 2$.

(11) This store lands in main memory.

(12) Second threads loads $y = 2$ from the main memory.

(13) Fourth thread takes the first assert.

(14) Store $y := 1$ issued by the third thread in (3) lands.

(15) Second thread loads $y = 1$ from the main memory.

(16) Second thread takes the second assert.

In L_P^2, this is not possible because operations in the buffer will be propagated to the memory in the order in which they entered the buffer. This means that the second thread will not be able to load $y = 2$ before loading $y = 1$.

Problem: In L_P^2, memory updates are forced to occur in the same order as the corresponding stores. In TSO, memory updates can be performed in opposite order if the stores stem from different threads.

Fix: We add to each thread a pointer to a position inside the buffer. From the perspective of some thread t whose pointer is pointing to some entry m of the buffer, the buffer looks as follows:

$$buf = \underbrace{buf'}_{\text{past memory states}} . \underbrace{m}_{\text{current memory state}} . \underbrace{buf''}_{\text{future memory states}} .$$

Updates of the main memory are simulates by moving the pointer to the left.

A possible channel content in L_P^3 might look as follows:

$$\underset{\substack{\uparrow\\ t_2}}{\begin{pmatrix} x & := 1 \\ y & := 1 \end{pmatrix}} \begin{pmatrix} x & := 2 \\ y & := 2 \end{pmatrix} \underset{\substack{\uparrow\\ t_4}}{\begin{pmatrix} x & := 2 \\ y & := 0 \end{pmatrix}} \underset{\substack{\uparrow\\ t_3}}{\begin{pmatrix} x & := 1 \\ y & := 0 \end{pmatrix}} \underset{\substack{\uparrow\\ t_1}}{\begin{pmatrix} x & := 0 \\ y & := 0 \end{pmatrix}}$$

We give a more detailed explanation of this construction later.

8.2 Remark

It is problematic that a the LCS channels are lossy, since the current memory state of a thread could be forgotten. To disallow this, we consider **lossy channel systems with strong symbols.** In this symbols, the symbols occurring in the channel are from a union of sets $M \uplus S$: Symbols from M can be lost as in normal LCS, symbols from S are strong symbols that cannot be lost.

For the reachability problem to be decidable, we need that there is some bound $k \in \mathbb{N}$ such that in each reachable state, each channel contains at most k strong symbols. Lossy channel with a bounded number of strong symbols can be encoded into LCS. We refer the reader to Exercise 8.6 for the details.

In our case, the number of strong symbols that occur is the number of threads and therefore bounded.

Towards L_P

Problem: In L_P^3, early reads are not modeled.

Fix: Remember the last store to an address.

Similar to remembering the current memory snapshot of each state, this is done by introducing additional strong symbols. As we have at most one last store per combination of thread on address, the number of occurrences of such symbols is bounded.

Formal construction of L_P

Given a program P, we define the **LCS with strong symbols L_P**

$$L_P = (Q, q_0, C, M, S, \rightarrow)$$

where

- $Q = TID \rightarrow LAB \times VAR \rightarrow DOM$ are the control states consisting of pc and val
- $C = \{buf\}$ is the single channel,
- $M = DOM \rightarrow DOM$ are the normal messages, i.e. memory snapshots, and
- $S = DOM \rightarrow DOM \times (TID \times DOM \cup \{\varepsilon\} \times \mathcal{P}(TID)) \setminus (DOM \rightarrow DOM \times \{\varepsilon\} \rightarrow \{\varnothing\})$ are the strong symbols.

A strong symbol is of the shape $(mem, lw, threads)$ where $mem \in DOM \rightarrow DOM$ is a memory snapshot. The last write lw is either $(i, a) \in TID \times DOM$ if the snapshot contains the last write to an address a by a thread a, or ε if this snapshot does not contain the last write of any thread. The set $threads \subseteq TID$ is the set of threads pointing to this snapshot, i.e. the set of threads that have this snapshot as their current memory state. We disallow memory snapshots in S that do neither contain the last write by any thread, nor have any thread pointing to them. Such snapshots can be represented by the normal symbols in M.

In any reachable state, the number of strong symbols will be bounded by $|TID| \cdot |DOM| \cdot |TID|$, since for each address $a \in DOM$, there are at most $|TID|$ many last writes to it (one per thread), and for each thread in TID, we need to store one pointer.

It remains to define the transition relation $\rightarrow$. Assume we are in control state $(pc, val) \in Q$ with $pci = \ell$. We define the transitions depending on the instruction labeled by ℓ.

8.3 Remark
In the following, we will describe transitions that actually need to be realized using a sequence of transitions each. This can be done by adding helper control states.

We in particular have several transitions that check whether the buffer contains a certain entry. This can be done by **rotating** through the buffer: We add a special marker at the end of the buffer and then proceed to move elements from the front to the back of the buffer. This allows us to touch each entry of the buffer. As soon as we see our marker again, we have rotated once through the buffer.

Any entry that is lost during the rotation could have also been lost at some other point in the computation.

We refer the reader to Exercise 8.6 for the details.

- Store ℓ: mem$[r] \leftarrow r'$; gotoℓ':
  For $val(r) = a$ and $val(r') = v$, we have an LCS transition from state (pc, val) proceeding as follows:
  1. Check whether the buffer contains a strong symbol of the shape $(mem, (t, a), threads)$. If yes, replace it by $(mem, \varepsilon, threads)$. (This is because after this instruction, we will have a new last store to a by thread t).
  2. Enqueue $(val\restriction_{DOM}[a := v], (a, t), \varnothing)$ into the buffer. (We restrict val to DOM and do not store the content of the registers in the memory snapshot.)
  3. Go to control state $(pc[t := \ell'], val[a := v])$.
- Load ℓ: $r \leftarrow$ mem$[r']$; gotoℓ':
  For each $val(r') = a$, there are two transitions. For early reads:
  1. Assert that the buffer contains some entry $(mem, (t, a), threads)$.
  2. Go to control state $(pc[t := \ell'], val[r := mem(a)])$.

  For loads from the main memory:

1. Assert that the buffer contains no entry $(mem, (t, a), threads)$, i.e. we cannot perform an early read.

2. Find the entry of the buffer $(mem', lw, threads')$ with $t \in threads'$, i.e. find the current memory state of t.

3. Go to control state $(pc[t := \ell'], val[r := mem'(a)])$.

- Memory fence:

  1. Assert that the head of the buffer is of the shape $(mem, lw, threads)$ with $t \in threads$.

  2. Go to control state $(pc[t := \ell'], val)$.

- Update:

  1. Assert that the buffer is of the shape $buf = w_1.m_1.m_2.w_2$ where

$$m_1 = (mem_1, lw_1, threads_1)$$
$$m_2 = (mem_1, lw_2, threads_2) \quad \text{with } t \in threads_2 \ ,$$

  i.e. m_2 is the current memory state of thread t.

  2. Replace m_1 and m_2 by m_1' and m_2, defined by

$$m_1' = (mem_1, lw_1 \setminus \{(t, *)\}, threads_1 \cup \{t\})$$
$$m_2' = (mem_2, lw_2, threads_2 \setminus \{t\}) \ ,$$

  i.e. m_1' is now the new memory state of thread t. The buffer has now the shape $w_1.m_1'.m_2'.w_2$.

  3. Go to the control state (pc, val').

- The rules for assertions and local assignments are straightforward and do not involve the buffer. In both cases, we use the register valuations (stored in val in the control state) to compute the value $[\![e]\!]$ of the expression e. Because there are only finitely many possible values, this can be encoded in the control states.

  An assert e blocks if $[\![e]\!]$ is zero, otherwise, we go to control state $(pc[t := \ell'], val)$.

  In the case of an assignment $r \leftarrow e$, we go to the control state $(pc[t := \ell'], val[r := [\![e]\!]])$.

8.4 Theorem: Atig, Bouajjani, Burckhardt, Musuvathi [Ati+10; Ati+12]
For a program P, one can construct a lossy channel system (with strong symbols) L_P such that a control state pc is reachable by a TSO execution of P if and only if pc is reachable in L_P.

Now recall that reachable in lossy channel systems is decidable using Abdulla's backwards search.

8.5 Corollary
Control-state reachability under TSO is decidable.

We will not give a formal proof of Theorem 8.4, but we will present some argumentation explaining why the construction of L_P is correct.

Shuffling regular languages: The construction makes use of the following automata-theoretic trick: Assume you want to shuffle two languages $\mathcal{L}(A_1), \mathcal{L}(A_2)$, where A_1 and A_2 are automata over disjoint alphabets $\Sigma_1, \Sigma_2, \Sigma_1 \cap \Sigma_2 = \emptyset$. Recall that the shuffle is the set of all possible interleavings obtained from a word from each language,

$$\mathcal{L}(A_1) \sqcup\!\sqcup \mathcal{L}(A_2) = \left\{ w \in (\Sigma_1 \uplus \Sigma_2)^* \mid \mathrm{proj}_{\Sigma_1}(w) \in \mathcal{L}(A_1), \mathrm{proj}_{\Sigma_2}(w) \in \mathcal{L}(A_2) \right\} .$$

(Note that this is not the general definition as it relies on the fact that Σ_1 and Σ_2 are disjoint.)

To construct an automaton accepting the shuffle we first modify the automata A_1 and A_2. Let A_1' be the automaton over $\Sigma_1 \uplus \Sigma_2$ obtained from A_1 by adding for each control state q and each letter a from Σ_2 a self-looping transition $q \xrightarrow{a} q$. Similarly, let A_2' be obtained from A_1 by adding a Σ_1-labeled self-loop to each control state of A_2.

We have

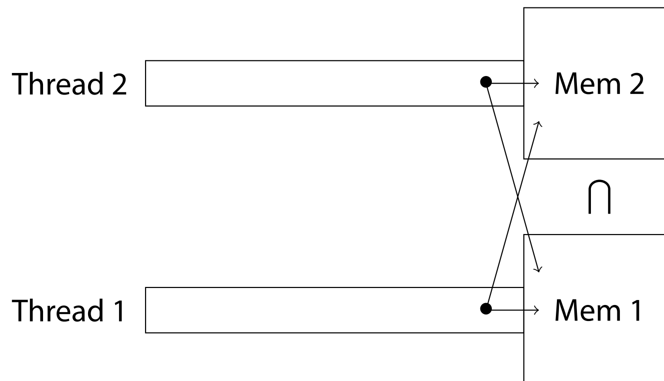
$$\mathcal{L}(A_1') \cap \mathcal{L}(A_2)' = \mathcal{L}(A_1) \sqcup\!\sqcup \mathcal{L}(A_2)$$

so the product automaton $A_1 \times A_2$ accepts the shuffle.

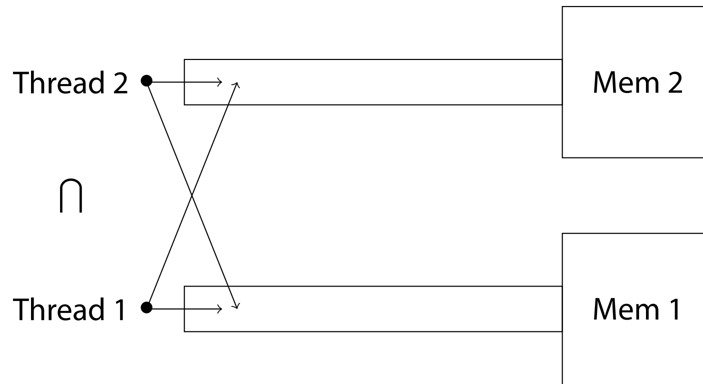
Applying the trick to TSO: Consider two threads executing under TSO.

This can be applied to TSO because the main memory sees a shuffle of the stores issued by both threads: The stores issued by one thread still arrive in their correct order, but the stores of the other thread might interleave at any point.

Using the trick explained above, we may instead assume that each thread has its own memory. We add loops to each thread that may produce arbitrary writes, seemingly coming from the other thread. An intersection then enforces that those writes were actually issued by the other thread.



Since the channels are FIFO, the stores leave the buffer in the order in which they are put in. Instead of guessing the buffered stores of the other thread at the memory (and later verifying the guesses using the intersection), we let the each thread already guess the stores of the other thread when inserting commands into the buffer.



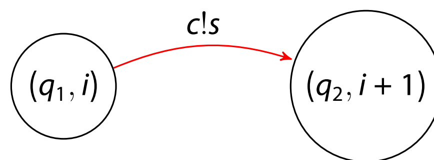
Now the content of both buffers is the same, but they may be propagated into the main memory at different speed: The buffer content of Thread 1 might be `c o n t e n`, with the store `t` already in its memory, while Thread 2 has already seen the stores `e,n,t` and has only `c o n t` remaining in its buffer. We may model this scenario by having one buffer and for each thread having a pointer into the buffer, e.g.

$$\texttt{c o n t } \underset{t_2}{\underset{\uparrow}{\texttt{e}}} \texttt{ n } \underset{t_1}{\underset{\uparrow}{\texttt{t}}}.$$

Exercises

8.6 Exercise: Generalised Lossy Channel Systems
Consider the following variant of LCS: Assume one of the symbols $s \in M$ is **strong**, i.e. can not be lost during sending or receiving in any channel, but the channels can contain at most $k \in \mathbb{N}$ instances of symbol s in total. A transition that wants to send the $k+1^{\text{st}}$ instance of symbol 5s is blocked.

Such an LCS with strong symbol s can be represented by a standard LCS with states $Q \times \{0, \ldots, k\}$, where Q is the set of states of the original system. The resulting transitions are schematically represented below (for $0 \leqslant i < k$).



You are asked to give an implementation of $(q_1, i) \xrightarrow{c!s} (q_2, i+1)$ by several lossy transitions. Your model should check that precisely i symbols s are present in the channel c before appending the extra s.

Hint: Take $M \cup \#$ as the alphabet of the resulting lcs]

Remark: One can show similarly that LCSes with a whole set S of strong symbol, where the total number of strong symbols per channel is bounded, can be simulated by standard LCS.

9. TSO reachability in a bounded number of rounds

As discussed in the previous section, reachability under TSO is decidable. The drawback is that the algorithm we discussed reduces the problem to reachability in lossy-channel systems and thus inherits its bad complexity. To overcome this problem, we present an **underapproximation algorithm**. Instead of checking whether a given program counter can be reached by an arbitrary computation, we check whether it can be reached by a computation in which each thread is only active for a bounded number of rounds. If such a computation exists, then the answer to the unrestricted reachability problem is also positive. If it does not exist, the target state might be unreachable or the bound on the number of rounds might be chosen too low.

In practice, reachability queries are used to detect bugs in programs (e.g. one is interested whether a state in which more than one thread is in the critical section can be reached). In most practical examples, bugs can be found with a low bound on the number of rounds, so we expect this this underapproximation technique to be useful. Nevertheless, for each bound k, one can construct a program such that a certain state is not reachable in k rounds, but in $k + 1$ rounds.

Our goal is to show how to modify a given program P into a program P' whose size is linear in the size of P such that

$$\mathrm{Reach}_{\mathrm{TSO}}^{k\text{-rounds}}(P) = \mathrm{Reach}_{\mathrm{SC}}\left(P'\right) .$$

In other words, the target state is reachable in P under TSO in k rounds if and only if the equivalent location is reachable in P under the strong memory model **sequential consistency (SC)**.

In sequential consistency, the buffer is not used and all stores and loads communicate directly with the main memory. We may either define SC by replacing the rule (STORE) by a rule that stores directly to the main memory without using the buffer, see Exercise 7.6, or we may obtain the set of SC computations as the subset of TSO computations in which each (UPDATE) happens directly after the corresponding (STORE).

Reachability under SC is a standard problem to which many verification techniques apply. In particular, it can be solved in PSPACE (assuming the size of domain of values is given in unary), see Exercise 7.7. Consequently, the resulting algorithm(s) have a much better complexity than the algorithm for unrestricted TSO reachability.

9.1 Remark
When defining parallel programs, we have omitted instructions for e loops, conditions,

conditional jumps and non-determinism. In this section, we will assume that we have such constructs available for SC. The proof for the PSPACE membership of SC reachability can be adapted to allow these instructions. Furthermore, there are standard tools for SC reachability that support these commands.

In particular, we assume that we have an instruction for **non-deterministic branching**, i.e. an instruction of the shape

$$\text{goto}\ell_1 \text{ or goto}\ell_2 \,.$$

It induces two transitions, one in which the computation continues at ℓ_1 and one where it continues at ℓ_2. Note that our transition relation was non-deterministic anyhow, since we cannot choose which thread becomes active at some point in time. Consequently, adding this instruction does not increase the complexity of the reachability problem.

We furthermore assume that there is a way to make a sequence of instructions of a thread **atomic**. While such an atomic block of instructions is executed, other threads cannot interfere: As soon as the first instruction is executed, all instructions in the block have to be executed before any other thread may become active again.

Sources
The theory from this section is from the paper [ABP11]. The presentation is based on Roland Meyer's handwritten notes on the topic,
tcs.cs.tu-bs.de/documents/ConcurrencyTheory_WS_20172018/notes/tso_bounded_round_reachability

We will first formally define what a round is, then define the bounded-round reachability problem and finally explain how the reduction outlined above works.

9.2 Definition
We start by augmenting the transition relation: We redefine $\rightarrow_{\text{TSO}}$ to be a subset of $CF \times TID \times CF$, i.e. we augment each transition by the identifier of the thread that was used for the transition. We have $cf \xrightarrow{i}_{\text{TSO}} cf'$ if $cf \rightarrow_{\text{TSO}} cf'$ according to our old definition and an instruction of thread i was executed or buffer content of thread i has been propagated to the main memory.

A computation

$$\sigma = cf_0 \xrightarrow{i_0}_{\text{TSO}} cf_1 \xrightarrow{i_1}_{\text{TSO}} cf_2 \xrightarrow{i_2}_{\text{TSO}} \dots \xrightarrow{i_{n-1}}_{\text{TSO}} cf_n$$

can be written as a sequence of phases

$$\sigma = \sigma_0.\sigma_1 \dots \sigma_m$$

such that in each phase σ_j, all transitions are made by the same thread $i(j)$, i.e.

$$\begin{aligned}\sigma_0 &= cf_0 \xrightarrow{i(0)}_{\mathrm{TSO}} cf_1 \xrightarrow{i(0)}_{\mathrm{TSO}} \dots \xrightarrow{i(0)}_{\mathrm{TSO}} cf_{n_0}\\ &\vdots\\ \sigma_m &= cf_{n_{m-1}} \xrightarrow{i(m)}_{\mathrm{TSO}} cf_{n_{m-1}+1} \xrightarrow{i(m)}_{\mathrm{TSO}} \dots \xrightarrow{i(m)}_{\mathrm{TSO}} cf_{n_m} = cf_n\end{aligned}$$

We assume that each phase in the decomposition is maximal, i.e. we have $i(j) \neq i(j+1)$ for all j.

A k-**round computation** is a computation σ such that in its phase decomposition, for each thread i, there are at most k phases σ_j with $i(j) = i$.

We use

$$\mathrm{Reach}_{\mathrm{TSO}}(P)k = \left\{pc\colon TID \to LAB \mid \exists \sigma = cf_0 \to^*_{\mathrm{TSO}} (pc, val, buf) k\text{-round computation}\right\}$$

to denote the set of locations reachable by k-round computations.

We are interested in the following decision problem.

9.3 Definition

TSO bounded round reachability

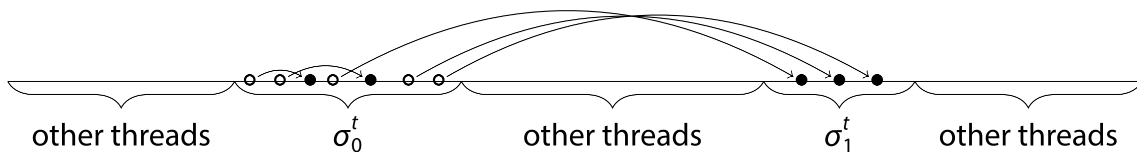
Decide: Program P, program counter pc, bound $k \in \mathbb{N}$

Decide: $pc \in \mathrm{Reach}^{k\text{-rounds}}_{\mathrm{TSO}}(P)$, i.e. is pc reachable by a k-round computation?

Here, we focus on the 2-round case, i.e. we assume that each thread is active at most 2 times. The arguments can be generalized to the k-round case for larger k.

Fix some 2-round computation $\sigma = \sigma_1 \dots \sigma_m$. Note that for each thread i, there are two phases in which it is active, say σ_0^i and σ_1^i (which may be empty).

We fix some thread $t \in TID$ and try to understand the communication of this thread with the rest of the thread. The computation looks as depicted by the following graphic.



Here, circles (∘) mark points in time when a store is issued using the (STORE) rule, and for each circle, the corresponding bullet (•) marks the point in time the store gets propagated to the main memory by the (UPDATE) rule. (We say that the store "does land".)

We make a few observations that altogether will lead us to the definition of the modified program P'.

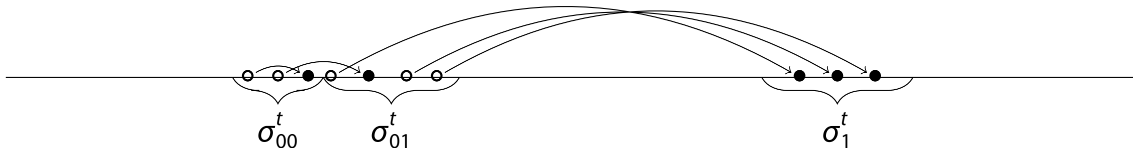
Observation 1: The program counter $pc(t)$, the register valuation $val(r_t)$ for registers of t and the buffer content $buf(t)$ is the same at the end of σ_0^t and at the beginning of σ_1^t, since other threads cannot interfere with these things.

Observation 2: Stores issue by thread t can only land during a phase of thread t.

This is simply because we defined the augmented transition relation accordingly.

Observation 3: Stores land in the order they were issued.

This is because the buffer is FIFO. There is a decomposition $\sigma_0^t = \sigma_{00}^t.\sigma_{01}^t$ of σ_0^t such that all stores issued in σ_{00}^t land during σ_0^t and all stores issued in σ_{01}^t land during σ_1^t.



9.4 Remark
Actually, it may also happen that some stores issued during σ_0^t stay in the buffer until the end of the computation and do not land at all. Because the buffer is FIFO, this would imply that all stores issued during σ_1^t also never land. In the following, we will not consider this special case. It can be dealt with using the same methods outlined below.

Consequence for other threads: Between the two phases of thread t, for each address a, the others threads may see the last store to address a that was made by thread t and has already landed in main memory during σ_0^t. All earlier writes by thread t to address a have already landed and memory and were overwritten by the last write to a that has landed. All later writes will not land until phase σ_1^t.

How to execute a load: To execute a load under TSO, we need to know

- whether a store to the address is still buffered (i.e. whether an early read should occur),
- if so, we need the value of the most recent such store
- if not, we need the value from the main memory.

Since during a phase in which thread t is active, stores issued by other threads do not land (Observeration 2), the buffer of t itself is the only buffer influencing the outcome of a load by thread t. Therefore, to execute a load, we only need the content of the main memory at the beginning of the current phase, and the most recent store issued by the active thread i to the address.

To model this, we introduce a function

$$view: DOM \rightarrow DOM$$

such that $view(a)$ returns the value that should be loaded from address a by the currently active thread.

- The modified program P' will store $view(a)$ for all addresses a in the memory,
- a load from address a in the original program P will be translated into a load of the value $view(a)$.
- at a beginning of each phase, *view* will be updated such that it reflects the view of the thread that is currently active,
- stores made by the thread will immediately update *view*.

How to execute a store: How a store issued by thread t should be handled depends on whether we are in σ_{00}^t or in σ_{01}^t.

Since the other threads do not interfere during σ_0^t, the value for some address a they see in the main memory is the last value written to this address written during σ_{00}^t (Here, we assume that the last write to a is indeed done by thread t). Consequently, we can assume that all stores in σ_{00}^t go directly into the main memory. This is exactly the behavior of stores under SC.

They stores issued during σ_{01}^t will land at some point during σ_1^t, but since other threads do not interfere during σ_1^t, we may assume that they land at the beginning of σ_1^t. Since the buffer is FIFO, later stores during σ_1^t will overwrite earlier stores to the same address.

Consequently, it is sufficient to store the last value stored per address. To this end, for each address a and each thread t, program P' will additionally store two values

$$mask_t(a) \quad \text{and} \quad queue_t(a)$$

where $mask_t(a)$ is non-zero if and only if there is a store buffered by t to address a. In this case, $queue_t(a)$ is the value of the last such store. These values will be used at the beginning of σ_1^t to update the main memory.

We now show how to create from P a program P' such that to the 2-round computation σ in P (under TSO), there is a 2-round computation in P' (under SC) that reaches an equivalent configuration.

Simulating P by P'

The code for program P' is obtained by modifying the code of program P.

We use atomic blocks to enforce that the computation of P' proceeds in rounds. For simplicity, we may assume that the code of thread i in P' consists of k copies of the code of P, where k is the number of rounds. Each copy forms an atomic block that is executed without other threads interfering.

Assertions and local assignments remain unchanged. Memory fences may be omitted, as we assume that P' is executed under SC.

All loads from address a in P will instead load $view(a)$ in P'. The rest of the code will be updated to keep $view$ consistent.

Simulating the first round σ_0^t of some thread t: We use non-deterministic branching to guess the break between σ_{00}^t and σ_{01}^t. In other words, we have two copies of the code of thread t.

- The first copy corresponds to σ_{00}^t. In it, all stores go directly to the main memory, as explained above. View a is also updated.

  After each store operation in this part of the code, non-deterministic branching is used to guess whether we stay inside it, or whether we jump to the second copy.

- The second copy corresponds to σ_{01}^t. Whenever a store operation, say $\text{mem}[r] \leftarrow r'$ is performed, the following happens. Let $a = val(r)$ be the target address of the store, and let $v = val(r')$ be the value that should be stored.
  - $\text{mem}[a]$ is not updated, as the store will not yet land in the main memory,

– *view*(a) is set to v, as the store will immediately become visible to the current thread. Note that loads by thread t will load from *view* to simulate early reads.

– $mask_t(a)$ will be set to 1 to indicate that there is a pending store for address a $queue_t(a)$ will be set to v to store the value that should land in main memory later during σ_1^t.

After the simulation of σ_0^t in P', the main memory (aside from *view*, *mask*, *queue*) is equal to the main memory after the execution of σ_0^t in P under TSO. Consequently, the other threads will not notice the difference. The active thread t itself does not notice the difference as it loads from *view*.

Simulating the second round σ_1^t: Recall that we can assume that all pending stores from σ_0^t land at the beginning of σ_1^t. We iterate over all addresses a and do the following: If $mask_t(a)$ is 0, then there is no pending store for a. If $mask_t(a)$ is 1, then there is a pending store for a and we use it to update the main memory i.e. we set $val(a) = queue_t(a)$.

We rebuild *view* so that *view*(a) again consistently contains the value that should be loaded for address a by thread t by setting $view(a) = val(a)$ for all a.

Similar to σ_0^t, σ_1^t decomposes into two parts: Stores issued during the first part still land during σ_1^t and thus can be directly written to the main memory and *view*. Stores issued during the second part do not land in the main memory at all (as there is no later phase during which they could land). We again guess non-deterministically the break point between the parts. In the second part, we let store operations update *view*(a), but do not update the main memory at all.

More than two rounds

If we have more than two rounds, each round decomposes into several parts: the part containing the stores landing during the same phase, the part containing the stores landing during the next phase, ..., the part containing the stores landing in the last phase (and maybe the stores not landing at all). We thus need $mask_{j,t}$ and $queue_{j,t}$ where j is the round during which the store should land. We leave the details to the reader as an exercise, Exercise 9.8.

9.5 Remark

In the explanation here, we have creates the code in P' out of several copies of P, which will lead to a polynomial blowup of the program size. By storing additional values in the memory, we can get rid of this blowup:

- Instead of having one copy per round, we have just one copy and a memory location $round_t$ that stores in which round the thread is. Whenever the behavior of P' should depend on the round, we query the value of $round_t$ and use conditional statements.
- Instead of having two copies, one for σ_{00}^t and one for σ_{01}^t, we have a memory location *part* that stores in which part we are. Instead of non-deterministically jumping to the second copy, we non-deterministically set *part* to 1. Whenever the behavior of P' should depend on the part in which the round is, we query the value of *part* and use conditional statements.

Using these tricks, we can create a version of P' whose size is linear in the size of P.

9.6 Theorem: Atig, Bouajjani, Parlato, CAV 2011 [ABP11]
For a program given P, we can construct a program P' whose size is linear in the size of P such that $\text{Reach}_{\text{TSO}}^{k\text{-rounds}}(P) = \text{Reach}_{\text{SC}}(P')$.

Using the PSPACE-completeness of SC reachability, see Exercise 7.7, we obtain the following corollary.

9.7 Corollary
TSO bounded round reachability (with k encoded in unary) is PSPACE-complete.

Exercises

9.8 Exercise: Bounded round reachability for $k > 2$
Describe the general case for the bounded round TSO-reachability problem that was described in the lecture. Let P be a parallel program with $n \in \mathbb{N}$ threads and a bound $k \in \mathbb{N}$ on the number of rounds that each thread can make. Explain how to construct a program P' such that for each program counter pc in P and its equivalent program counter pc' in P', the following holds.

$$pc \text{ is TSO-reachable in } P \text{ iff } pc' \text{ is SC-reachable in } P'.$$

Note: You do not have to give a formal construction. It is sufficient to list the additional global variables needed, explain their meaning and how they are used by P'.

10. Robustness against TSO

We consider a different approach to overcome the high complexity of the TSO reachability problem. A programmer usually thinks in terms of sequential consistency, where each possible execution of the program corresponds to a certain interleaving of the source code of all threads. In fact, understanding parallel programs executed under SC already turns out to be difficult.

Hence, any behavior of a program that it exhibits under TSO, but not under SC, should be considered a programming error. We should have

$$B_{TSO}(P) = B_{SC}(P) ,$$

where B is an appropriate definition of behavior. If this holds, we call P **robust against (execution under) TSO**.

The first problem is finding this appropriate definition of behavior. On the one hand, the notion of behavior should be strong enough to guarantee that the executions of the program under TSO are not too drastically different from its execution under SC. In particular, a program that is bug-free when executed under SC should be bug-free under TSO. On the other hand, we want a weak notion: If we enforce that the computations are very similar, we will disallow many TSO-computations and a program will not be robust unless it makes excessive use if memory fences, which decreases performance. In the end, our goal is to come up with an algorithm to check robustness.

Unfortunately, weaker notions are harder to check. For example, consider defining $B = \text{Reach}$, i.e. we say that a program is robust if the locations it can reach by TSO computations are the same as the locations it can reach by SC computations. This is the weakest notion that makes any sens. For a weaker notion, a program could be robust although it can reach an error location under TSO, but not under SC. The bad thing is that checking robustness for this notation is just checking TSO reachability for a polynomial number of locations, and thus as hard as TSO reachability.

In this section, we consider trace-based robustness. This is stronger than the equivalence of reachability sets (in particular, it implies this equivalence), but still weak enough to allow some relaxed TSO executions that cannot happen under SC. As we will show here, checking robustness for this notion is PSPACE-complete. As in the previous section, we reduce checking robustness to checking SC reachability in a modified program, so standard tools apply.

Sources
The theory from this section is from the papers [BMM11] and [BDM13]. The presentation is based on Roland Meyer's handwritten notes on the topic,
tcs.cs.tu-bs.de/documents/ConcurrencyTheory_WS_20172018/notes/tso_robustness.pdf

Traces and trace-based robustness

We start by defining traces. To this end, we augment the transition relation by additional labels as in the previous section, but this time, we put more information into the labels. We define the set of **actions** as

$$ACT = TID \times (\{isu, loc\} \cup (\{ld, st\} \times DOM \times DOM)) \,.$$

Each transition will be labeled by an element from ACT, i.e. a tuple (i, op), where i is the identifier of the thread that performed the operation (as in the previous section) and op is the operation that was performed. The operation op is either a local action loc (an assertion, a register assignment or a memory fence), the issue of a store isu, a load (ld, a, v) which may be an early read or a load from the main memory, or a store (st, a, v) that lands in the main memory. Here, a is the address and v is the value as expected.

For the sake of completeness, we give the formal definition of the labeled transition relation.

10.1 Definition
Consider the same setting as in 7.5, i.e. we consider an instruction of thread i that is executed. Let $\rightarrow_{\text{TSO}} \subseteq CF \times ACT \times CF$ be the smallest relation satisfying the following rules.

$$\text{(EARLY)}\ \frac{\text{<inst>} = r \leftarrow \text{mem}[r'],\, a = val(r'),\, buf(i)\restriction_a = (a = v).\beta}{(pc, val, buf) \xrightarrow{(i,ld,a,v)}_{\text{TSO}} (pc', val[r := v], buf)}$$

$$\text{(LOAD)}\ \frac{\text{<inst>} = r \leftarrow \text{mem}[r'],\, a = val(r'),\, buf(i)\restriction_a = \varepsilon,\, v = val(a)}{(pc, val, buf) \xrightarrow{(i,ld,a,v)}_{\text{TSO}} (pc', val[r := v], buf)}$$

$$\text{(STORE)}\ \frac{\text{<inst>} = \text{mem}[r] \leftarrow r',\, a = val(r),\, v = val(r')}{(pc, val, buf) \xrightarrow{(i,isu)}_{\text{TSO}} (pc', val, buf[i := (a = v).buf(i)])}$$

$$\text{(UPDATE)}\ \frac{buf(i') = \beta.(a = v)}{(pc, val, buf) \xrightarrow{(i',st,a,v)}_{\text{TSO}} (pc, val[a = v], buf[i' := \beta])}$$

$$(\text{MFENCE})\ \frac{\text{<inst>} = \text{mfence}, \mathit{buf}(i) = \varepsilon}{(pc, val, buf) \xrightarrow{(i,loc)}_{\text{TSO}} (pc', val, buf)}$$

$$(\text{ASSERT})\ \frac{\text{<inst>} = \text{assert}\ e, [\![e]\!] \neq 0}{(pc, val, buf) \xrightarrow{(i,loc)}_{\text{TSO}} (pc', val, buf)}$$

$$(\text{ASSIGN})\ \frac{\text{<inst>} = r \leftarrow e, [\![e]\!] = v}{(pc, val, buf) \xrightarrow{(i,loc)}_{\text{TSO}} (pc', val[r := v], buf)}$$

Note that we have flattened nested tuples e.g. we write (i, st, a, v) instead of $(i, (st, a, v))$.

To a computation $\sigma = cf_0 \rightarrow^*_{\text{TSO}} cf$ we associate the sequence of transition labels $\tau \in ACT^*$ of the augmented transition relation, and we write $cf_0 \xrightarrow{\tau}_{\text{TSO}} cf$. Because we want to relate TSO computations to SC computations, we are interested in reaching a configuration in which the buffer has been completely emptied. We formally define the set of **TSO computations** as

$$\mathrm{C}_{\text{TSO}}(P) = \left\{\tau \in ACT^* \,\middle|\, cf_0 \xrightarrow{\tau}_{\text{TSO}} cf \text{ where } cf = (pc, val, buf) \text{ with } buf(i) = \varepsilon \text{ for all } i\right\}.$$

We define the set of **SC computations** $\mathrm{C}_{\text{SC}}(P)$ as the subset of $\mathrm{C}_{\text{TSO}}(P)$ in which each issue (i, isu) is followed by the corresponding store (i, st, a, v). This means we assume that each store is buffered and then directly propagated to the main memory. The resulting effect is as if the store would be written directly to the main memory.

10.2 Example
Consider the main part of Dekker's mutex again.

$$\begin{array}{ll|ll} \ell_0: & \text{mem}[x] \leftarrow 1; \text{goto}\ell_1; & \ell_0': & \text{mem}[y] \leftarrow 1; \text{goto}\ell_1; \\ \ell_1: & r \leftarrow \text{mem}[y]; \text{goto}\ell_2; & \ell_1': & r' \leftarrow \text{mem}[x]; \text{goto}\ell_2; \end{array}$$

The following sequence is a TSO computation, but not an SC computation:

$$\tau = (t_1, isu).(t_1, ld, y, 0).(t_2, isu)(t_2, st, y_1).(t_2, ld, x, 0).(t_1, st, x, 1)\,.$$

The store that is issued in the first action is propagated to the main memory by the very last action.

We could define our behavior based on C (i.e. a program would be robust if $\mathrm{C}_{\text{TSO}}(P) = \mathrm{C}_{\text{SC}}(P)$), but this would be very strong notion, as it would forbid any computation that actually uses the buffer. Instead, we abstract the computation into a trace, a graph that captures its shape.

10.3 Definition
Let $\tau \in C_{TSO}(P)$ be a computation. Its trace $\text{Tr}(\tau)$ is a node-labeled graph

$$\text{Tr}(\tau) = (N, \lambda, \rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src})$$

where

- N is a finite set of nodes,
- $\lambda \colon N \rightarrow ACT$ is the labeling function, and
- $\rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src} \subseteq N \times N$ are relations:
  - the **program order** $\rightarrow_{po}$ relates operations of each thread by the order in which the corresponding instructions occur in the source code, i.e. if $(n, n') \in \rightarrow_{po}$, then the instruction corresponding to n is followed by a goto to the instruction corresponding to n',
  - the **store order** $\rightarrow_{st}$ (also called coherence relation) relates stores to the same address in the order they land in main memory, i.e. if $(n, n') \in \rightarrow_{st}$, then store n' overwrites the value of some address a that was previously set by n, and
  - the **source relation** $\rightarrow_{src}$ (also called reads-from relation) relates loads to the store from which they read, i.e. if $(n, n') \in \rightarrow_{src}$, then n' is a load that reads the value from address a written by store n.

It is defined by induction on τ. The trace associated to the empty computation ε is the empty graph.

Assume that $\text{Tr}(\tau) = (N, \lambda, \rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src})$ is the trace of τ. Then the trace of $\tau.act$ is

$$\text{Tr}(\tau.act) = \left(N \cup \{n\}, \lambda', \rightarrow'_{po}, \rightarrow'_{st}, \rightarrow'_{src}\right)$$

where n is a new or already existing node, depending on act.

If $act = (t, st, a, v)$, then pick the unique minimal node n with respect to $\rightarrow_{po}$ labeled with $\lambda(n) = (t, isu)$. Set $\lambda' = \lambda[n := act]$ and $\rightarrow'_{po} = \rightarrow_{po}$. Intuitively, when a store lands in main memory, it replaces the issue node of the store in the trace. Because the buffer of each thread is FIFO, we need to find the minimal issue node.

If act is of a different type, we add a fresh node $n \notin N$ and set $\lambda' = \lambda \cup \{n := act\}$. Let n_t be the unique maximal node with respect to $\rightarrow_{po}$ with thread identifier t. We set $\rightarrow'_{po} = \rightarrow_{po} \cup \{(n_t, n)\}$. (Node n_t might not exist if act is the first operation by thread t; In this case, we do not modify $\rightarrow_{po}$.)

For stores, i.e. $act = (t, st, a, v)$, we also need to update the store order. Let n_a be the maximal node with respect to $\rightarrow_{st}$ labeled by $(*, st, a, *)$ (i.e. the last store to address a by any thread). We set $\rightarrow'_{st} = \rightarrow_{st} \cup \{(n_a, n)\}$. If n_a does not exist (i.e. if act is the first store to address a) or act is not a store, we have $\rightarrow'_{st} = \rightarrow_{st}$.

If act is a store or a load, we also need to update the source relation. (Otherwise, we have $\rightarrow'_{src} = \rightarrow_{src}$.) If $act = (t, ld, a, v)$, let n_a be the maximal node with respect to $\rightarrow_{st}$ labeled by $(*, st, a, *)$ (i.e. the last store to address a by any thread). We set $\rightarrow'_{src} = \rightarrow_{src} \cup \{(n_a, n)\}$. Intuitively, n reads the value written by the last store to address a that has already been propagated to main memory. (Node n_a may not exist if act loads the initial value; In this case, we do not modify $\rightarrow'_{src}$.)

Note that this may introduce an inconsistent entry: act might load value v and n_a might store some other value v'. This can happen if act performs an early read of a store $a := v$ that has been issued by thread t but has not landed in main memory. We will fix this problem when this store $a := v$ lands.b

If $act = (t, st, a, v)$, we need to update the source relation for all loads in the same thread that can perform an early read form this store (but not for loads from other threads that still saw the old value in the main memory). Let *Early* be the set of actions $n' \in N$ with $n \rightarrow^*_{po} n'$ and $\lambda(n) = (t, ld, a, v)$, i.e. all loads of the same thread from address a that follow the issuing of the store. We set

$$\rightarrow'_{src} = (\rightarrow_{src} \setminus \{(*, n') \mid n' \in Early\}) \cup \{(n, n') \mid n' \in Early\}.$$

We write

$$\mathrm{Tr}_{\mathrm{TSO}}(P) = \mathrm{Tr}(\mathrm{C}_{\mathrm{TSO}}(P)) = \{\mathrm{Tr}(\tau) \mid \tau \in \mathrm{C}_{\mathrm{TSO}}(P)\}$$

to denote the **set of all traces (of TSO computations) of** P, and similarly, $\mathrm{Tr}_{\mathrm{SC}}(P) = \mathrm{Tr}(\mathrm{C}_{\mathrm{SC}}(P))$ for the set of SC traces. Traces provide the right notion of robustness.

10.4 Definition
A parallel program P is **(trace-)robust against TSO** if $\mathrm{Tr}_{\mathrm{TSO}}(P) = \mathrm{Tr}_{\mathrm{SC}}(P)$.

Note that trace-based robustness is strictly stronger than state-based robustness (i.e. requiring $\mathrm{Reach}_{\mathrm{TSO}}(P) = \mathrm{Reach}_{\mathrm{SC}}(P)$.

10.5 Lemma
If $\mathrm{Tr}_{\mathrm{TSO}}(P) = \mathrm{Tr}_{\mathrm{SC}}(P)$ holds, then we also have $\mathrm{Reach}_{\mathrm{TSO}}(P) = \mathrm{Reach}_{\mathrm{SC}}(P)$. The reverse implication does not hold.

Proof: Exercise 10.19. □

Checking robustness is the following decision problem.

10.6 Definition

(Trace-based) Robustness

Decide: Program P

Decide: Is P robust, i.e. does $\mathrm{Tr}_{\mathrm{TSO}}(P) = \mathrm{Tr}_{\mathrm{SC}}(P)$ hold?

We will now first develop a criterion that allows us to check for a trace $\mathrm{Tr}(\tau)$ whether $\mathrm{Tr}(\tau) \in \mathrm{Tr}_{\mathrm{SC}}(P)$ holds. Afterwards, we discuss how one can check whether all TSO traces of a program satisfy the criterion, i.e. whether $\mathrm{Tr}_{\mathrm{TSO}}(P) \subseteq \mathrm{Tr}_{\mathrm{SC}}(P)$ holds. (Note that the other inclusion always holds.)

Our criterion should take a $\mathrm{Tr}(\tau)$ and tell us whether there is a SC-computation $\tau' \in C_{\mathrm{SC}}(P)$ such that $\mathrm{Tr}(\tau) = \mathrm{Tr}(\tau')$. Computation τ' essentially consists of the same actions as τ, they are just scheduled in a different order. In particular, each store needs to be immediately scheduled after the corresponding issue, but this is already taken care of by the fact that in the trace, the store and the issue are represented by a single vertex.

To obtain the criterion, we first define a new order, the **happens-before relation** $\rightarrow_{\mathrm{hb}}$ on $\mathrm{Tr}(\tau)$. The idea is that any SC-scheduling of the actions in τ has to respect $\rightarrow_{\mathrm{hb}}$: If $act \rightarrow_{\mathrm{hb}}^{+} act'$, then act has to be scheduled before act'.

We first note that the three relations that we have already defined should be subsets of $\rightarrow_{\mathrm{hb}}$:

- $\rightarrow_{\mathrm{po}} \subseteq \rightarrow_{\mathrm{hb}}$:
  Just as a TSO scheduling, an SC scheduling has to respect the program order. Actions coming from instructions of the same thread need to be scheduled in the order in which the instructions appear in the source code.

- $\rightarrow_{\mathrm{src}} \subseteq \rightarrow_{\mathrm{hb}}$:
  If some load ld should read the value written by some store st, i.e. $st \rightarrow_{\mathrm{src}} ld$, then in particular, st needs to be scheduled before ld.

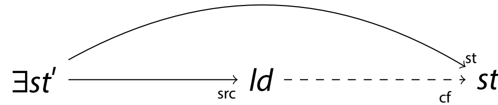
- $\rightarrow_{\mathrm{st}} \subseteq \rightarrow_{\mathrm{hb}}$:
  If some store st' should overwrite the value written by some other store st, ie $st \rightarrow_{\mathrm{st}} st'$, then in particular, st needs to be scheduled before st'.

The relation obtained by uniting $\rightarrow_{po}, \rightarrow_{src}$ and $\rightarrow_{st}$ does not yet characterize the SC schedulability of a trace. We need to add one more relation that is derived from $\rightarrow_{st}$ and $\rightarrow_{src}$ as follows.

10.7 Definition: Conflict relation
Let $\mathrm{Tr}(\tau) = (N, \lambda, \rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src})$ be a trace. The **conflict relation** $\rightarrow_{cf} \subseteq N \times N$ is defined as follows:

$$ld \rightarrow_{cf} st \quad \text{iff} \quad \exists st' \in N\colon st' \rightarrow_{src} ld \text{ and } st' \rightarrow_{st} st$$
$$\text{or } ld \text{ loads the initial value and } st \text{ is the first store on the address .}$$

Illustration:



Intuitively, the load *ld* reads the value from some some store st' that is then overwritten by the store *st*. Consequently, *ld* needs to be scheduled before *st*, otherwise it would read the value stored by *st*. Here, it is important that under SC early reads are not possible.

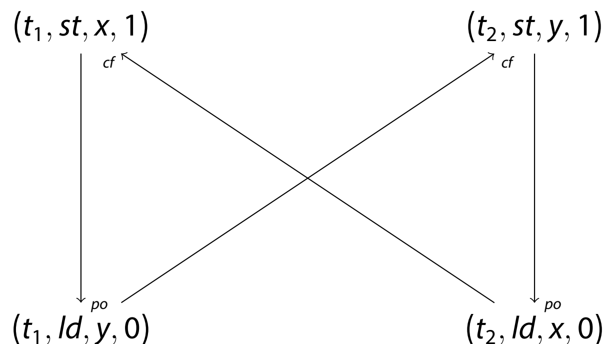
We combine all these relation into a single relation, the **(SC-)happens-before relation**

$$\rightarrow_{hb} = \rightarrow_{po} \cup \rightarrow_{src} \cup \rightarrow_{st} \cup \rightarrow_{cf} \ .$$

10.8 Example
Consider the computation

$$\tau = (t_1, isu).(t_1, ld, y, 0).(t_2, isu)(t_2, st, y_1).(t_2, ld, x, 0).(t_1, st, x, 1)$$

from Example 10.2. Its associated trace is as follows:



As one can see, the happens-before relation forms a cycle. It turns out that this is in fact the general criterion for not being an SC trace. If $\rightarrow_{hb}$ contains a cycle, we have some action *act* with *act* $\rightarrow^{+}_{hb}$ *act*, i.e. it should be scheduled strictly before itself, which is impossible.

10.9 Lemma: Shasha & Snir, TOPLAS 1988 [ShSn88]
Let $\mathrm{Tr}(\tau) \in \mathrm{Tr}_{\mathrm{TSO}}(P)$ be a trace. We have $\mathrm{Tr}(\tau) \in \mathrm{Tr}_{\mathrm{SC}}(P)$ iff $\rightarrow_{hb}$ is acyclic.

Proof sketch:
For one direction, one can prove that under SC, all computations have traces with acyclic happens-before relations.

For the other direction, note that if $\rightarrow_{hb}$ for $\mathrm{Tr}(\tau)$ is acyclic, then its reflexive and transitive closure $\rightarrow^{*}_{hb}$ is antisymmetric. This means $\rightarrow^{*}_{hb}$ is a partial order. Any partial order can be extended to a total order. Let $\rightarrow_{sc}$ be the total order we obtain by extending $\rightarrow^{*}_{hb}$. The SC-computation τ' we obtain be scheduling the instructions in the source code as given by $\rightarrow_{sc}$ has $\mathrm{Tr}(\tau) = \mathrm{Tr}(\tau') \in \mathrm{Tr}_{\mathrm{SC}}(P)$ as desired.

We leave the details to the reader, see Exercise 10.22. □

Let us call a trace $\mathrm{Tr}(\tau) \in \mathrm{Tr}_{\mathrm{TSO}}(P)$ **violating** if it is not contained in $\mathrm{Tr}_{\mathrm{SC}}(P)$. The lemma of Sasha and Snir provides a semantic criterion for a single trace to be violating. It is not at all clear how to check whether a program is robust, i.e. whether all its (potentially infinitely many) traces are non-violating. In the rest of this section, we want to show that it can be checked in PSPACE whether a violating trace exists. To this end, we will proceed as follows:

(1) We define minimal violations, computations whose trace are violating in which the number of delayed stores and the delay are minimal.

(2) We study the shape of these violations. We will see that it is sufficient that one single thread (the "attacker") is delaying stores.

(3) We devise an algorithm to detect such violations.

For (2), we need combinatorial reasoning. For (3), we need algorithm design.

Minimal violations and locality

The key to showing that robustness is to show that it is sufficient to consider computations in which a single thread delays its stores. We start by defining minimal violations.

10.10 Definition: Minimal violation
Consider a computation $\tau = \alpha.a.\beta.b.\gamma \in C_{TSO}(P)$ with Thread(a) = Thread$(b) = t$. Here, Thread(a) refers to the thread to which operation a belongs.

The **distance** between a and b in τ is defined as $\delta_\tau(a, b) = |\beta \downarrow_\tau|$ i.e. it is the number of operations of t that appear in β.

The **number of delays** in τ is given as $\#(\tau) = \Sigma_{isu,st\in\tau}\delta_\tau(isu, st)$ i.e. it is the sum of the distance between all the issues and stores that appear in τ.

A violating computation τ is **minimal** if $\#(\tau)$ is minimal among the number of delays for all the violating computations.

Clearly, if there is a violating computation at all, then there is a minimal one. There may be different computations with the same minimal number of delays. In this case, all of them are minimal.

We wish to prove the following theorem which states that in any minimal violation, only a single thread delays its store.

10.11 Theorem: Locality
In a minimal violation, only a single thread re-orders its action.

Towards proving this theorem, we will prove a sequence of auxiliary lemmas that will be used in the proof of the theorem.

10.12 Lemma
Consider any minimal violation of the form $\tau = \alpha.isu.\beta.st.\gamma \in C_{TSO}(P)$, where isu, st are issue and store instructions of a thread t. Then one of the following holds.

- $\beta \downarrow_t = \varepsilon$ i.e. there are no instructions of t in β
- $\beta \downarrow_t = \beta'.ld.\beta''$ with $addr(ld) \neq addr(st)$ and β'' contains only store instruction. Here ld refers to a load instruction and $addr$ refers to the address to which the load/store instruction performs its action.

In words: A store is either not delayed at all, or it is delayed beyond a load instruction of the same thread that loads a different address.

Proof:
Let us suppose that β contains one or more actions of t, otherwise we are already done.

Case: All actions of t are stores:
Consider the computation $\tau' = \alpha.\beta.isu.st.\gamma$. Clearly, τ' is also a valid TSO computation. Moreover τ' has the same trace as τ, but $\#(\tau') < \#(\tau)$, which contradicts the minimality of τ.

Case: Not all actions of t are stores:
Let a be the last non-store action in $\beta \downarrow_t$, then β can be decomposed as $\beta = \beta_1.a.\beta_2$. This means that all actions of t in β_2 are stores. Notice that a cannot be a fence since stores cannot be delayed beyond a fence. Then one of the following holds.

1. a is a *isu* action.
2. a is a local action.
3. a is a load action.

For the Cases 1, 2 and for the Case 3 with $addr(a) = addr(st)$, we can easily obtain $\tau' = \alpha.isu.\beta_1.\beta_2.st.a$. We have that τ' is a valid TSO computation with the same trace. Furthermore, $\#(\tau') < \#(\tau)$ which contradicts the minimality of τ. □

We next introduce the **happens-before-through** relation. Informally, an action a happens before b through β, if the happens before relation between a and b can be traced through β.

10.13 Definition: Happens-before through
Let $\tau = \alpha.a.\beta.b.\gamma \in C_{\text{TSO}}(P)$. We say a **happens-before** b **through** β if there are subsequences $c_1 \ldots c_n$ of $a.\beta.b$ such that $c_i \rightarrow_{\text{hb}} c_{i+1}$ or $c_i \rightarrow^{*}_{\text{po}} c_{i+1}$ for all $0 \leqslant i \leqslant n$ and $c_0 = a, c_n = b$.

We write $a \rightarrow^{+}_{\text{hb}} b$ though β.

The following lemma states that the happens before relation is robust against insertions. This is easy to see since to establish the relation, we only need a subsequence.

10.14 Lemma
Consider $\tau = \alpha.a.\beta.b.\gamma$ and $\tau' = \alpha'.a.\beta'.b.\gamma'$ such that $\forall t, \tau \downarrow_t = \tau' \downarrow_t$. Moreover assume that β is a subsequence of β'. Then if $a \rightarrow^{+}_{\text{hb}} b$ through β , then $a \rightarrow^{+}_{\text{hb}} b$ through β'.

In the following proposition, we establish a crucial structural property on the shape of a minimal violating computation. This will be used later to prove Theorem 10.11.

10.15 Proposition: Dichotomy
For any minimal violation $\tau = \alpha.a.\beta.b.\gamma$, one of the following holds.

1. $a \rightarrow^{+}_{\mathrm{hb}} b$ though β.
2. There is $\tau' = \alpha.\beta_1.b.a.\beta_2.\gamma$, such that $\mathrm{Tr}(\tau') = \mathrm{Tr}(\tau)$ and $\tau' \downarrow_t = \tau \downarrow_t$ for all t.

In words, a minimal violation either contains a happens-before-through relation between two commands, or they can be reordered to be next to each other while preserving the trace. If we apply dichotomy to an issue and its corresponding delayed store, we obtain an happens-before-through (the case that they can be reordered cannot occur, as it would contradict minimality).

Proof:
Since proving $(1 \vee 2)$ is the same as proving $\neg 1 \implies 2$, we will be proving the latter, i.e. we will assume $\neg 1$ and prove 2. We will proceed by induction on the length of β. We will additionally strengthen our hypothesis as follows.

In addition to the property 2, we will additionally show that β_2 is a subsequence of β.

Base case: We assume $|\beta| = 0$, $\tau = \alpha.a.b.\gamma$ and $a \nrightarrow_{\mathrm{hb}} b$.

Case Thread(a) = Thread(b):
We have $a \rightarrow^{*}_{\mathrm{po}} b$ or $b \rightarrow^{*}_{\mathrm{po}} a$. Since we assume $a \nrightarrow_{\mathrm{hb}} b$, $b \rightarrow^{+}_{\mathrm{po}} a$ has to hold. Consequently, b is a store action delayed beyond a. Swapping a and b will avoid the delay to give $\tau' = \alpha.b.a.\gamma$. This already contradicts minimality since both τ and τ' have the same trace.

Case Thread(a) $\neq$ Thread(b):
One of the following is true.

- One of the action is local
- The actions access different address
- Both are load instructions

In all the three cases, swapping the action produces the required τ'.

Inductive case: For this, we will assume that the statement is true for all β' with $|\beta'| = n$. Consider $\tau = \alpha.a.\beta.c.b.\gamma$ with $|\beta.c| = n+1$. Suppose $a \not\rightarrow^*_{\mathrm{hb}} b$ through $\beta.c$, then we have at least one of $a \not\rightarrow^*_{\mathrm{hb}} c$ and $c \not\rightarrow_{\mathrm{hb}} b$.

Case $a \not\rightarrow^*_{\mathrm{hb}} c$ through β:
In this case we can apply induction hypothesis to a and c. We obtain $\tau' = \alpha.\beta_1.c.a.\beta_2.b.\gamma$ with $\mathrm{Tr}(\tau) = \mathrm{Tr}(\tau')$ and $\tau' \downarrow_t = \tau \downarrow_t$ for all threads t. Now suppose $a \rightarrow^*_{\mathrm{hb}} b$ through β_2 in τ', then we also have $a \rightarrow^*_{\mathrm{hb}} b$ through $\beta \cdot c$ in τ. Hence we have $a \not\rightarrow^*_{\mathrm{hb}} b$ through β_2 in τ'. We can apply the induction hypothesis again to obtain $\tau'' = \alpha.\beta_1.c.\beta_{21}.ba.\beta_{22}.\gamma$ with $\mathrm{Tr}(\tau'') = \mathrm{Tr}(\tau')$ and $\tau'' \downarrow_t = \tau' \downarrow_t$. We further have that β_{22} is a subword of β_2, which is a subword of $\beta.c$ as required.

Case $c \not\rightarrow_{\mathrm{hb}} b$:
In this case, we apply the induction hypothesis to b and c. We obtain $\tau' = \alpha.a.\beta.b.c.\gamma$ with $\mathrm{Tr}(\tau) = \mathrm{Tr}(\tau')$ and $\tau \downarrow_t = \tau' \downarrow_t$ for all threads t. We apply the induction hypothesis to τ' to get the required $\tau'' = \alpha.\beta_1.b.a.\beta_2.c.\gamma$ with $\mathrm{Tr}(\tau'') = \mathrm{Tr}(\tau')$ and $\tau'' \downarrow_t = \tau'' \downarrow_t$ for all threads t, and β_2 is a sub-sequence of β. □

10.16 Corollary
Consider a minimal violation of the form

$$\tau = \tau_1 \cdot isu \cdot \tau_2 \cdot ld \cdot \tau_3 \cdot st\ \tau_4\,,$$

where st is the store corresponding to isu. Then $\mathrm{Tr}(\tau)$ contains the cycle $st \rightarrow^+_{\mathrm{po}} ld \rightarrow^+_{\mathrm{hb}} st$.

Proof:
Notice that $st \rightarrow^+_{\mathrm{po}} ld$ already holds since isu was issued before the ld.

To show that $ld \rightarrow^+_{\mathrm{hb}} st$, we will argue that $ld \rightarrow^+_{\mathrm{hb}} st$ through τ_3. Using dichotomy, Proposition 10.15, one of the following holds:

1. $ld \rightarrow^+_{\mathrm{hb}} st$ through τ_3
2. A computation τ' obtained by re-ordering of ld and st has the property $\mathrm{Tr}(\tau') = \mathrm{Tr}(\tau)$ and $\tau' \downarrow_t = \tau \downarrow_t$.

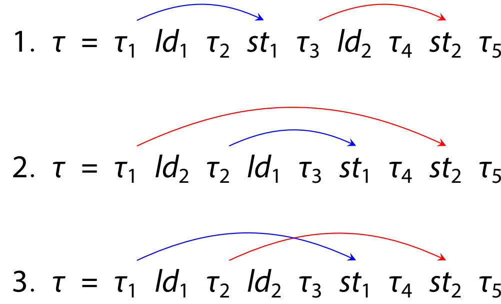
Notice that 2 is impossible since this would violate $\tau' \downarrow_t = \tau \downarrow_t$, where $t = \mathrm{Thread}(ld) = \mathrm{Thread}(st)$. Hence from 1, we get the desired result. □

With these results in place, we are now ready to prove the main Theorem 10.11. Recall that we wanted to prove that in a minimal violation, only one thread re-orders its action.

Proof:
Consider a minimal violation τ. Towards a contradiction, we will assume that at least two threads delay stores. By Lemma 10.12 we have that each store is delayed past a load of the same thread.

Let st_2 be the overall last store that was delayed in τ, and let $t_2 = \text{Thread}(st_2)$ be the corresponding thread. Let ld_2 be the last load that was overtaken by st_2 (i.e. ld_2 was the last load that happened before st_2). Similarly, let st_1 be the overall last store delayed in some thread $t_1 \neq t_2$ and let ld_1 be the last load overtaken by $st1$.

One of the following three situations has to occur.



We argue in each of the cases that τ is not minimal.

Case 1:
Remove the red part, i.e. consider $\tau' = \tau_1.ld_1.\tau_2.st_1.\tau_{st}$, where τ_{st} contains all the stores of t_2 issued before st_1. Clearly, $\#(\tau') < \#(\tau)$. Furthermore, the trace of τ' contains a cycle: We have $st_1 \rightarrow_{po}^{+} ld_1$ and $ld_1 \rightarrow_{hb}^{+} st_1$. Thus, τ' is a a violating computation, a contradiction to the minimality of τ.

Case 2:
Notice that starting from ld_2, thread t_2 does not do any actions except to delay stores until st_2 (by Lemma 10.12). This means ld_2 and all the program-order later actions of t_2 can be removed without effecting the feasibility of the computation. Let $\tau' = \tau_1.\tau_2.ld_1.\tau_3.st_1.\tau_4.st_2$. Notice that we also removed τ_5 since this part can have loads of other threads that can depend on the stores of t_2. Clearly $\#(\tau') < \#(\tau)$ and $\text{Tr}(\tau')$ is cyclic, a contradiction to the minimality of τ.

Case 3: Consider $\tau' = \tau_1.ld_1.\tau_2.ld_2.\tau_3.st_1.(\tau_4 \downarrow_{t_2}).st_2$, obtained by deleting τ_5 and by deleting from τ_4 all the actions that do not belong to thread t_2. We still have $ld_1 \rightarrow_{hb}^{*} st_1$

through $\tau_2 ld_2 \tau_3$, so τ' is cyclic. (Otherwise, we are done by dichotomy.) We also have $\#(\tau') \leqslant \#(\tau)$. We apply dichotomy, Proposition 10.15, to obtain that $ld_2 \rightarrow^*_{\text{hb}} st_2$ through $\tau_3 \cdot st_1 \cdot \tau_4 \downarrow_{t_2}$. Moreover, we also have $st_2 \rightarrow^*_{\text{po}} ld_2$. By Lemma 10.12, we can deduce that ld_1 is the last program order action of thread t_1 in τ'. We delete it to get $\tau'' = \tau_1.\tau_2.ld_2.\tau_3.st_1.(\tau_4 \downarrow_{t_2}).st_2$. We further have $\#(\tau'') < \#(\tau') \leqslant \#(\tau)$ and $\text{Tr}(\tau'')$ is cyclic since $st_2 \rightarrow^*_{\text{po}} ld_2 \rightarrow^*_{\text{hb}} st_2$ continues to hold, a contradiction to minimality. □

Having proved that at most one thread needs to delay its store for any (minimal) violation to occur, we next would like to characterize the set of all possible (minimal) violations as a simpler structure. Our aim is to define such a simple structure and prove robustness by proving absence of this simple structure in the program. For this, we will define what is called an attack.

10.17 Definition: Attack on robustness
An **attack** is a triple $A = (t_A, st, ld)$ where t_A is the thread called the attacker, st, ld are the store and load instructions of t_A. A TSO witness τ for A is a computation of the form shown below that satisfies the properties listed below.

$$\tau = \tau_1 \ \overset{\curvearrowright}{isu \ \tau_2 \ ld_A \ \tau_3 \ st_A} \ \tau_4$$

(W1) Only t_A delays stores.

(W2) st_A is the first store instruction of the attacker that is delayed. ld_A is the last load action of t_A overtaken by st_A.

(W3) For all actions act in $ld_A \cdot \tau_3 \cdot st_A$, we have $ld_A \rightarrow^*_{\text{hb}}$ act.

(W4) Sequence τ_4 only contains the stores of the attacker that were issued before ld_A.

(W5) All these stores st and st_A satisfy $addr(st) \neq addr(ld_A)$.

If a TSO witness for an attack A exists then we call the attack **feasible**.

10.18 Theorem
Program P is robust iff no attacks are feasible.

Proof:
For the ⇒ direction, notice that a TSO witness of an attack already comes with a happens-before cycle

$$st_A \rightarrow_{\text{po}}^{*} ld_A \rightarrow_{\text{hb}}^{+} st_A$$

For the other direction, we will show that if P is not robust, then there is a feasible attack. For this, let us assume that P is not robust, i.e. the set of violating computations is non-empty. We select a minimal violation τ, i.e. a violating computation τ such that $\#(\tau)$ is minimal. By Theorem 10.11, we know that only one thread t_A uses its buffer. The attack that we are going to define will use this thread t_A as attacker. Hence, 1 holds by construction.

Initially, the attacker t_A executes under SC-semantics and stores immediately follow their issue. Eventually, the attacker starts delaying the store. Let st_A be the first store that is delayed by the attacker. Similarly let ld_A be the last load that is overtaken (which has to exist by Lemma 10.12.) This already gives us 2.

The computations looks as depicted in the following figure.

$$\tau = \tau_1 \; isu \; \tau_2 \; ld_A \; \tau_3 \; st_A \; \tau_4$$

We get $ld_A \rightarrow_{\text{hb}}^{*} st_A$ and 3 by dichotomy, Proposition 10.15. With a cycle of the form $st_A \rightarrow_{\text{po}}^{*} ld_A \rightarrow_{\text{hb}}^{*} st_A$, we can already stop with the last action of τ_3, τ_4 only needs to contain stores of the attacker that have been delayed past ld_A. We can further also assume that ld_A to be the last program order action of the attacker. From this, we get 4.

Finally we get 5 by a straight forward application of Lemma 10.12. □

Instrumentation

TODO: Still missing.

Exercises

10.19 Exercise: Trace robustness strictly implies reachability robustness
Prove the following Lemma from the lecture.

a) If $\mathrm{Tr}_{\mathrm{TSO}}(P) = \mathrm{Tr}_{\mathrm{SC}}(P)$ for some program, then $\mathrm{Reach}_{\mathrm{TSO}}(P) = \mathrm{Reach}_{\mathrm{TSO}}(SC)$.

   Here, $\mathrm{Reach}_{\mathrm{TSO}}(P) = \{pc \mid cf_0 \rightarrow^*_{\mathrm{TSO}} (pc, val, buf) \text{ with } buf(i) = \varepsilon \text{ for all } i\}$ and $\mathrm{Reach}_{\mathrm{SC}}(P)$ is obtained by restricting the definition to computations in which each issue (STORE) is followed by the store (UPDATE).

b) The reverse implication does not hold.

10.20 Remark: Relations

Recall the following basic definitions for **relations**.

Let N be a set and let $\leqslant \subseteq N \times N$ be a relation.

Recall that N is **reflexive** if $x \leqslant x$ for all $x \in N$. It is **antisymmetric** if $x \leqslant y$ and $y \leqslant x$ imply $x = y$ (for all $x, y \in N$). It is **transitive** if $x \leqslant y$ and $y \leqslant z$ imply $x \leqslant z$ (for all $x, y, z \in N$). If all three properties hold, we call $\leqslant$ a **partial order**.

A partial order is called **total** (or linear) if any two elements are comparable, i.e.
$\forall x, y \in N\colon x \leqslant y$ or $y \leqslant x$.

We let $\leqslant^*$ denote the reflexive-transitive closure of $\leqslant$, the smallest subset of $N \times N$ that contains $\leqslant$ and is reflexive and transitive.

We may see $(N, \leqslant)$ as a directed graph. We call $\leqslant$ **acyclic** if this graph does not contain a non-trivial cycle $x_0 \leqslant x_1 \leqslant \ldots \leqslant x_m \leqslant x_0$. (Cycles of the shape $x_0 \leqslant x_0$ are trivial.)

10.21 Exercise: Relations

Let N be a **finite** set and let $\leqslant \subseteq N \times N$ be a relation.

a) Explain how to construct $\leqslant^*$ from $\leqslant$ within a finite number of steps.

b) Prove that $\leqslant^*$ is a partial order (i.e. antisymmetric) if and only if $\leqslant$ is acyclic.

c) Now assume that $\leqslant_{po}$ is some partial order. Prove that there is a total order $\leqslant_{to} \subseteq N \times N$ containing $\leqslant_{po}$, i.e. $\leqslant_{po} \subseteq \leqslant_{to}$.

d) (Bonus exercise, not graded.) Do b) and c) still hold if N is infinite?

10.22 Exercise: Shasha and Snir

Prove the Lemma by Shasha and Snir:

A trace $\mathrm{Tr}(\tau) \in \mathrm{Tr}_{\mathrm{TSO}}(P)$ is in $\mathrm{Tr}_{\mathrm{SC}}(P)$ if and only if its happens-before relation $\rightarrow_{\mathrm{hb}}$ is acyclic.

Proceed as follows:

a) Show that for traces of SC computations, $\rightarrow_{\text{hb}}$ is necessarily acyclic.

b) Show how from a trace with acyclic $\rightarrow_{\text{hb}}$, one can construct an SC computation τ' with $\text{Tr}(\tau') = \text{Tr}(\tau)$.

   Hint: Use Exercise 10.21.

10.23 Exercise

Consider two traces $\tau = \alpha.a.b.\gamma$ and $\tau' = \alpha'.a.\beta.b.\gamma'$ where thread$(c) \neq$ thread(a) and thread$(c) \neq$ thread(b) for all c in β. Prove the following:

$$\text{If } a \rightarrow_{\text{hb}} b \text{ in } \text{Tr}_{\text{TSO}}(\tau) \text{ then } a \rightarrow^{+}_{\text{hb}} b \text{ in } \text{Tr}_{\text{TSO}}(\tau')$$

10.24 Exercise

Consider the following program implementing an instance of the **non-blocking write** protocol by H. Kopetz and J. Reisinger:

Thread 1	Thread 2
ℓ_1 : $h \leftarrow \text{mem}[g];\ \text{goto}\ell_2$	ℓ_9 : $h \leftarrow \text{mem}[g];\ \text{goto}\ell_{10}$
ℓ_2 : $\text{mem}[g] \leftarrow h+1;\ \text{goto}\ell_3$	ℓ_{10} : $\text{mem}[g] \leftarrow h+1;\ \text{goto}\ell_{11}$
ℓ_3 : $\text{mem}[x] \leftarrow 42;\ \text{goto}\ell_4$	ℓ_{11} : $\text{mem}[x] \leftarrow 43;\ \text{goto}\ell_{12}$
ℓ_4 : $\text{mem}[g] \leftarrow h+2;\ \text{goto}\ell_5$	ℓ_{12} : $\text{mem}[g] \leftarrow h+2;$
ℓ_5 : $r \leftarrow \text{mem}[g];\ \text{goto}\ell_6$	
ℓ_6 : $v \leftarrow \text{mem}[x];\ \text{goto}\ell_7$	
ℓ_7 : $s \leftarrow \text{mem}[g];\ \text{goto}\ell_8$	
ℓ_8 : assert $r \neq s \lor r$ is odd; $\text{goto}\ell_5$	
ℓ_8 : assert $r = s \land r$ is even;	

Note that there are two instructions labeled by ℓ_8. Assume that when executing gotoℓ_8, the execution non-deterministically jumps to any of them.

Prove that the program is not robust under TSO. Initially assume $\text{mem}[g] = 0$ and $g \neq x$.

10.25 Exercise

Consider a computation $\tau = \tau_1.act_1.\tau_2 \in C_{\text{SC}}(P)$ where for all act_2 in τ_2 we have $act_1 \rightarrow^{*}_{hb} act_2$. Show that the computation $\tau.act$ satisfies $act_1 \rightarrow^{*}_{hb} act$ if and only if

1. there is an action act_2 in $act_1.\tau_2$ with $thread(act_2) = thread(act)$, or
2. act is a load whose address is stored in $act_1.\tau_2$, or
3. act is a store (with issue) whose address is loaded or stored in $act_1.\tau_2$.

10.26 Exercise: The one and only

Consider again the program from Exercise 10.24.

Check whether the following attacks are feasible:

a) $A_1 = (t_1, \ell_4, \ell_5)$,

b) $A_2 = (t_2, \ell_{11}, \ell_6)$.

References

[ABP11] M. F. Atig, A. Bouajjani, and G. Parlato. *Getting Rid of Store-Buffers in TSO Analysis*. In: CAV. 2011, pp. 99–115.

[Ati+10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. *On the verification problem for weak memory models*. In: POPL. 2010, pp. 7–18.

[Ati+12] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. *What's Decidable about Weak Memory Models?* In: ESOP. 2012, pp. 26–46.

[BDM13] A. Bouajjani, E. Derevenetc, and R. Meyer. *Checking and Enforcing Robustness against TSO*. In: ESOP. 2013, pp. 533–553.

[BMM11] A. Bouajjani, R. Meyer, and E. Möhlmann. *Deciding Robustness against Total Store Ordering*. In: ICALP. 2011, pp. 428–440.

[Esp98] J. Esparza. *Decidability and Complexity of Petri Net Problems - An Introduction*. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. Springer, 1998, pp. 374–428.

[FL14] A. Finkel and J. Leroux. *Neue, einfache Algorithmen für Petrinetze*. In: Informatik Spektrum 37.3 (2014), pp. 229–236.

[FL15] A. Finkel and J. Leroux. *Recent and simple algorithms for Petri nets*. In: Software and System Modeling 14.2 (2015), pp. 719–725.

[Kos82] S. R. Kosaraju. *Decidability of Reachability in Vector Addition Systems (Preliminary Version)*. In: STOC. 1982, pp. 267–281.

[Lam92] J. Lambert. *A structure to decide reachability in Petri nets*. In: Theoretical Computer Science 99.1 (1992), pp. 79–104.

[Ler09] J. Leroux. *The General Vector Addition System Reachability Problem by Presburger Inductive Invariants*. In: LICS. 2009, pp. 4–13.

[Ler10] J. Leroux. *The General Vector Addition System Reachability Problem by Presburger Inductive Invariants*. In: Logical Methods in Computer Science 6.3 (2010).

[Ler11a] J. Leroux. *Vector Addition System Reachability Problem: A Short Self-contained Proof*. In: LATA. 2011, pp. 41–64.

[Ler11b] J. Leroux. *Vector addition system reachability problem: a short self-contained proof*. In: POPL. 2011, pp. 307–316.

[Ler12] J. Leroux. *Vector Addition Systems Reachability Problem (A Simpler Solution)*. In: Turing-100. 2012, pp. 214–228.

[Lip09] R. J. Lipton. *An EXPSPACE Lower Bound*. `https://rjlipton.wordpress.com/2009/04/08/an-expspace-lower-bound/`. Blog. 2009.

[Lip76] R. J. Lipton. *The Reachability Problem Requires Exponential Space*. Tech. rep. Yale University, Department of Computer Science, 1976.

[LS15] J. Leroux and S. Schmitz. *Demystifying Reachability in Vector Addition Systems*. In: LICS. 2015, pp. 56–67.

[May81] E. W. Mayr. *An Algorithm for the General Petri Net Reachability Problem*. In: STOC. 1981, pp. 238–246.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[Rac78] C. Rackoff. *The covering and boundedness problems for vector addition systems*. In: TCS 6.2 (1978).

[Rei85] W. Reisig. *Petri nets: An Introduction*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1985.

[ShSn88] D. E. Shasha and M. Snir. *Efficient and Correct Execution of Parallel Programs that Share Memory*. In: ACM Trans. Program. Lang. Syst. 10.2 (1988), pp. 282–312.

[ST77] G. S. Sacerdote and R. L. Tenney. *The Decidability of the Reachability Problem for Vector Addition Systems (Preliminary Version)*. In: STOC. ACM, 1977, pp. 61–76.