TECHNISCHE UNIVERSITÄT DRESDEN

Concurrency Theory

12. Lecture: Petri Net Complexity & Languages

Dr. Stephan Mennicke

Institute for Theoretical Computer Science
Knowledge-Based Systems Group

June 30, 2025

International Center
for Computational Logic

Last Lecture

- an introduction to Petri net properties
- some decision procedures for
  1. bounded Petri nets (e.g., liveness)
  2. general Petri nets (e.g., deadlock-freedom)
- the reachability problem

Today

- Complexity of Petri nets
- Petri net languages

Complexity of Petri Net Problems

Some Complexity Results

Unfortunately, the problems we have discussed have very high computational complexity. We are going to prove them all EXPSPACE-hard.

Boundedness for general Petri nets is EXPSPACE-complete (same for *coverability*).

It was believed that deadlock-freedom, liveness, and reachability are also EXPSPACE-complete. However, all three problems have **non-elementary** complexity.

Detour: Non-Elementary Complexity

Consider the family of functions $(\exp_k)_{k \in \mathbb{N}}$ defined as follows:

- $\exp_0(x) = x$
- $\exp_{k+1}(x) = 2^{\exp_k(x)}$

Class k-EXPSPACE contains all problems that can be solved by a *Turing machine* using at most $\exp_{k(n)}$ space where n for inputs of length n.

$$\bigcup_{k=0}^{\infty} k\text{-Expspace}$$

is the class of all *elementary problems*.

Detour: Non-Elementary Complexity

While $n \mapsto \exp_n(n)$ grows quite fast*, there are functions growing even faster.

Ackermann function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,n) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

All known algorithms for deadlock-freedom, liveness, and reachability for Petri nets have non-primitive recursive runtime.

*still *primitive recursive*, i.e. **For**-loops suffice

Simulating Exponentially Bounded Automata

When n is the size of the machine, an *exponentially bounded automaton* is a Turing machine that uses (i.e., works/writes) at most 2^n tape cells.

A deterministic, exponentially bounded automaton of size n can be simulated by a Petri net of size $\mathcal{O}(n^2)$.

If the given automaton A requires 2^n space, then the corresponding Petri net solves the same problem and requires at least $2^{\mathcal{O}(\sqrt{n})}$ space.

Bounded automata and Petri nets do not fit well. We use a particular kind of *counter programs* (cf. Minsky machines). Petri nets can simulate *bounded counter programs*.

Recall and Adapt: Counter Programs

For program locations ℓ, ℓ_1, ℓ_2 and counter x, the following lines represent all counter program basic commands:

$$\ell : x := x + 1$$

$$\ell : x := x - 1$$

$$\ell : \textbf{goto } \ell_1$$

$$\ell : \textbf{if } x = 0 \quad \textbf{then } \ell_1$$

$$\textbf{else } \ell_2$$

$$\ell : \textbf{halt}$$

We assume all initial counter values to be 0. Semantics is similar to that of Minsky machine, except for the *unconditional* decrement.

Recall and Adapt: Counter Programs

If $\ell : x := x - 1$ is executed on a state in which $x = 0$, the program execution fails and, therefore, the run *aborts*. Proper termination is signaled by reaching the last line of the program containing **halt**.

A counter program is k-bounded if after any step in its unique execution every counter has value $\leq k$.

Theorem 12.1 There is a polynomial time procedures accepting deterministic bounded automata A of size n and returns a counter program C with $\mathcal{O}(n)$ commands simulating A on the empty tape.

Recall and Adapt: Counter Programs

A halts if and only if C halts. If A is exponentially bounded, then C is 2^{2^n}-bounded.

Is is sufficient to show that a 2^{2^n}-bounded counter program of size $\mathcal{O}(n)$ can be simulated by a Petri net of size $\mathcal{O}(n^2)$.

Giving a mathematical description of a Petri net with places and transitions is rather cumbersome. We refer here to **yet another** kind of programs, easy to implement in Petri nets: *net programs*.

Net Programs

Since Petri nets (without extensions) cannot test for 0, net programs replace the conditional jump by a nondeterministic one.

$$\ell : x := x + 1$$

$$\ell : x := x - 1$$

$$\ell : \textbf{goto } \ell_1$$

$$\ell : \textbf{goto } \ell_1 \textbf{ or goto } \ell_2$$

$$\ell : \textbf{gosub } \ell_1$$

$$\ell : \textbf{return}$$

$$\ell : \textbf{halt}$$

Net Programs

The Petri net corresponding to a net program with k commands has $\mathcal{O}(k)$ places, $\mathcal{O}(k)$ transitions, and its initial marking has size $\mathcal{O}(k)$. Thus, size $\mathcal{O}(k^2)$.

Let C be a 2^{2^n}-bounded counter program with $\mathcal{O}(n)$ commands.

We construct a net program $N(C)$ with $\mathcal{O}(n)$ commands, corresponding to a Petri net of size $\mathcal{O}(n^2)$.

$N(C)$ will be nondeterministic, even though C is deterministic. $N(C)$ *simulates* C in the following sense: C halts (i.e., executes the **halt** command) if and only if some computation of $N(C)$ halts (other computations *fail*).

Core Idea: Exploiting Boundedness

Every variable x of $N(C)$ has an auxiliary *complement variable* $\overline{x}$.

$N(C)$ takes care of initializing $\overline{x}$ with 2^{2^n} at the beginning of the program. $N_{init}(C)$

The simulation takes care of the invariant $\overline{x} = 2^{2^n} - x$. $N_{sim}(C)$

Then $N(C) = N_{init}(C); N_{sim}(C)$.

$N_{sim}(C)$

We replace each command of C by a respective (sequence of) net program commands.

$x := x + 1$	$x := x + 1; \overline{x} := \overline{x} - 1$
$x := x - 1$	$x := x - 1; \overline{x} := \overline{x} + 1$
goto ℓ_1	**goto** ℓ_1

if $x = 0$ **then goto** ZERO **else goto** NONZERO

- such conditional jumps are replaced by a subroutine call

$$\mathsf{Test}_n(x, \mathrm{ZERO}, \mathrm{NONZERO})$$

Designing Conditional Jumps

$$\mathsf{Test}_n(x, \mathrm{ZERO}, \mathrm{NONZERO})$$

- If $x = 0$ ($1 \leq x \leq 2^{2^n}$, resp.), some execution of the program leads to ZERO (NONZERO, resp.), and **no** computation leads to NONZERO (ZERO, resp.).
- The program has no side-effects: after any execution leading to ZERO or NONZERO no variable has changed its value.

Since we have to *read* certain counter values, it is easier to design a subroutine with a side-effect first:

$$\mathsf{Test}'_n(x, \mathrm{ZERO}, \mathrm{NONZERO})$$

After any execution leading to ZERO, values of x and $\overline{x}$ are swapped.

Designing Conditional Jumps

$$\mathsf{Test}_n(x, \mathrm{ZERO}, \mathrm{NONZERO}) :$$

$$\mathsf{Test}'_n(x, \mathrm{CONTINUE}, \mathrm{NONZERO})$$

$$\mathrm{CONTINUE} : \mathsf{Test}'_n(x, \mathrm{ZERO}, \mathrm{NONZERO})$$

The Net Program for Test'_n

Since x never exceeds 2^{2^n}, testing $x = 0$ is a nondeterministic choice between

1. decreasing x by 1 — if successful, $x > 0$
2. decreasing $\overline{x}$ by 2^{2^n} — if successful, $\overline{x} = 2^{2^n}$ and $x = 0$

If we choose to decrease x by 1 but $x = 0$, … **discuss**

For the decrement of $\overline{x}$ by 2^{2^n}, use another subroutine $\mathsf{Dec}_n(s_n)$:

- if the initial value of $s_n < 2^{2^n}$, every execution fails;
- if the value of $s_n \geq 2^{2^n}$,
  - all executions terminating with **return** have the effect that $s_n := s_n - 2^{2^n}; \overline{s_n} := \overline{s_n} + 2^{2^n}$;
  - all other executions fail.

The Net Program for Test'_n

$\mathsf{Test}'_n(x, \mathrm{ZERO}, \mathrm{NONZERO}):$

$$
\begin{aligned}
& \textbf{goto } \mathrm{nonzero} \textbf{ or goto } \mathrm{loop}; \\
\mathrm{nonzero} :\ & x := x - 1; x := x + 1; \\
& \textbf{goto } \mathrm{NONZERO} \\
\mathrm{loop} :\ & \overline{x} := \overline{x} - 1; x := x + 1; \\
& s_n := s_n + 1; \overline{s_n} := \overline{s_n} - 1; \\
& \textbf{goto } \mathrm{exit} \textbf{ or goto } \mathrm{loop} \\
\mathrm{exit} :\ & \textbf{gosub } \mathrm{dec}_n; \\
& \textbf{goto } \mathrm{ZERO}
\end{aligned}
$$

The Net Program Dec_n

By induction on n. Dec_0 has to increase s by $2^{2^0} = 2$.

$$
\begin{aligned}
&\mathsf{Dec}_0(s): \\
&\qquad s := s - 1; \overline{s} := \overline{s} + 1; \\
&\qquad s := s - 1; \overline{s} := \overline{s} + 1; \\
&\qquad \textbf{return}
\end{aligned}
$$

For Dec_{i+1} we rely on the subroutine Dec_i. Note,

$$2^{2^{i+1}} = \left(2^{2^i}\right)^2 = 2^{2^i} \cdot 2^{2^i}$$

The Net Program Dec_n

$$
\begin{aligned}
&\mathsf{Dec}_{i+1}(s) : \\
\text{outer\_loop} &: y_i := y_i - 1; \overline{y_i} := \overline{y_i} + 1; \\
\text{inner\_loop} &: z_i := z_i - 1; \overline{z_i} := \overline{z_i} + 1; \\
&\quad s := s - 1; \overline{s} := \overline{s} + 1; \\
&\quad \mathsf{Test}'_i(z_i, \text{inner\_exit}, \text{inner\_loop}); \\
\text{inner\_exit} &: \mathsf{Test}'_i(y_i, \text{outer\_exit}, \text{outer\_loop}); \\
\text{outer\_exit} &: \textbf{return}
\end{aligned}
$$

The Net Program $N_{init}(C)$

Initialize all co-counters with 2^{2^n}

Petri Net Languages

Recall: LTSs and Trace Semantics

Given an LTS $(Q, \Sigma, \longrightarrow)$, we have $s \xrightarrow{t} s'$ $(t \in \Sigma)$ for direct successors and, generally, the *trace relation* $s \xrightarrow{\sigma} s'$ for $\sigma \in \Sigma^\star$. The set of all traces from $s_0 \in Q$ will be denoted $L(s_0)$ and, if a set of terminal states $S_f \subseteq Q$ is specified, the *terminal language* of s_0 is $L_{t(s_0)} = \left\{\sigma \in \Sigma^\star \,\middle|\, \exists s \in S_f : s_0 \xrightarrow{\sigma} s\right\}$.

For a Petri net $N = (P, T, F, m_0)$, we can forge the same definitions, based on the reachability graph of N. It is customary to incorporate a labeling function $\ell : T \to \Sigma$ so that more than one Petri net transition may be associated with the same *action*.

Also the use of internal actions, like τ in CCS, is often used.

Recall: LTSs and Trace Semantics

Definition 12.1 Let Σ be an alphabet with $\tau \notin \Sigma$ (reserved for internal actions). We call a tuple $N = (P, T, F, m_0, \Sigma, \ell)$ a *labeled Petri net over* Σ if (P, T, F, m_0) is a Petri net and $\ell : T \to \Sigma \cup \{\tau\}$. If $\forall t \in T : \ell(t) \neq \tau$, we call N a τ-free Petri net.

A *terminal Petri net* is a tuple $N = \left(P, T, F, m_0, M_f\right)$ where (P, T, F, m_0) is a Petri net and $M_f \subseteq \mathbb{N}^P$ is a set of *final markings*. Analogously to labeled Petri nets, we obtain the notion of *terminal labeled Petri nets* by introducing a respective labeling function on the set of transitions.

Recall: LTSs and Trace Semantics

A (terminal) labeled Petri net N is called *free* if $\Sigma = T$ and $\ell = \mathrm{id}_T$.
For a (terminal) labeled Petri net N, define the *free version of* N by $N^f := (P, T, F, m_0, M_f, T, \mathrm{id}_T)$.

Associating τ with the empty word ε helps us extending labeling functions to homomorphisms $\ell : T^\star \to \Sigma^\star$ as follows:

1. $\ell(\varepsilon) := \varepsilon$;
2. $\ell(t) := \begin{cases} \varepsilon & \text{if } \ell(t) = \tau \\ \ell(t) & \text{otherwise;} \end{cases}$
3. $\ell(\sigma t) := \ell(\sigma)\ell(t)$

Petri Net Languages

$$L(N) := \left\{ w \in \Sigma^\star \,\middle|\, \exists \sigma \in T^\star : m_0 \xrightarrow{\sigma} \wedge\, w = \ell(\sigma) \right\}$$

$$L_t(N) := \left\{ w \in \Sigma^\star \,\middle|\, \exists \sigma \in T^\star, m \in M_f : m_0 \xrightarrow{\sigma} m \wedge \ell(\sigma) = w \right\}$$

- $\mathcal{L}^\tau$ class of all languages of arbitrary Petri nets
- $\mathcal{L}$ class of all languages of τ-free Petri nets
- $\mathcal{L}^f$ class of all languages of free Petri nets
- $\mathcal{L}_t^\tau$ class of all terminal languages of arbitrary Petri nets
- $\mathcal{L}_t$ class of all terminal languages of τ-free Petri nets
- $\mathcal{L}_t^f$ class of all terminal languages of free Petri nets

On Language Equivalence

Recall

- For Petri nets, bisimilarity is undecidable.
- Bisimilarity for deterministic Petri nets is the same as language (i.e., trace) equivalence.

Definition 12.2 For two Petri nets N_1 and N_2, the *language equivalence problem* asks if $\mathcal{L}(N_1) = \mathcal{L}(N_2)$.

We denote the problem by LEP or LEP_t (for terminal language equivalence). Likewise, we have LEP^τ and LEP^f (together with their terminal counterparts LEP_t^τ and LEP_t^f).

On Language Equivalence

Theorem 12.3 LEP and LEP_t are undecidable.

Proof. see exercise tomorrow … ∎

Theorem 12.4 LEP^f is decidable.

- free Petri nets are also called *unlabeled Petri nets*
- although the Petri net firing rule is nondeterministic also if the underlying Petri net is free, free Petri nets are deterministic from a language point-of-view

Deciding Language Equivalence for Free Petri Nets

Let $N_1 = (P_1, T_1, F_1, m_1, \ell_1)$ and $N_2 = (P_2, T_2, F_2, m_2, \ell_2)$ be unlabeled Petri nets with disjoint sets of nodes.

Starting from the Petri net

$$N_1 + N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, m_1 + m_2, \ell_1 \cup \ell_2),$$

- add a duplicate transition t' for each transition $t \in T_1 \cup T_2$ with same label, pre-, and postset as t; sets denoted by T_1' and T_2'
- add a fresh place p with arcs $\{p\} \times (T_1 \cup T_2)$ and $(T_1' \cup T_2') \times \{p\}$;
- for each label $a \in \Sigma$, add places p_1^a, p_2^a and for $t \in T_i$ with $\ell_i(t) = a$, add arcs $\left(t, p_j^a\right), \left(p_j^a, u'\right)$ for transition duplicate $u' \in T_j$ with $\ell_j(u') = a$;
- add a transition t_ω with arcs $(p, t_\omega), (t_\omega, p)$

Finally, N_1 and N_2 produce the same sets of traces if and only if the constructed net is deadlock-free.

Closure Properties of Petri Net Languages

1. Every regular language is a Petri net language;
2. Shuffle, union, concatenation $\mathcal{L}, \mathcal{L}_t, \mathcal{L}^\tau, \mathcal{L}_t^\tau$
3. Synchronization $\mathcal{L}, \mathcal{L}_t, \mathcal{L}^\tau, \mathcal{L}_t^\tau$
   - Restriction, Hiding
   - Intersection also $\mathcal{L}^f$ and $\mathcal{L}_t^f$

But what about languages of free Petri nets?