Lipton 1 ____ 2
Lipton 2 ____ 7

3. Coverability is EXPSPACE-hard, Lipton '76

Presentation follows the survey article

Decidability and Complexity of Petri Net Problems — An Introduction

by Javier Esparza '96.

Rule of thumb:

- All interesting questions about the behavior of (general) Petri nets are EXPSPACE-hard.
- More precisely, they require at least $2^{O(n)}$ space where n is the size of the input.

Follows (for all problems) from one fact:

Reduction:

- A deterministic exponentially-bounded Turing machine of size n can be simulated by a PN of size $O(n)$
- There is a polynomial-time procedure to construct this net.

Problem / Solution:

- Turing machines and Petri nets do not fit well.
- Known:

$$\text{Bounded Turing machines} \underset{\text{(can be simulated by)}}{\overset{(1)}{\leq}} \text{Bounded counter programs} \underset{\text{(Lipton's result)}}{\overset{(2)}{\leq}} \text{Petri nets.}$$

Concerning (1):

There is a polynomial-time procedure that

accepts a deterministic Turing machine A of size n
and returns a counter program C_A with $O(n)$ commands
that satisfies the following

↳ C_A simulates the computation of A on the empty tape.

↳ In particular: C_A halts iff A halts.

↳ Moreover: if A is exponentially-bounded,
then C_A is 2^{2^n}-bounded.

Construction:

↳ A Turing machine over $\{0,1\}$ can be simulated
by two stacks (over $\{0,1\}$):

- cut the tape in half
- moving the head left
is mimicked by
popping a bit from the left stack
and pushing it onto the right stack.

↳ A stack over $\{0,1\}$ can be simulated by two counters
- One counter holds the value
represented in binary by the bits on the stack
(least significant bit on top).
- Pushing a 1 onto a stack representing value $v \in \mathbb{N}$
is implemented by computing

$$2v+1.$$

↳ A stack over $\{0,1\}$ of size at most 2^{n^k}
can be simulated by two counters
that are bounded by $2^{2^{n^{k'}}}$.

Hence:
- A 1-tape Turing machine with space 2^{n^k}
can be simulated by 4 counters bounded by $2^{2^{n^{k'}}}$.
- Minsky '67: ↳ A Turing machine can be simulated by a 2-counter machine
↳ But the values need one more exponential.

Concerning (2): Goal of this lecture

↳ Using (1), it is sufficient to simulate a 2^{2^n}-bounded counter program of size $O(n)$ by a PN of size $O(n)$.

↳ We develop such an encoding.

3.1 Petri Net Programs

↳ Encode counter programs into PN programs

↳ A PN program is a convenient description of a PN

↳ The PN semantics (translation from PN programs to PN) is easy:
every command yields a transition.

Commands:

l: $x := x+1$	
l: $x := x-1$	(only if $x>0$)
l: goto l_1	// unconditional jump
l: goto l_1 or goto l_2	// non-deterministic jump
l: gosub l_1	// subroutine call
l: return	// subroutine return
l: halt	// stop execution.

Syntactic correctness:

↳ Decompose a program into
- main program that only calls 1st level subroutines
- 1st level subroutines only call 2nd level subroutines

etc.

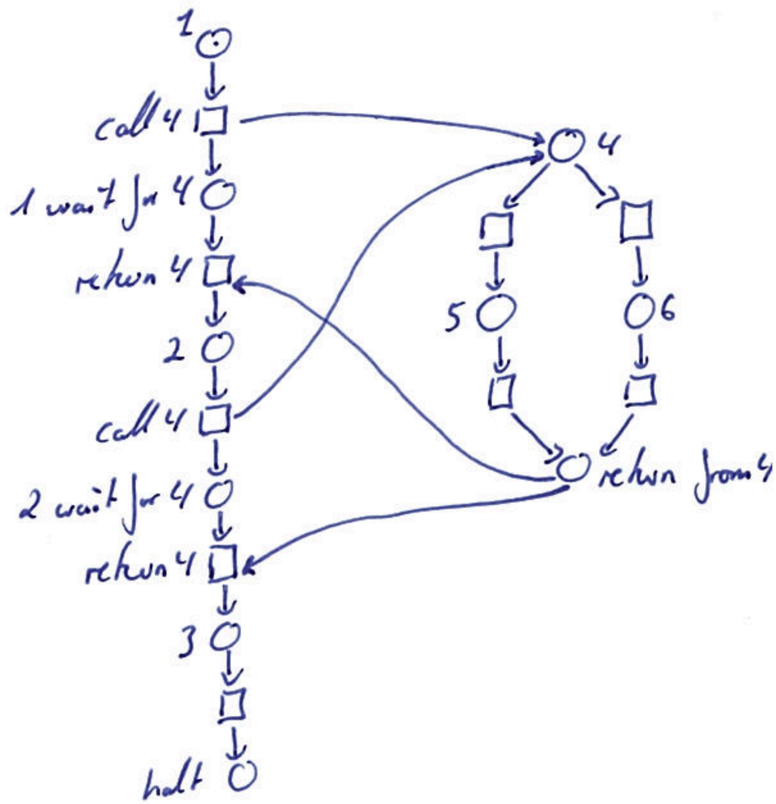
↳ All programs given here are well-structured (in this sense).

PN Semantics:

- Immediate for all commands except subroutine calls/returns
- Semantics of subroutines on an example

```
main           { 1: gosub 4;
                 2: gosub 4;
                 3: halt;

1st level      { 4: goto 5 or goto 6;
subroutine       5: return;
                 6: return;
```



Note:

- The same part of the PN
  (subroutine 4)
  can be used several times.
- This saves exponential space /
  keeps the PN exponentially smaller.
- Illustration:

  routine a calls 2x routine b
  routine b calls 2x routine c
  routine c calls 2x routine d
  } a calls 8x d.

Size of the PN Semantics

- PN Semantics for a PN program with k commands
  has
  $O(k)$ places, $O(k)$ transitions, and $O(k)$ edges.
- The initial marking is also of size $O(k)$.
  - Altogether, the PN for the PN program
    has a size of $O(k)$.

3.2 Construction

Goal:
- Consider a 2^{2^n}-bounded counter program C with $O(n)$ commands.
- Construct a PN program NP_C with $O(n)$ commands that simulates C.
- PN program NP_C corresponds to a PN of size $O(n)$.

The notion of simulation:
- Note that C is deterministic while NP_C will be non-deterministic.
- Ensure

  C halts iff some computation of NP_C halts
  (to be distinguished from fail).

Technically:
- Each variable x of NP_C has a complement variable $\bar{x}$.
- The following invariant will hold:

  $$\bar{x} = 2^{2^n} - x.$$

- The program is as follows

  $$\boxed{NP_C = NP_C^{init};\ NP_C^{sim}}$$

  ↳ NP_C^{init} sets $\bar{x}$ to 2^{2^n}

  ↳ NP_C^{sim} simulates C, preserving the invariant.

3.2.1 Construction of NP_C^{sim}:

Replace each command of C by a PN program:

$x := x+1 \rightsquigarrow x := x+1;\ \bar{x} := \bar{x} - 1;$

$x := x-1 \rightsquigarrow x := x-1;\ \bar{x} := \bar{x} - 1;$

l: goto l' $\rightsquigarrow$ l: goto l';

Translation of

l: if $x=0$ then goto lzero
  else goto lnonzero;

Construct a PN program

$Test_n$(x, lzero, lnonzero)

as follows.

Specification of $Test_n$:

- If $x=0$ then some execution of the program leads to lzero and no execution leads to lnonzero.
- If $1 \leq x \leq 2^{2^n}$ then some execution of the program leads to lnonzero and no execution leads to lzero.
- The program $Test_n$ has no side-effects: after any execution leading to lzero or lnonzero, no variable has changed its value.

Program $Test_n$(x, lzero, lnonzero):

  $Test_n'$(x, lcont, lnonzero);

lcont: $Test_n'$($\bar{x}$, lzero, lnonzero);

Program Test$_n'$:

- Easier to design
- But has a side-effect
  after an execution leading to true
  the values of x and $\bar{x}$ are swapped.
- Swapping twice ensures we have no side-effect for Test$_n$
- Remaining specification coincides with Test$_n$.

Construction of Test$_n'$:

Idea:
- Since $x \le 2^{2^n}$, testing $x = 0$ can be replaced by
  ↳ decrementing x $(x := x - 1)$ and if we succeed $\Rightarrow x > 0$
  ↳ decreasing $\bar{x}$ by 2^{2^n} and if we succeed $\Rightarrow \bar{x} = 2^{2^n}$
  $\Rightarrow x = 0$
  (Invariant)
- If we guess incorrectly (decrement although $x = 0$, decrease although $\bar{x} < 2^{2^n}$), PN program blocks.

Problem:
- Decrease $\bar{x}$ by 2^{2^n}
- Done by function Dec$_n$ below.

Specification of Dec$_n$(s_n):

- Makes use of an auxiliary variable s_n (depends on n)
- If the initial value of s_n is $< 2^{2^n}$, Dec$_n$ fails.
- If the value of s_n is $= 2^{2^n}$, then all executions of Dec$_n$ that terminate with a return command have the effect

  $s_n := s_n - 2^{2^n}$ and $\bar{s}_n := \bar{s}_n + 2^{2^n}$.
- There are no side-effects
- All other executions fail.

Program $Test_n'(x, l_{zero}, l_{nonzero})$:

goto l_{pos} or goto l_{loop}; // non-deterministic choice

l_{pos} : $x := x - 1$; $x := x + 1$; goto $l_{nonzero}$ // x is not zero

l_{loop} : $\bar{x} := \bar{x} - 1$; $x := x + 1$; $s_n := s_n + 1$; $\bar{s}_n := \bar{s}_n - 1$; // move $\bar{x}$ to s_n, maintain the invariant

goto l_{exit} or goto l_{loop}; // continue moving $\bar{x}$ or decide for having moved everything

l_{exit} : gosub Dec_n; goto l_{zero};

// Once everything has been transferred, call Dec_n.

// If successful, continue at l_{zero}.

Construction of Dec_n:

Proceed by induction on n:

Base case: $n = 0$ — Dec_0 decreases by $2^{2^0} = 2^1 = 2$.

Subroutine $Dec_0(s_0)$:

$s_0 := s_0 - 1$; $\bar{s}_0 := \bar{s}_0 + 1$;
$s_0 := s_0 - 1$; $\bar{s}_0 := \bar{s}_0 + 1$;
return;

Induction hypothesis: Assume Dec_i is already known, and hence we can use $Test_i'$.

Induction step:

- To construct Dec_{i+1}, note the following trick:

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

- Decrease by 2^{2^i} for 2^{2^i} many times.

Implementation:

Two nested loops

```
while {
    while {
        s_{i+1} := s_{i+1} - 1      } Execute 2^{2^i} times
    }                               } Execute 2^{2^i} times.
}
```

↳ Loop variables counted from 2^{2^i} to 0

↳ Termination by testing loop variable for being 0

$\Rightarrow$ Test!

Subroutine $Dec_{i+1}(s)$:

// initially $y_i = 2^{2^i} = z_i = \overline{s_i}$, $\overline{y_i} = 0 = \overline{z_i} = s_i$

// initialization by NP_C^{init}.

```
lout :  y_i := y_i - 1; ȳ_i := ȳ_i + 1;                    }
  lin : z_i := z_i - 1; z̄_i := z̄_i + 1;          }           }
        s_{i+1} := s_{i+1} - 1; s̄_{i+1} := s̄_{i+1} + 1;  } inner loop  } outer loop
        Test!(z_i, lindone, lin);                 }           }
lindone : Test!(y_i, loutdone, lout);                        }
loutdone : return;
```

Note that Test! undoes the swapping of z_i and $\overline{z_i}$ when lindone is taken.

Similar for y_i and $\overline{y_i}$ when loutdone is taken in the second Test!.

3.2.2 Construction of NP_C^{init}

Variables that have to be initialized by NP_C^{init}:

- $x_1, \dots, x_\ell$ of counter program C with initial value 0
  $\overline{x_1}, \dots, \overline{x_\ell}$ with initial value 2^{2^n}
- $s_0, \dots, s_n$ have initial value 0
  $\overline{s_i}$ has initial value 2^{2^i} for all $0 \leq i \leq n$.
- y_i, z_i have initial value 2^{2^i} for all $0 \leq i \leq n-1$
  complement variables $\overline{y_0}, \overline{z_0}, \dots, \overline{y_{n-1}}, \overline{z_{n-1}}$ have initial value 0.

Program NP_C^{init}:

$inc_0(\overline{s_0}, y_0, z_0);$
$inc_1(\overline{s_1}, y_1, z_1);$
$\vdots$
$inc_{n-1}(\overline{s_{n-1}}, y_{n-1}, z_{n-1});$
$inc_n(\overline{s_n}, \overline{x_1}, \dots, \overline{x_\ell});$

Here, we invoke programs $inc_i(v_1, \dots, v_m)$ that yield

$v_1 := v_1 + 2^{2^i}$
$\vdots$
$v_m := v_m + 2^{2^i}$.

No side-effects.

Definition of programs inc_i:

Again by induction, similar to decrement

Program $inc_0(v_1, \dots, v_m)$:

$v_1 := v_1 + 1;\ v_1 := v_1 + 1;$
$\vdots$
$v_m := v_m + 1;\ v_m := v_m + 1;$

Program $Inc_{i+1}(v_1, \ldots, v_m)$:

// Initially. $y_i = 2^{2^i} = z_i = \bar{s}_i$

$\bar{y}_i = 0 = \bar{z}_i = s_i$

// This assumption uses the induction hypothesis.
Note that NP_c^{init} calls Inc_0 to Inc_n
in the required order.

l_{outer}: $y_i := y_i - 1$; $\bar{y}_i := \bar{y}_i + 1$;

l_{inner}: $z_i := z_i - 1$; $\bar{z}_i := \bar{z}_i + 1$;

$v_1 := v_1 + 1$;

⋮

$v_m := v_m + 1$;

$Test_i'(z_i, l_{innerdone}, l_{inner})$;

$l_{innerdone}$: $Test_i'(y_i, l_{outerdone}, l_{outer})$;

$l_{outerdone}$: <u>return</u>;

3.3 Complexity

- Instance of $Test_n$ for each conditional jump } constant length
- <u>One</u> instance of $Dec_n, \ldots, Dec_0$ }
- <u>One</u> instance of $Inc_n, \ldots, Inc_0$
  - ↳ Instances $Inc_0, \ldots, Inc_{n-1}$ have constant size
  - ↳ Instance Inc_n has size $O(n)$.

Together: $O(n)$

Petri net: $O(n)$.

□