Proof Spaces

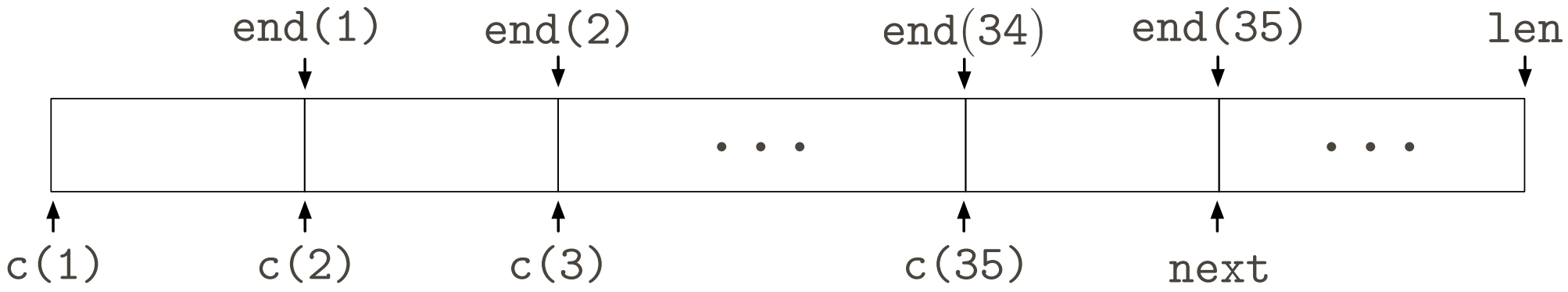
Andreas Podelski

Azadeh Farzan
Zachary Kincaid
Matthias Heizmann
Jochen Hoenicke

```
   global int len; //   length of array
   global int array(len) : tasks; //   array of tasks
   global int next; //   position of next available task block
   global lock m; //   lock protecting next

   thread T:
         local int : c;       //   position of current task
         local int : end; //   position of last task in acquired block
         //   acquire block of tasks
1        lock(m);
2              if(next + 10 <= len)
3                    { c := next; next := next + 10; end := next; }
4              else
5                    { c := next; next := next + 10; end := len; }
6        unlock(m);
         //   perform block of tasks
7        while (c < end):
8              tasks[c] := 0; //   mark task c as started
               ...                              //   work on the task c
9              tasks[c] := 1; //   mark task c as finished
10             assert(tasks[c] == 1); //   no other thread has started task c
11             c := c + 1;
```

```
   global int len; //   length of array
   global int array(len) : tasks; //   array of tasks
   global int next; //   position of next available task block
   global lock m; //   lock protecting next

   thread T:
         local int : c;      //   position of current task
         local int : end; //   position of last task in acquired block
         //   acquire block of tasks
1        lock(m);
2              if(next + 10 <= len)
3                    { c := next; next := next + 10; end := next; }
4              else
5                    { c := next; next := next + 10; end := len; }
6        unlock(m);
         //   perform block of tasks
7        while (c < end):
8              tasks[c] := 0; //   mark task c as started
               ...                              //   work on the task c
9              tasks[c] := 1; //   mark task c as finished
10             assert(tasks[c] == 1); //   no other thread has started task c
11             c := c + 1;
```

```
thread T:
        local int : c;      //  position of current task
        local int : end; //  position of last task in acquired block
        //  acquire block of tasks
1       lock(m);
2           if(next + 10 <= len)
3                   { c := next; next := next + 10; end := next; }
4           else
5                   { c := next; next := next + 10; end := len; }
6       unlock(m);
```



threads $1, 2, \ldots, 35$
have acquired block of tasks
have not yet started working

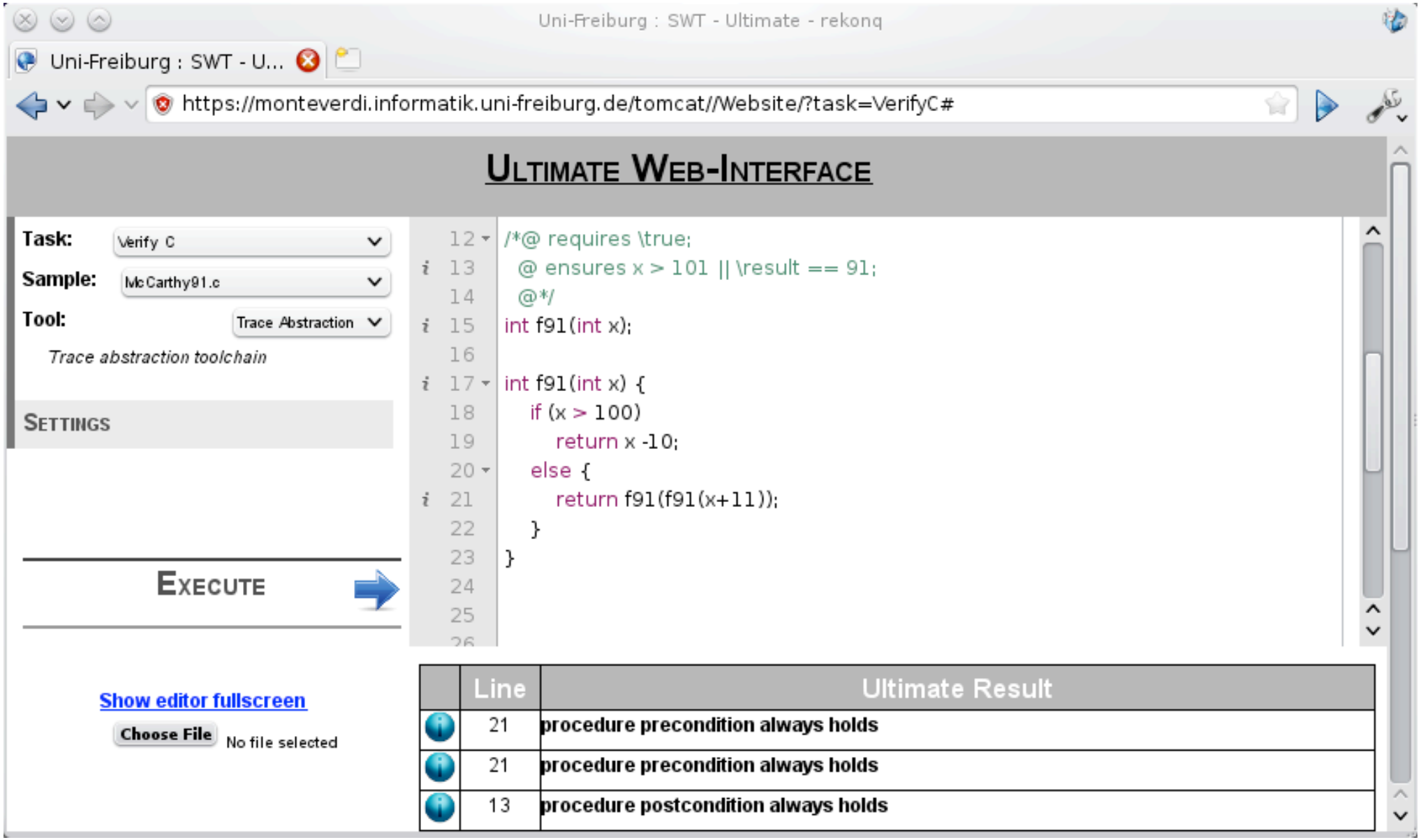
proof spaces

- new paradigm for automatic verification
- sequential/concurrent/parametrized programs
- automata

automated verification

termination ..... Buchi automata
recursion ..... nested word automata
concurrency ..... alternating finite automata
parametrized ..... predicate automata
proofs that count ..... Petri net $\subseteq$ counting automaton

Ultimate Automizer



VW-Abgas-Affäre: Bosch weist Mitschuld von sich

Die Technik für die umstrittenen Dieselmodelle in der VW-Abgas-Affäre lieferte Bosch. Doch an der Manipulation will das Unternehmen nicht beteiligt gewesen sein: Die Verantwortung liege allein beim Autobauer.

simplify task for program verification:

Don’t give a proof.

Show that a proof *exists*.

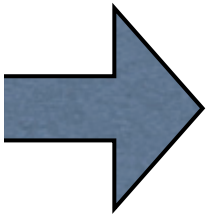
automata:
existence of accepting run

inclusion check:
show that, for every word in the given set,
an accepting run *exists*

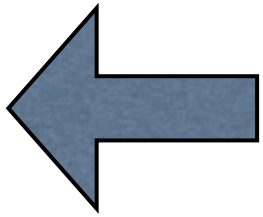
simplify task for program verification:

Show that,
for every program execution,
a proof exists.

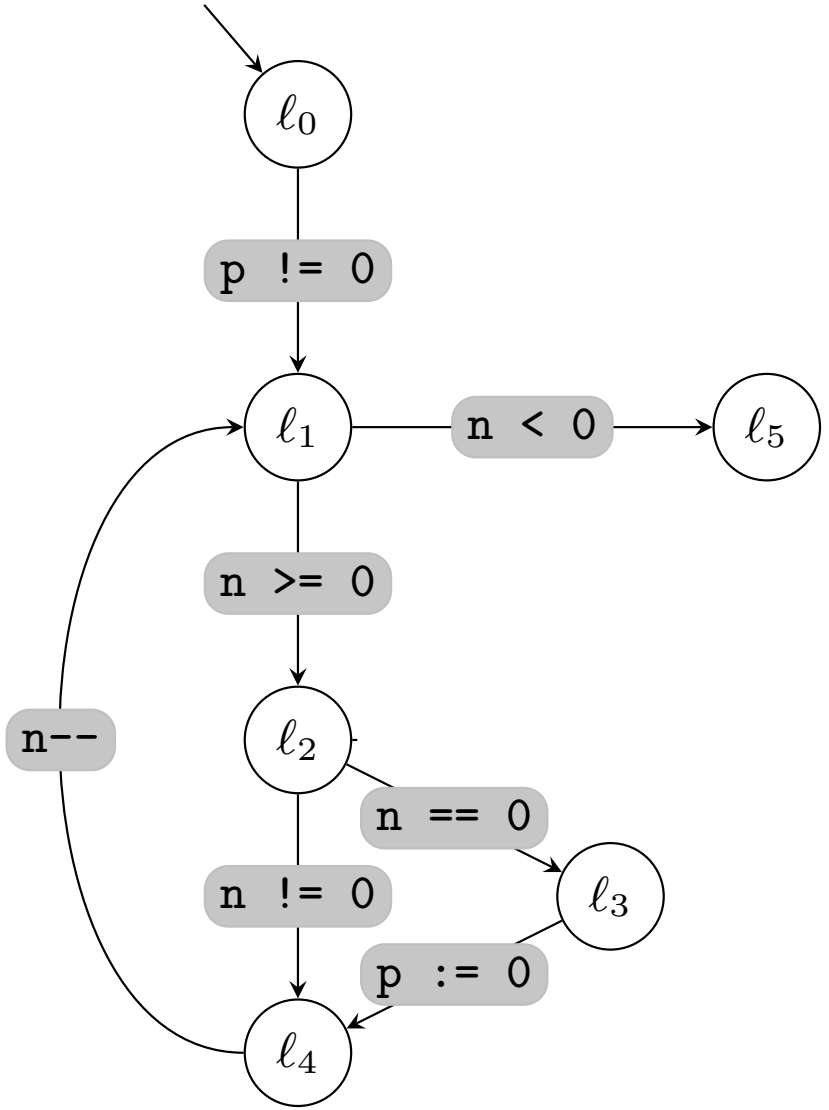
proof spaces



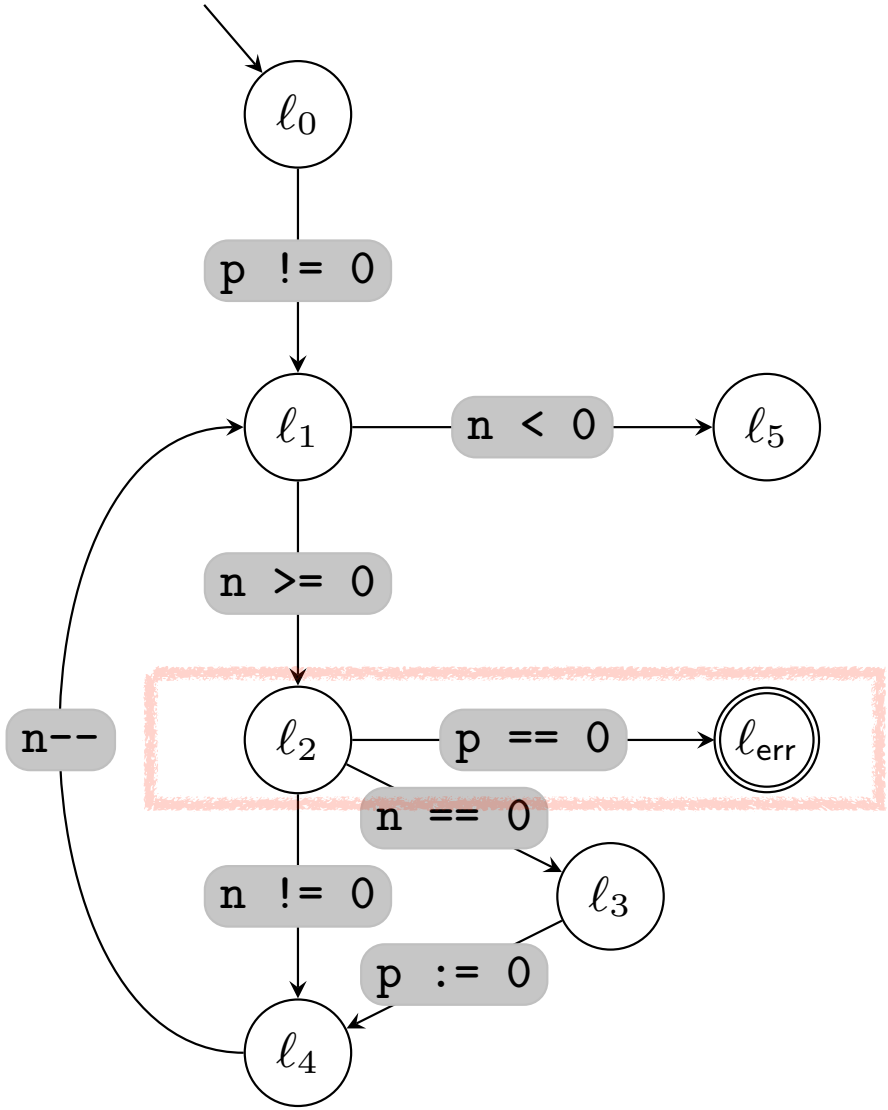
- automata from unsatisfiability proofs
- proof spaces



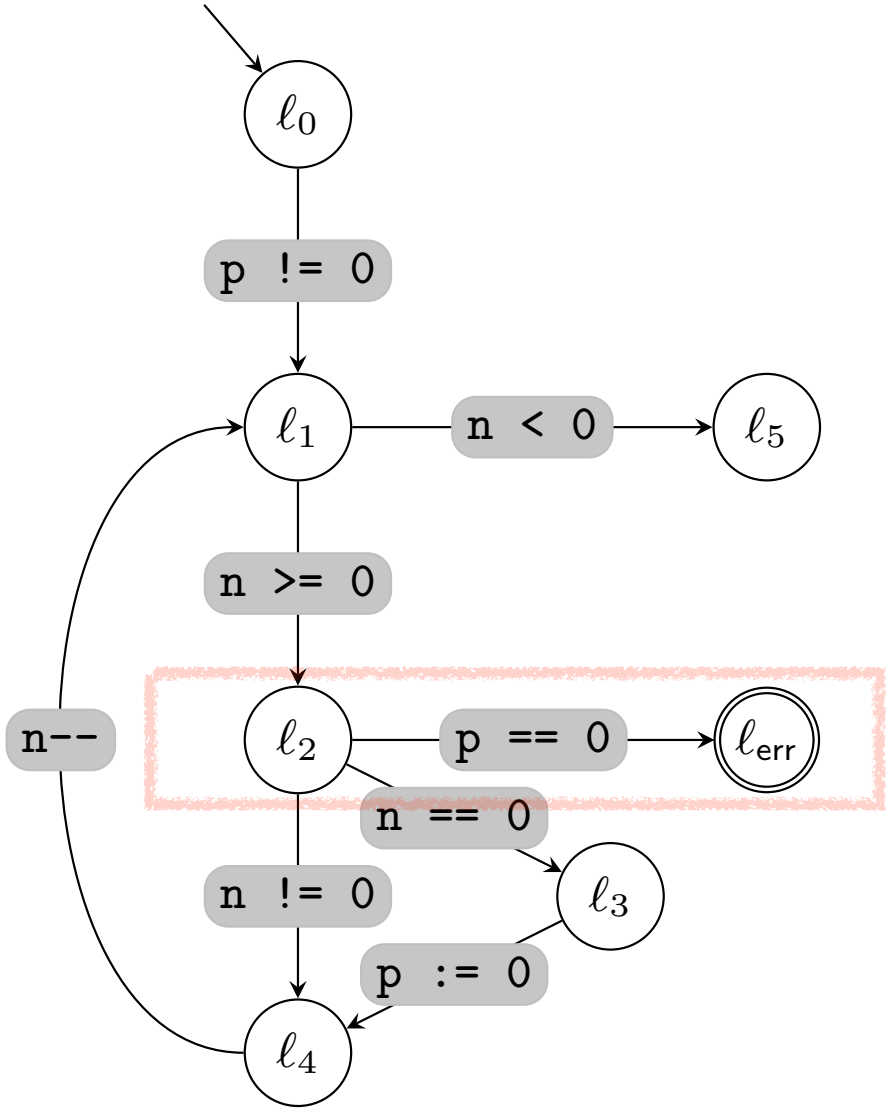
```
ℓ0: assume p != 0;

ℓ1: while(n >= 0)
    {
ℓ2:
        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }

ℓ5:
```



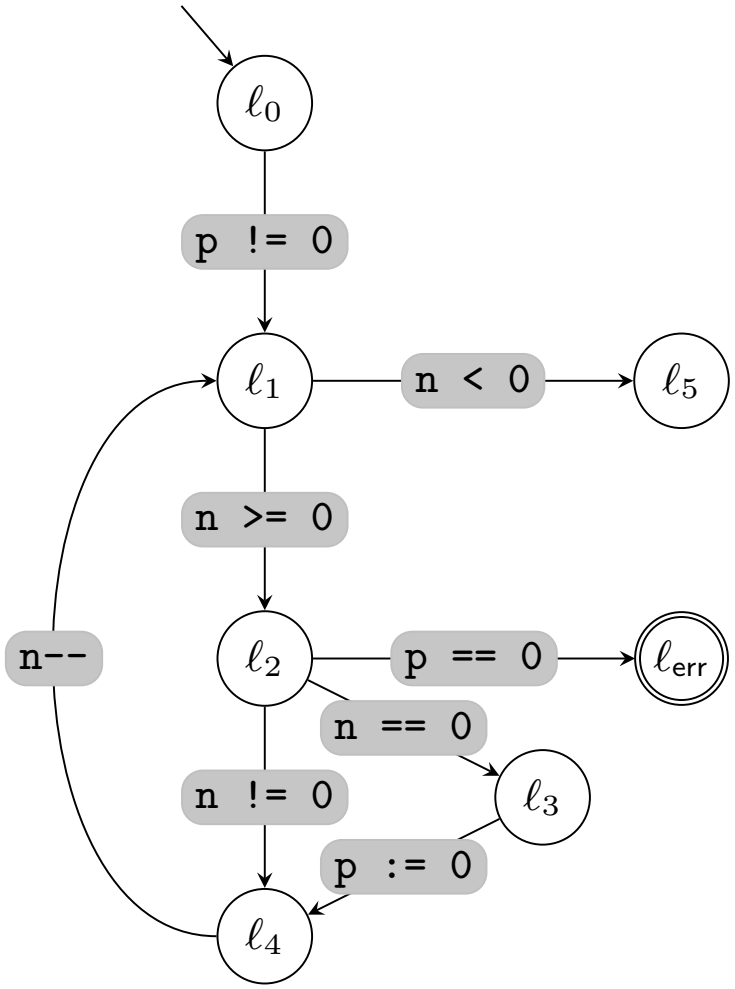
```
ℓ0: assume p != 0;

ℓ1: while(n >= 0)
    {
ℓ2:     assert p != 0;

        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }

ℓ5:
```



```
ℓ0: assume p != 0;

ℓ1: while(n >= 0)
    {
ℓ2:     assert p != 0;

        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }

ℓ5:
```

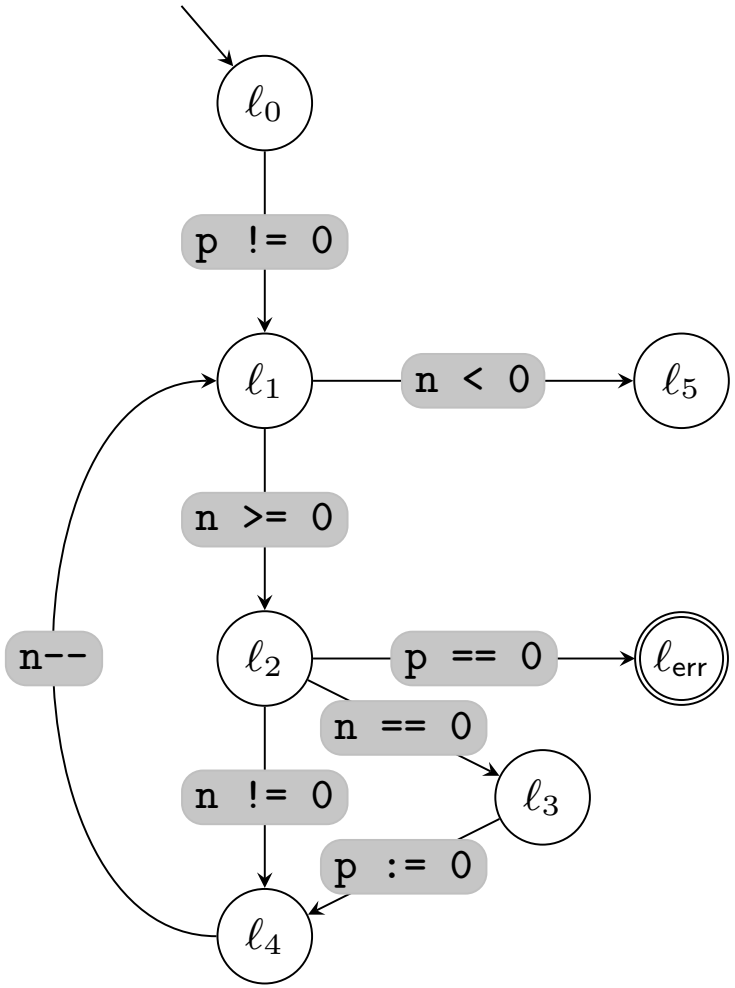


no execution violates assertion = no execution reaches error location



automaton

alphabet: {statements}




```
(p != 0)
(n >= 0)
(p == 0)
```

infeasible trace

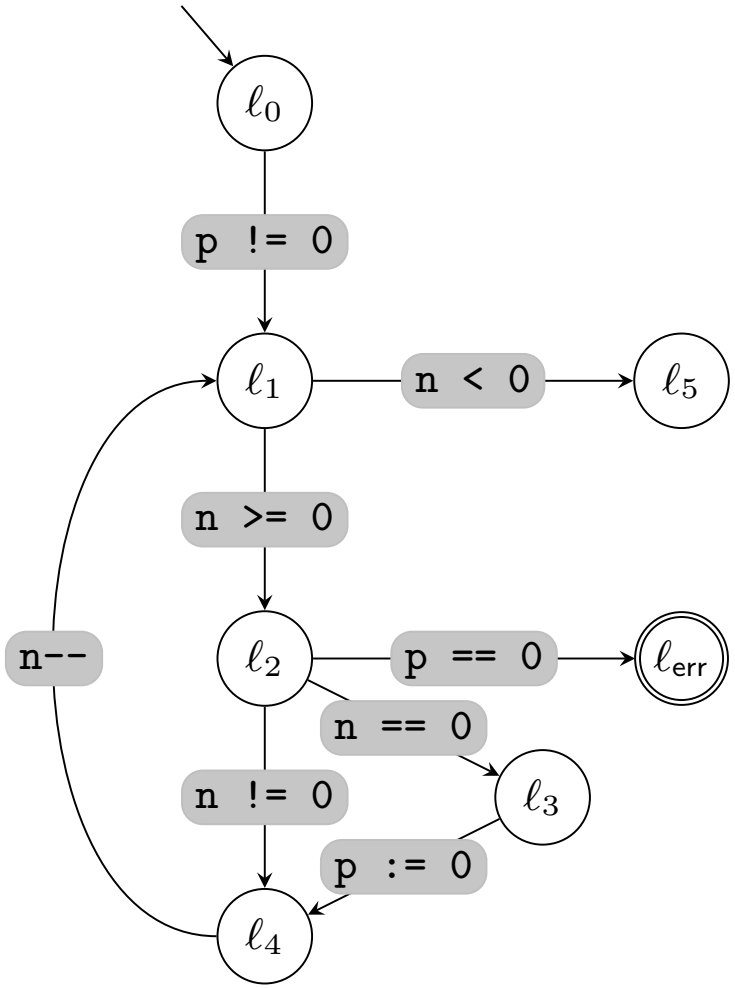
unsatisfiable formula

```
x == 1 ; x == -1 ;
```

$$x = 1 \ \wedge \ x = -1$$

```
x := 1 ; x == -1 ;
```

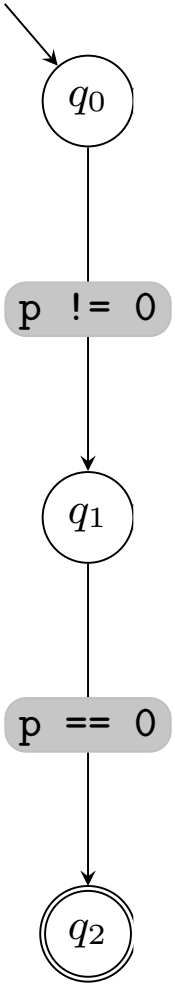
$$x' = 1 \ \wedge \ x' = -1$$




```
(p != 0)
(n >= 0)
(p == 0)
```

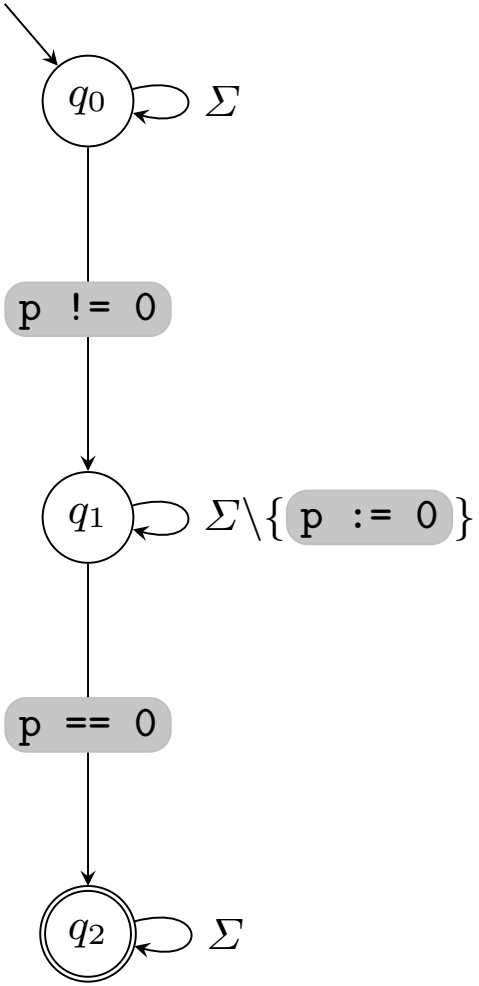
```
(p != 0)

(p==0)
```

```
(p != 0)

(p==0)
```



(p != 0)

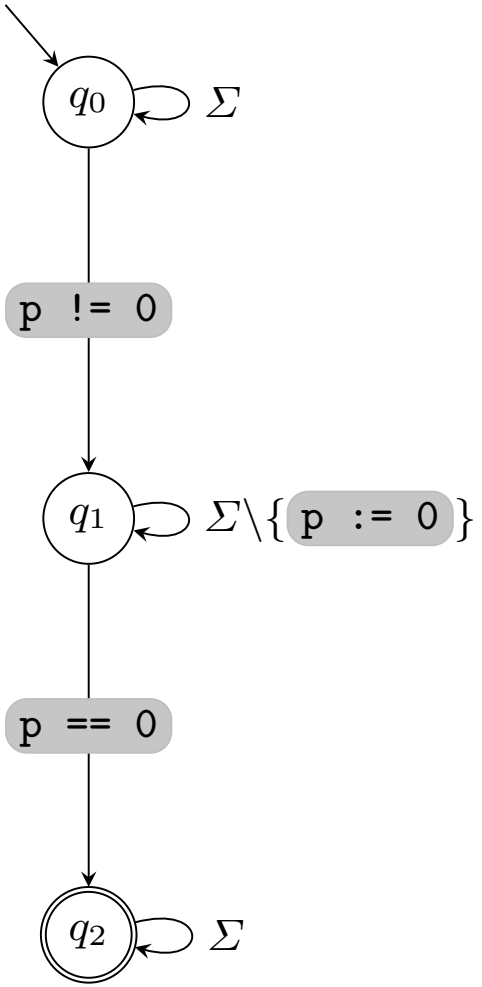
(p==0)



(p != 0)

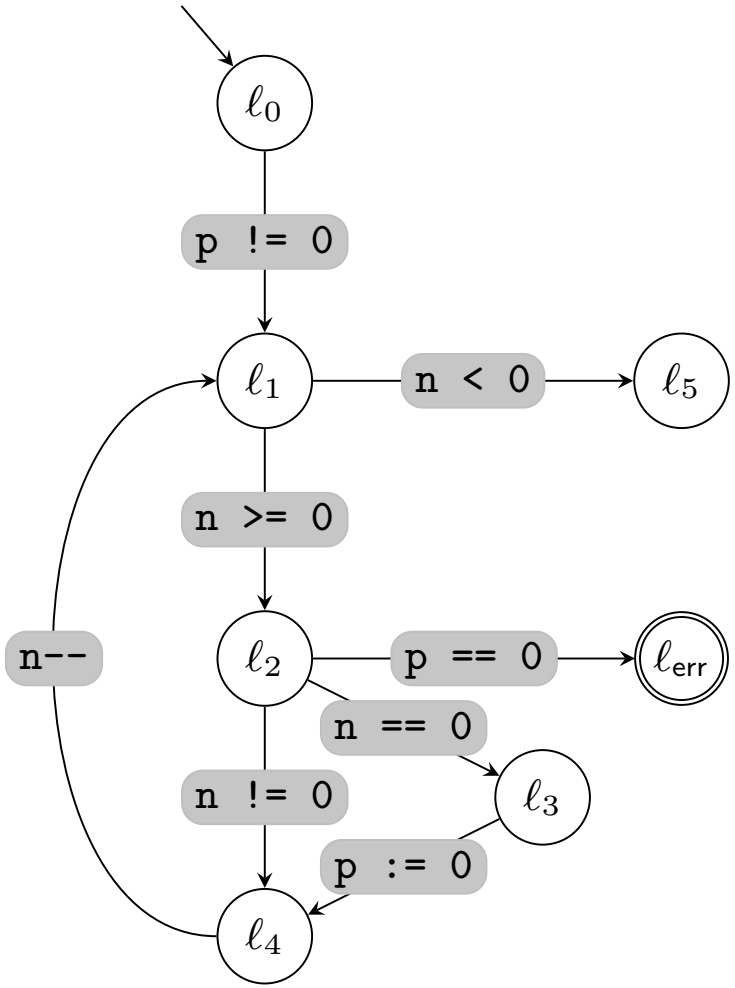
(p==0)

automaton constructed from unsatisfiability proof



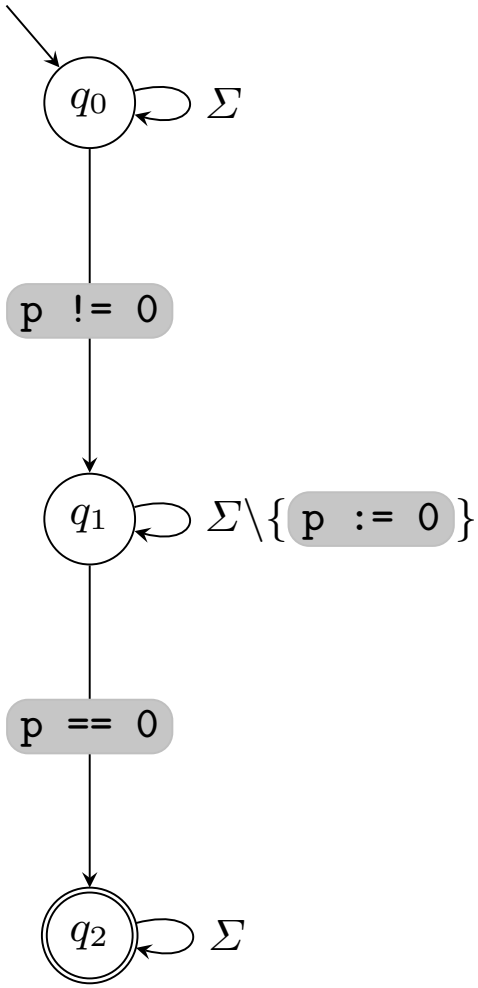

```
(p != 0)
(n >= 0)
(p == 0)
```

accepts all traces with the *same* unsatisfiability proof

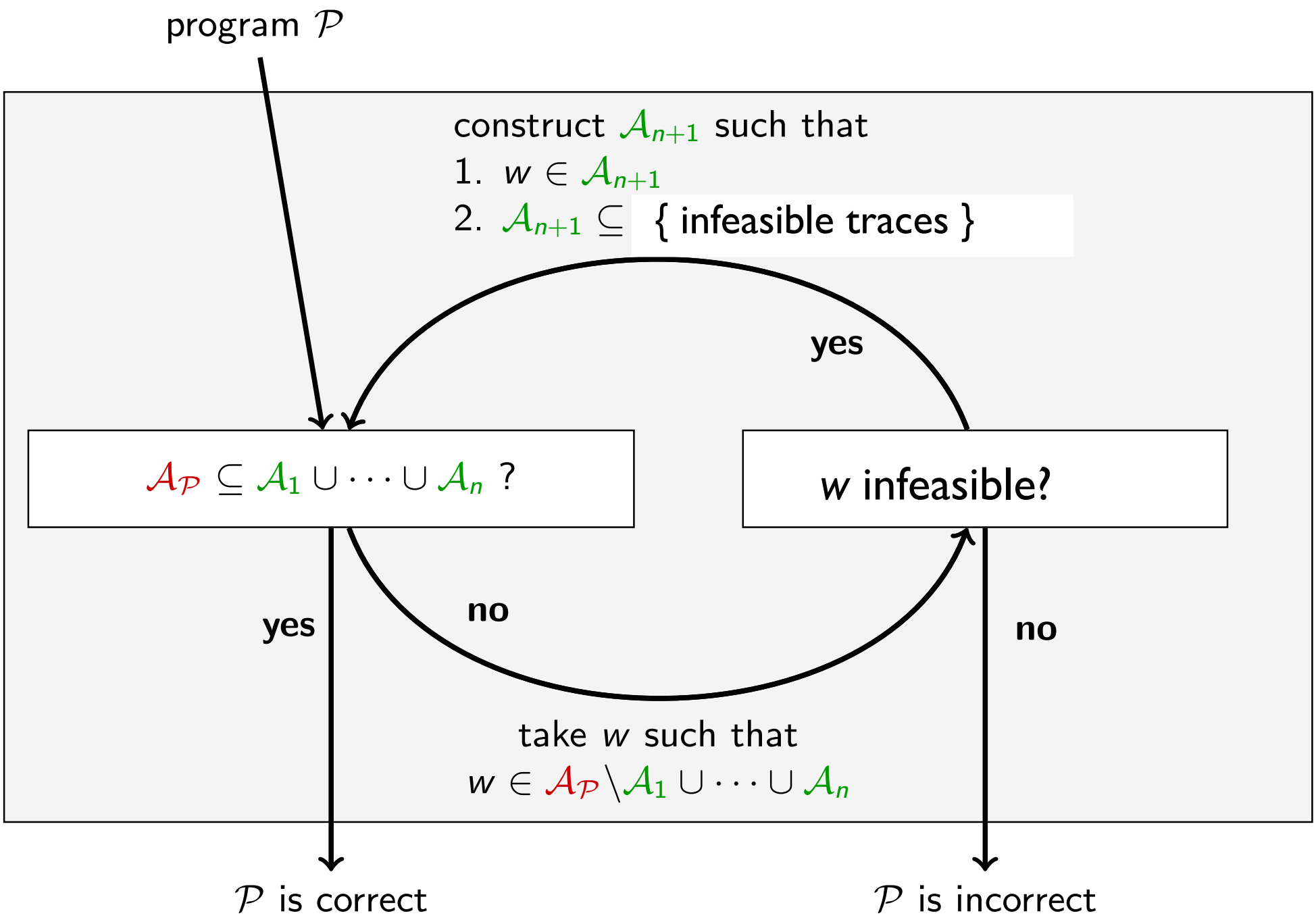


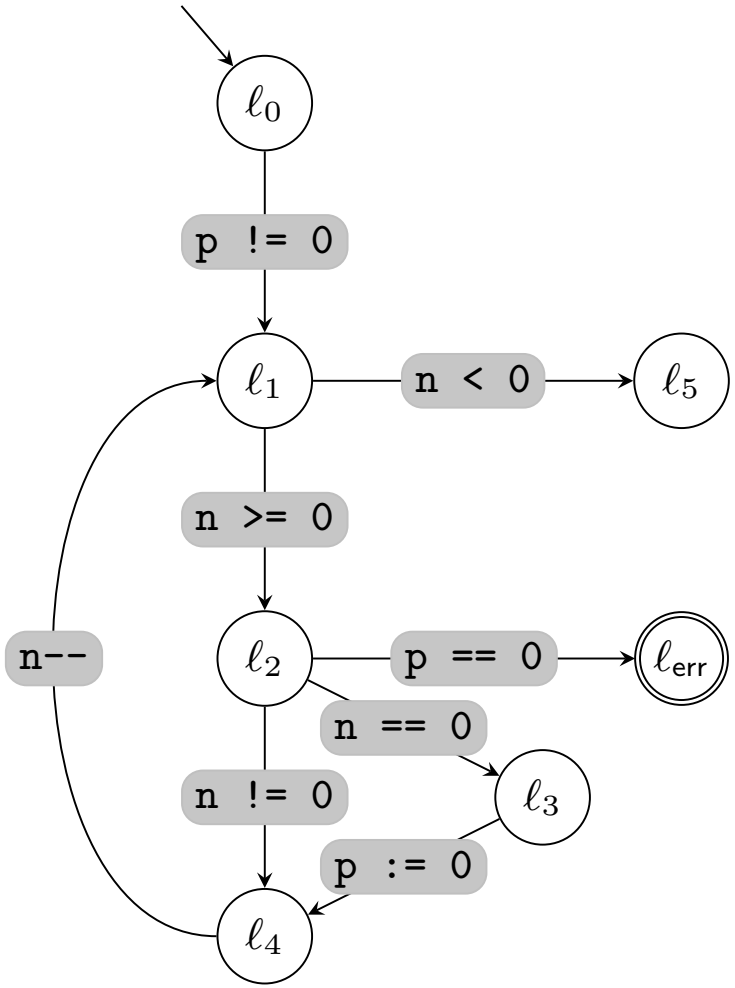
?

$\subseteq$



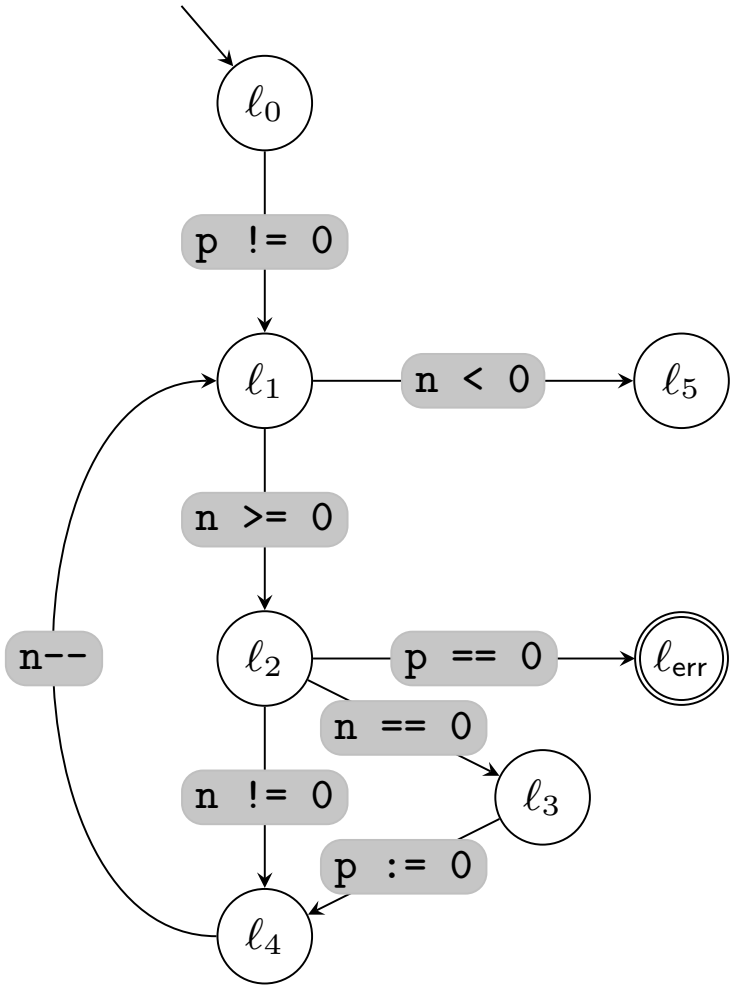
does a proof *exist* for every trace ?


program $\mathcal{P}$
construct $\mathcal{A}_{n+1}$ such that
1. $w \in \mathcal{A}_{n+1}$
2. $\mathcal{A}_{n+1} \subseteq$ { infeasible traces }
yes
$\mathcal{A}_\mathcal{P} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$?
w infeasible?
yes
no
no
take w such that
$w \in \mathcal{A}_\mathcal{P} \setminus \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect



new trace:

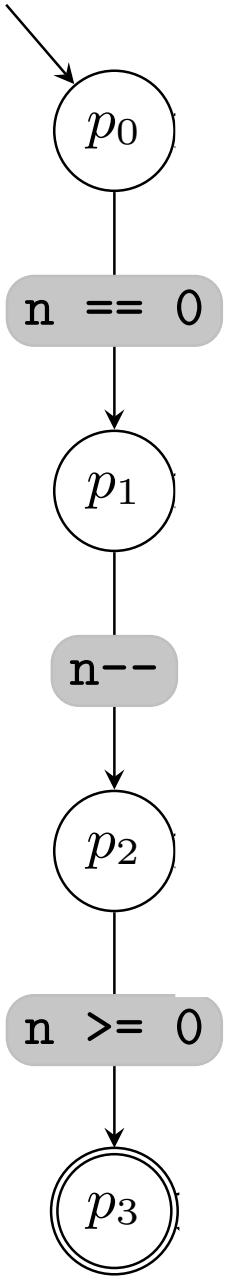
```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```




```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```

```
(n == 0)
(n--)
(n >= 0)
```

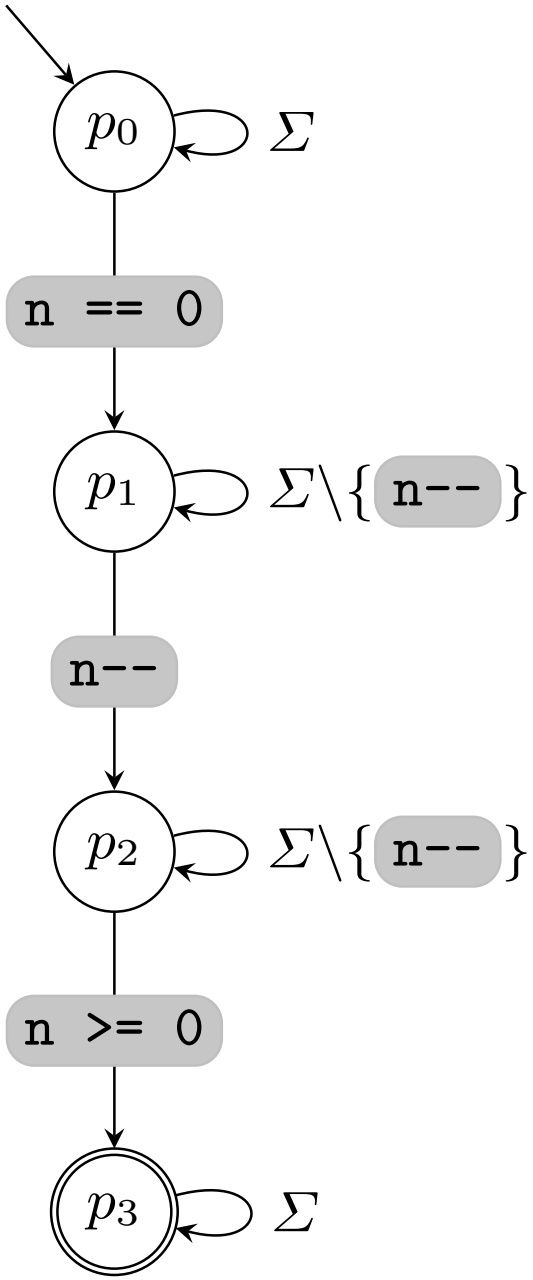
```
(n == 0)

(n--)
(n >= 0)
```



(n == 0)

(n--)

(n >= 0)

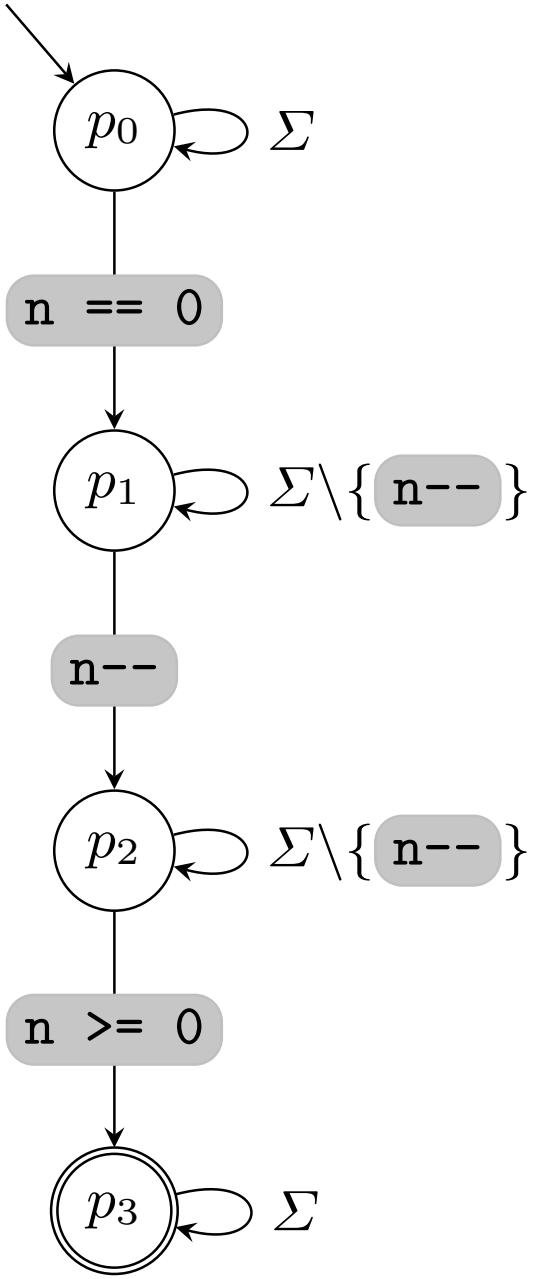

p_0
Σ
n == 0
p_1
$\Sigma\backslash\{$ n-- $\}$
n--
p_2
$\Sigma\backslash\{$ n-- $\}$
n >= 0
p_3
Σ

(n == 0)

(n--)

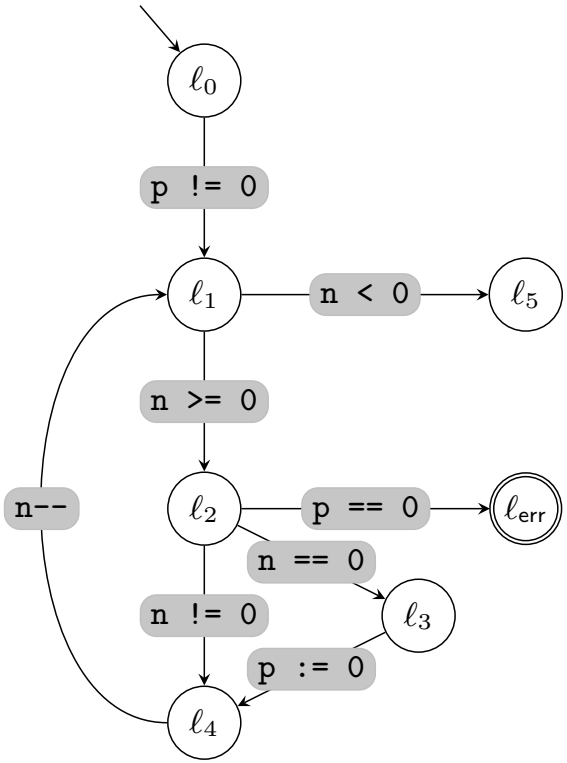
(n >= 0)

automaton constructed from unsatisfiability proof



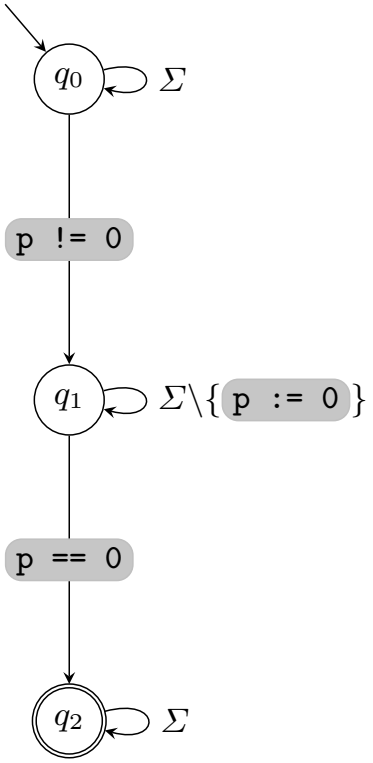

```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```

accepts all traces with the *same* unsatisfiability proof

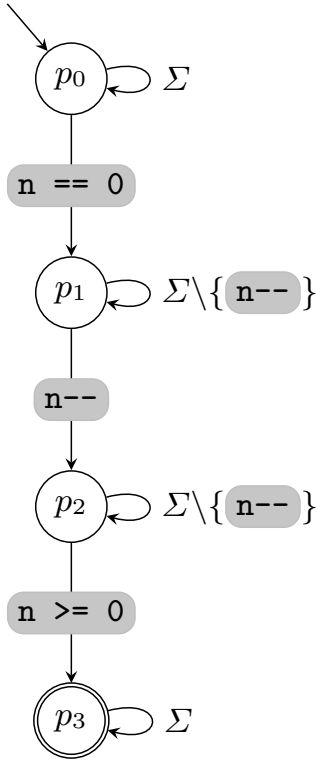


?

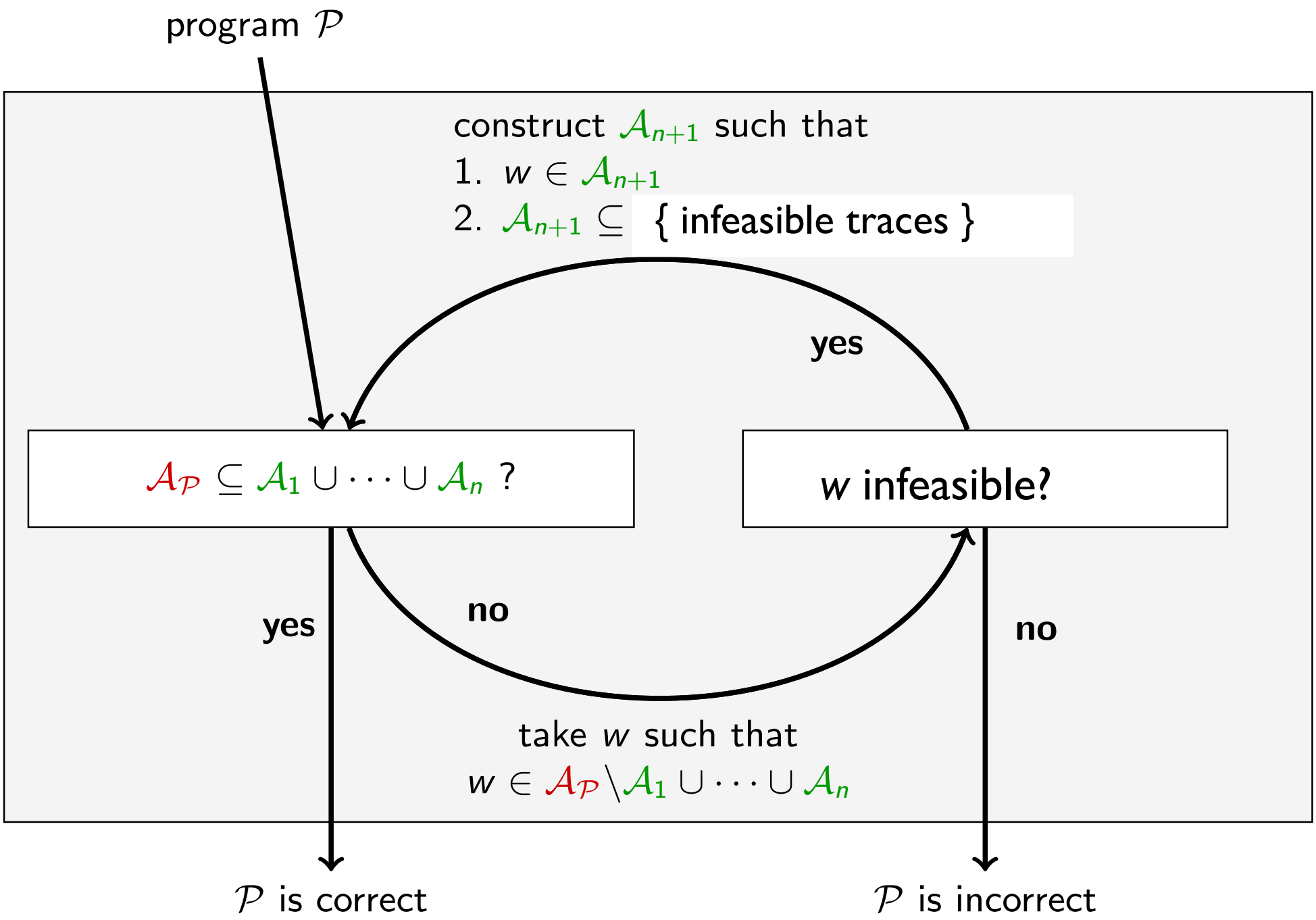
$\subseteq$



$\cup$



does a proof *exist* for every trace ?

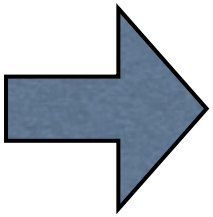

program $\mathcal{P}$
construct $\mathcal{A}_{n+1}$ such that
1. $w \in \mathcal{A}_{n+1}$
2. $\mathcal{A}_{n+1} \subseteq$ { infeasible traces }
yes
$\mathcal{A}_\mathcal{P} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$?
w infeasible?
yes
no
no
take w such that
$w \in \mathcal{A}_\mathcal{P} \setminus \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

automata constructed from unsatisfiable core

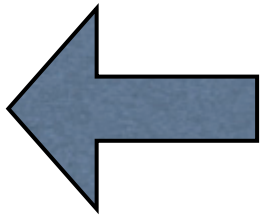
are not sufficient in general

(verification algorithm not complete)

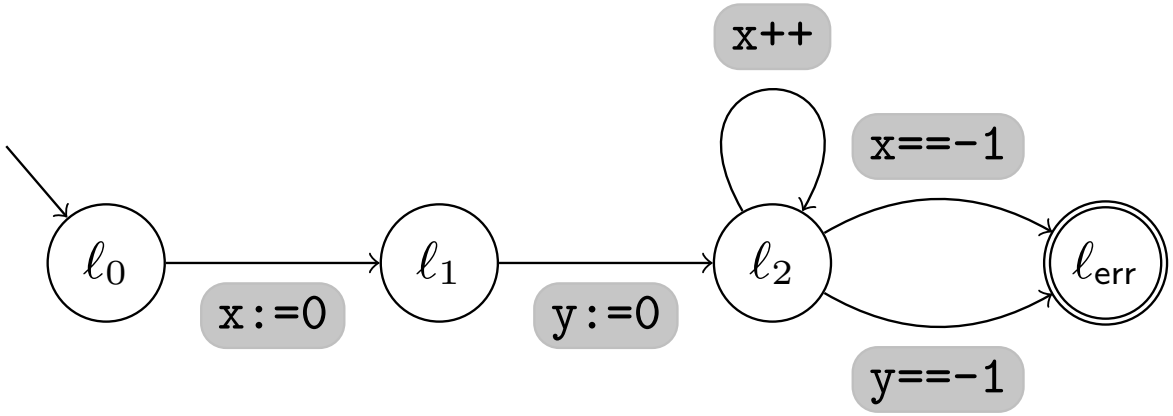
proof spaces

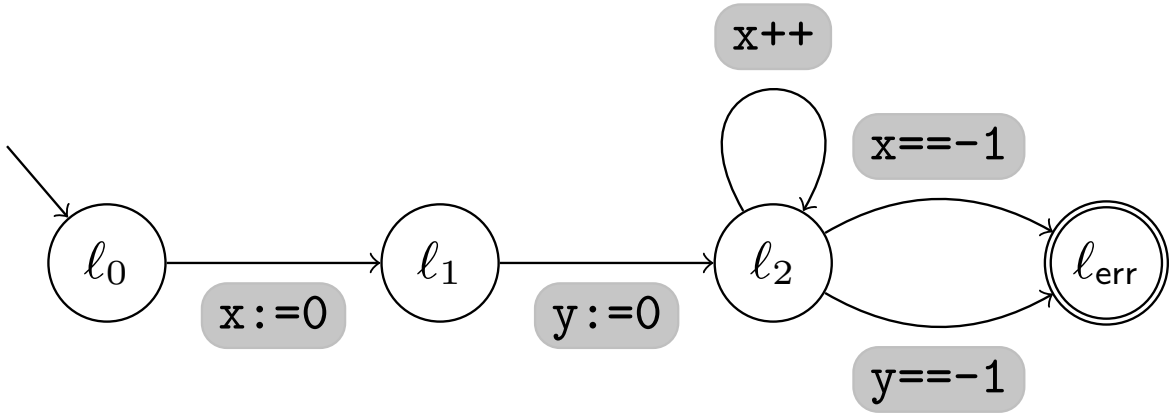


- automata from unsatisfiability proofs
- proof spaces



```
ℓ0: x := 0;
ℓ1: y := 0;
ℓ2: while(nondet) {x++;}
    assert(x != -1);
    assert(y != -1);
```





x:=0

y:=0

x++

x==-1

Hoare triples proving infeasibility :

$$
\begin{array}{lll}
\{\ true\ \} & \texttt{x:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{y:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x++} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x==-1} & \{\ false\ \}
\end{array}
$$

infeasibility $\Leftrightarrow$ pre/postcondition pair *(true, false)*

inference rule for sequencing

$$\frac{\{p\}\ \mathbf{s}\ \{q'\} \qquad \{q'\}\ \mathbf{s'}\ \{q\}}{\{p\}\ \mathbf{s}\ ;\ \mathbf{s'}\ \{q\}}$$

proof space

infinite space of Hoare triples "{*pre*} *trace* {*post*}"

closed under inference rule of sequencing

generated from finite basis of Hoare triples "{*pre*} *stmt* {*post*}"

proof of sample trace:

$$
\begin{array}{lll}
\{\ true\ \} & \texttt{x:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{y:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x++} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x==-1} & \{\ false\ \}
\end{array}
$$

finite basis of Hoare triples "{*pre*} *stmt* {*post*}"

can be obtained from proofs of sample traces

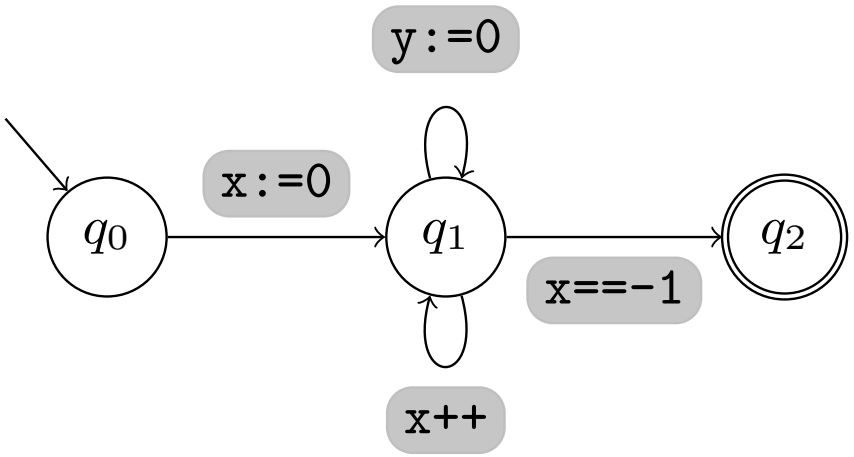
proof space

infinite space of Hoare triples "{*pre*} *trace* {*post*}"

closed under inference rule of sequencing

finite basis of Hoare triples "{*pre*} *stmt* {*post*}" $\longmapsto$ automaton

$$
\begin{array}{lll}
\{\ true\ \} & \texttt{x:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{y:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x++} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x==-1} & \{\ false\ \}
\end{array}
\quad \longmapsto \quad
$$



sequencing of Hoare triples in basis $\longmapsto$ run of automaton

proof space contains "{*true*} *trace* {*false*}"
if
exists sequencing of Hoare triples in basis
if
exists accepting run of automaton

proof space

infinite space of Hoare triples "{*pre*} *trace* {*post*}"

closed under inference rule of sequencing

generated from finite basis of Hoare triples "{*pre*} *stmt* {*post*}"

paradigm:

- construct proof space

- check proof space

inference rule for sequencing

$$\frac{\{p\}\ \mathbf{s}\ \{q'\} \qquad \{q'\}\ \mathbf{s'}\ \{q\}}{\{p\}\ \mathbf{s}\ \mathbf{;}\ \mathbf{s'}\ \{q\}}$$

inference rule for parallelism

$$\frac{\{p\}\ s\ \{q\} \qquad \{p'\}\ s\ \{q'\}}{\{p \wedge p'\}\ s\ \{q \wedge q'\}}$$

"interference freedom"

inference rule for unbounded number of threads

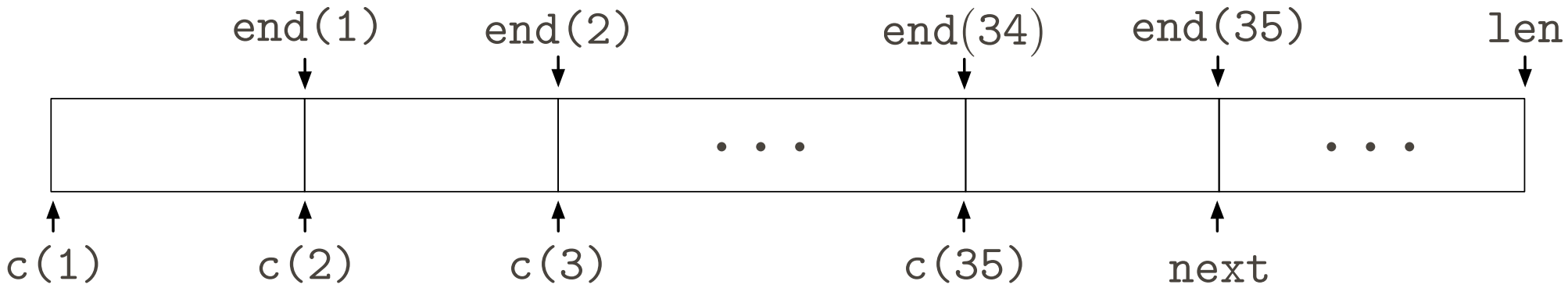
$$\frac{\{p\}\ \mathbf{s}\ \{q\}}{\{p'\}\ \mathbf{s'}\ \{q'\}}\ \text{renaming of thread id's}$$

"symmetry"

```
   global int len; //   length of array
   global int array(len) : tasks; //   array of tasks
   global int next; //   position of next available task block
   global lock m; //   lock protecting next

   thread T:
         local int : c;       //   position of current task
         local int : end; //   position of last task in acquired block
         //   acquire block of tasks
1        lock(m);
2              if(next + 10 <= len)
3                    { c := next; next := next + 10; end := next; }
4              else
5                    { c := next; next := next + 10; end := len; }
6        unlock(m);
         //   perform block of tasks
7        while (c < end):
8              tasks[c] := 0; //   mark task c as started
               ...                              //   work on the task c
9              tasks[c] := 1; //   mark task c as finished
10             assert(tasks[c] == 1); //   no other thread has started task c
11             c := c + 1;
```

```
   global int len; //   length of array
   global int array(len) : tasks; //   array of tasks
   global int next; //   position of next available task block
   global lock m; //   lock protecting next

   thread T:
         local int : c;      //   position of current task
         local int : end; //   position of last task in acquired block
         //   acquire block of tasks
1        lock(m);
2              if(next + 10 <= len)
3                    { c := next; next := next + 10; end := next; }
4              else
5                    { c := next; next := next + 10; end := len; }
6        unlock(m);
         //   perform block of tasks
7        while (c < end):
8              tasks[c] := 0; //   mark task c as started
               ...                              //   work on the task c
9              tasks[c] := 1; //   mark task c as finished
10             assert(tasks[c] == 1); //   no other thread has started task c
11             c := c + 1;
```

```
thread T:
        local int : c;      //  position of current task
        local int : end; //  position of last task in acquired block
        //  acquire block of tasks
1       lock(m);
2           if(next + 10 <= len)
3                   { c := next; next := next + 10; end := next; }
4           else
5                   { c := next; next := next + 10; end := len; }
6       unlock(m);
```



threads $1, 2, \ldots, 35$
have acquired block of tasks
have not yet started working

$\{\texttt{true}\}\ \pi\ \{\texttt{end}(2) \leq \texttt{c}(9)\}$

for given trace π
(fixed set of threads),
proof
can be
computed
automatically
by SMT solver

$\{true\}$
lock(m) :2
$\{true\}$
assume(next + 10 <= len) :2
$\{true\}$
c := next :2
$\{true\}$
next := next + 10 :2
$\{true\}$
end := next :2
$\{\texttt{end}(2) \leq \texttt{next}\}$
unlock(m); :2
$\{\texttt{end}(2) \leq \texttt{next}\}$
lock(m) :9
$\{\texttt{end}(2) \leq \texttt{next}\}$
assume(next + 10 <= len) :9
$\{\texttt{end}(2) \leq \texttt{next}\}$
c := next :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
next := next + 10 :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
end := next :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
unlock(m) :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$

$\{\texttt{true}\}\ \pi\ \{\texttt{end}(2) \leq \texttt{c}(9)\}$

for given trace π
(fixed set of threads),
proof
can be
computed
automatically
by SMT solver

```
{true}
lock(m)                          :2
{true}
assume(next + 10 <= len)         :2
{true}
c := next                        :2
{true}
next := next + 10                :2
{true}
end := next                      :2
{end(2) ≤ next}
unlock(m);                       :2
{end(2) ≤ next}
lock(m)                          :9
{end(2) ≤ next}
assume(next + 10 <= len)         :9
{end(2) ≤ next}
c := next                        :9
{end(2) ≤ c(9)}
next := next + 10                :9
{end(2) ≤ c(9)}
end := next                      :9
{end(2) ≤ c(9)}
unlock(m)                        :9
{end(2) ≤ c(9)}
```

$\{\texttt{true}\}\ \pi\ \{\texttt{end}(2) \leq \texttt{c}(9)\}$

for given trace π
(fixed set of threads),
proof
can be
assembled
automatically
from a basis of
atomic Hoare triples

$\{true\}$
lock(m) :2
$\{true\}$
assume(next + 10 <= len) :2
$\{true\}$
c := next :2
$\{true\}$
next := next + 10 :2
$\{true\}$
end := next :2
$\{\texttt{end}(2) \leq \texttt{next}\}$
unlock(m); :2
$\{\texttt{end}(2) \leq \texttt{next}\}$
lock(m) :9
$\{\texttt{end}(2) \leq \texttt{next}\}$
assume(next + 10 <= len) :9
$\{\texttt{end}(2) \leq \texttt{next}\}$
c := next :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
next := next + 10 :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
end := next :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$
unlock(m) :9
$\{\texttt{end}(2) \leq \texttt{c}(9)\}$

basis for *Thread Pooling*

$\{true\}$	$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{true\}$
$\langle \texttt{c := next} : 1 \rangle$	$\langle \texttt{next := next + 10} : 1 \rangle$	$\langle \texttt{end := next} : 1 \rangle$
$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{\texttt{c}(1) < \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

$\{\texttt{end}(1) \leq \texttt{next}\}$	$\{true\}$	$\{\texttt{len} \leq \texttt{next}\}$
$\langle \texttt{c := next} : 2 \rangle$	$\langle \texttt{assume(next + 10 > len)} : 1 \rangle$	$\langle \texttt{end := len} : 1 \rangle$
$\{\texttt{end}(1) \leq \texttt{c}(2)\}$	$\{\texttt{len} \leq \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

inference by symmetry

```
{true}
lock(m)                                  :2
{true}
assume(next + 10 <= len)                 :2
{true}
c := next                                :2
{true}
next := next + 10                        :2
{true}
end := next                              :2
{end(2) ≤ next}
unlock(m);                               :2
{end(2) ≤ next}
```

basis for *Thread Pooling*

$\{true\}$	$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{true\}$
$\langle \texttt{c := next} : 1 \rangle$	$\langle \texttt{next := next + 10} : 1 \rangle$	$\langle \texttt{end := next} : 1 \rangle$
$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{\texttt{c}(1) < \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

$\{\texttt{end}(1) \leq \texttt{next}\}$	$\{true\}$	$\{\texttt{len} \leq \texttt{next}\}$
$\langle \texttt{c := next} : 2 \rangle$	$\langle \texttt{assume(next + 10 > len)} : 1 \rangle$	$\langle \texttt{end := len} : 1 \rangle$
$\{\texttt{end}(1) \leq \texttt{c}(2)\}$	$\{\texttt{len} \leq \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

inference by symmetry

```
{end(2) ≤ next}
lock(m)                              :9
{end(2) ≤ next}
assume(next + 10 <= len)             :9
{end(2) ≤ next}
c := next                            :9
{end(2) ≤ c(9)}
next := next + 10                    :9
{end(2) ≤ c(9)}
end := next                          :9
{end(2) ≤ c(9)}
unlock(m)                            :9
{end(2) ≤ c(9)}
```

("rename 2/1 and 9/2")

basis for *Thread Pooling*

$\{true\}$	$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{true\}$
$\langle \texttt{c := next} : 1 \rangle$	$\langle \texttt{next := next + 10} : 1 \rangle$	$\langle \texttt{end := next} : 1 \rangle$
$\{\texttt{c}(1) \leq \texttt{next}\}$	$\{\texttt{c}(1) < \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

$\{\texttt{end}(1) \leq \texttt{next}\}$	$\{true\}$	$\{\texttt{len} \leq \texttt{next}\}$
$\langle \texttt{c := next} : 2 \rangle$	$\langle \texttt{assume(next + 10 > len)} : 1 \rangle$	$\langle \texttt{end := len} : 1 \rangle$
$\{\texttt{end}(1) \leq \texttt{c}(2)\}$	$\{\texttt{len} \leq \texttt{next}\}$	$\{\texttt{end}(1) \leq \texttt{next}\}$

proof space

infinite space of Hoare triples "{*pre*} *trace* {*post*}"

closed under inference rules of sequencing,
conjunction, symmetry

generated from finite basis of Hoare triples "{*pre*} *stmt* {*post*}"

paradigm:

- construct proof space

- check proof space

simplify task for program verification:

Don’t give a proof.

Show that a proof *exists*.

automata:
existence of accepting run

inclusion check:
show that, for every word in the given set,
an accepting run *exists*

simplify task for program verification:

Show that,
for every program execution,
a proof exists.

Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Jochen Hoenicke, Azadeh Farzan, Zachary Kincaid, Markus Lindenmann, Betim Musa, Christian Schilling, Alexander Nutz, Stefan Wissert, Evren Ermis

- Refinement of Trace Abstraction. SAS 2009
- Nested interpolants. POPL 2010
- Interpolant Automata. ATVA 2012
- Ultimate Automizer with SMTInterpol - (Competition Contribution). TACAS 2013
- Automata as Proofs. VMCAI 2013
- Inductive data flow graphs. POPL 2013
- Software Model Checking for People Who Love Automata. CAV 2013
- Ultimate Automizer with Unsatisfiable Cores - (Competition Contribution). TACAS 2014
- Termination Analysis by Learning Terminating Programs. CAV 2014
- Proofs that count. POPL 2014:
- Ultimate Automizer with Array Interpolation - (Competition Contribution). TACAS 2015
- Automated Program Verification. LATA 2015
- Fairness Modulo Theory: A New Approach to LTL Software Model Checking. CAV 2015
- Proof Spaces for Unbounded Parallelism. POPL 2015