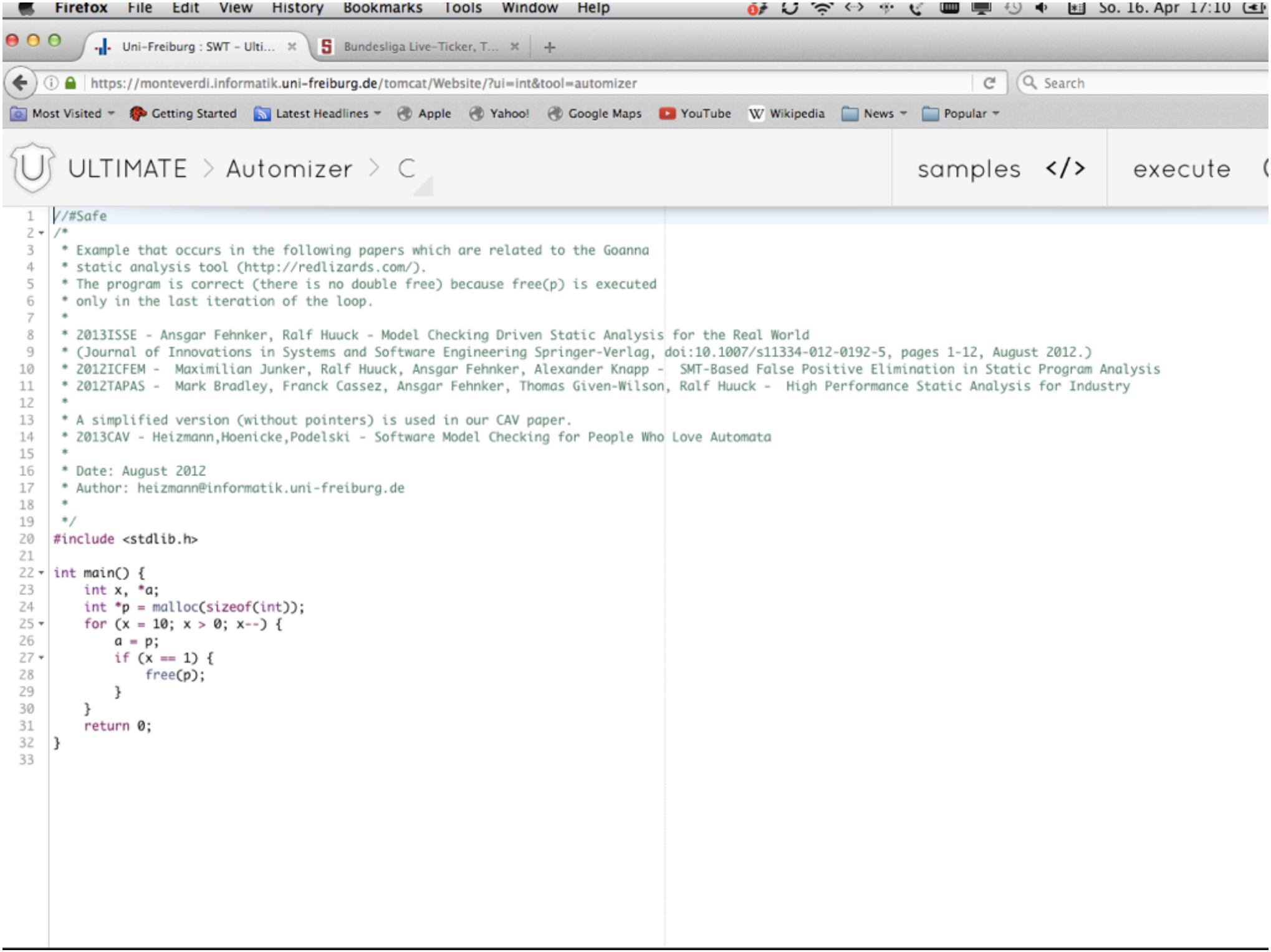
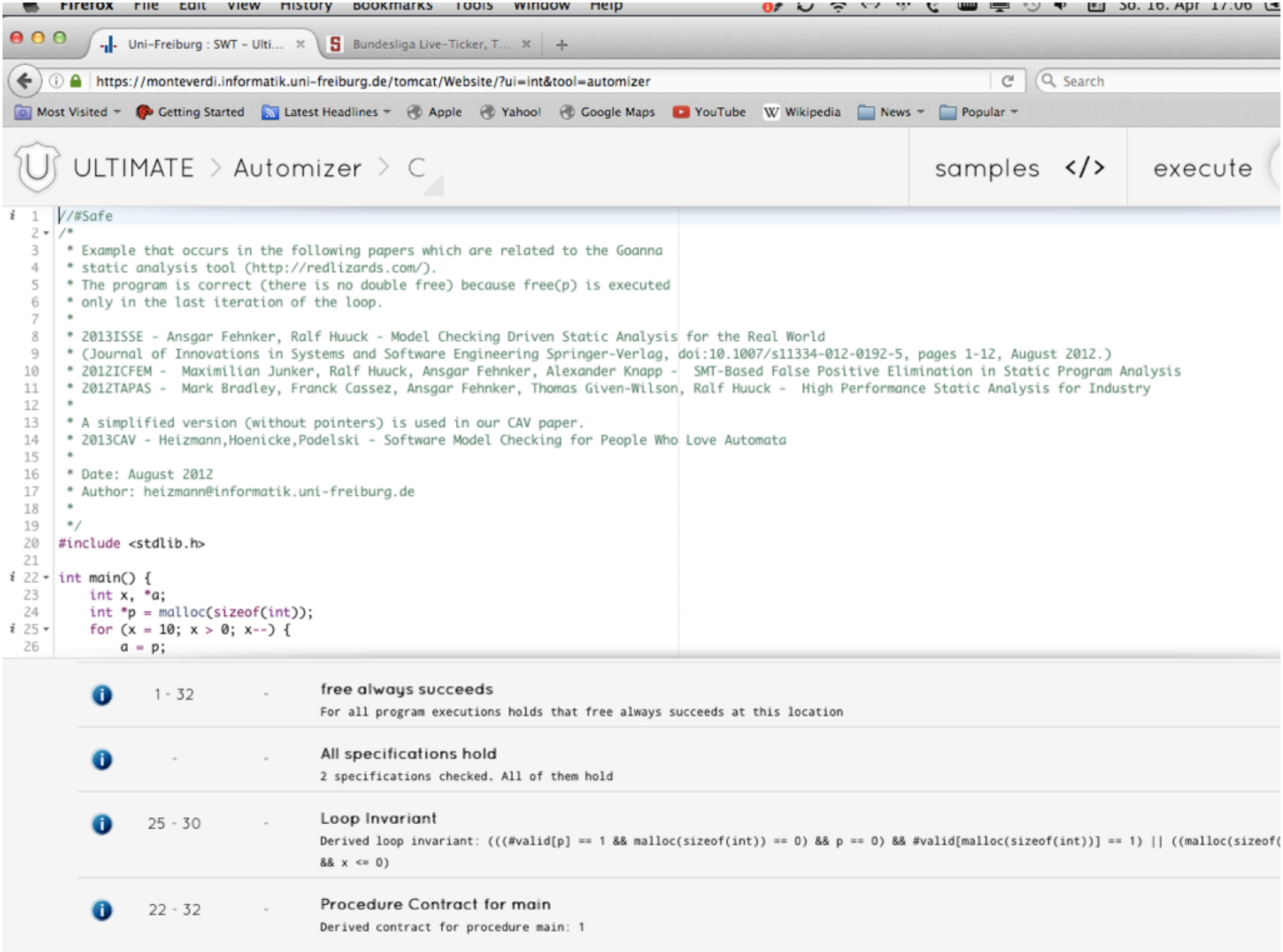
Software Model Checking with Automizer

(Sequential Programs)

Andreas Podelski

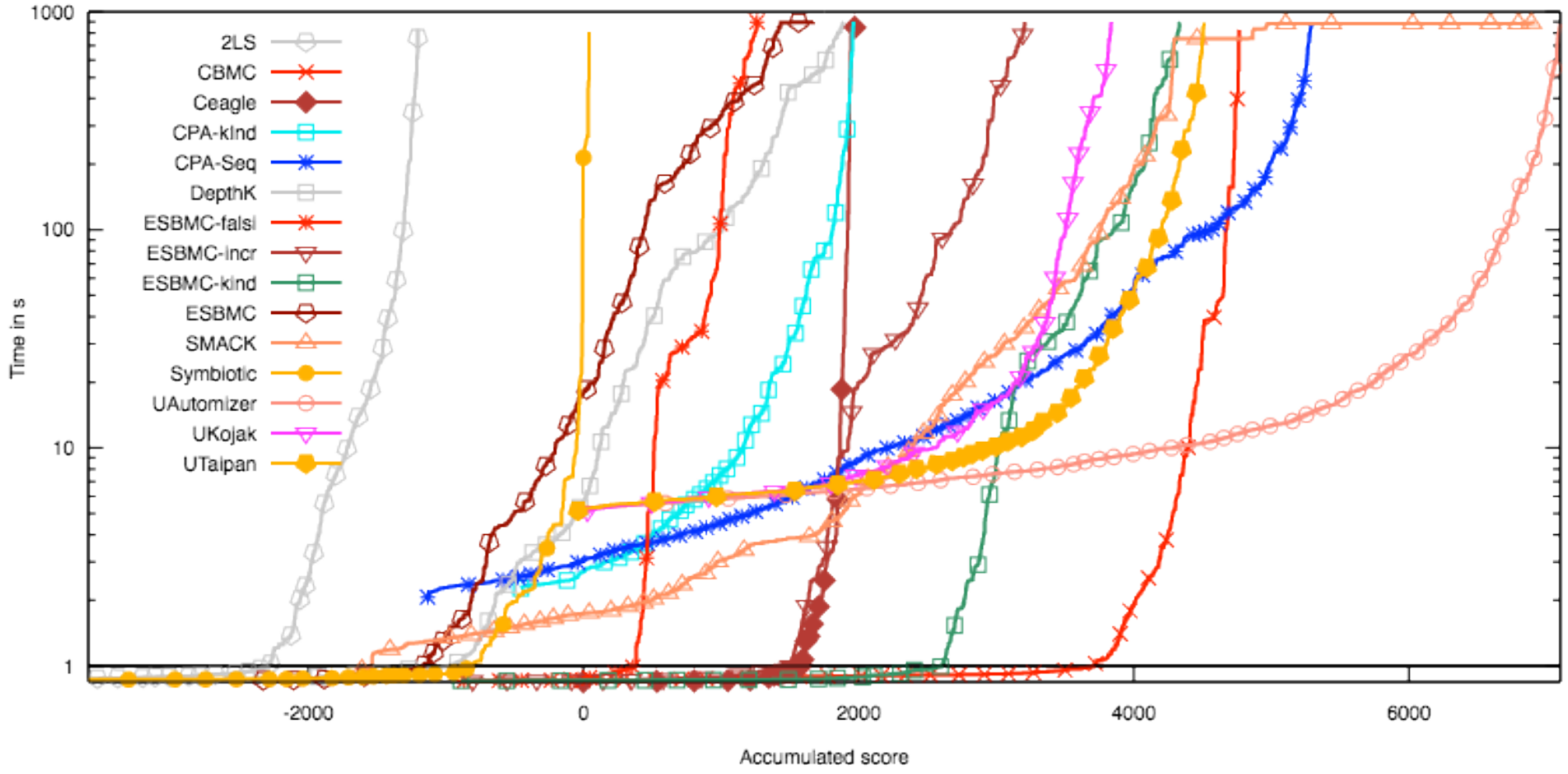
University of Freiburg
Germany

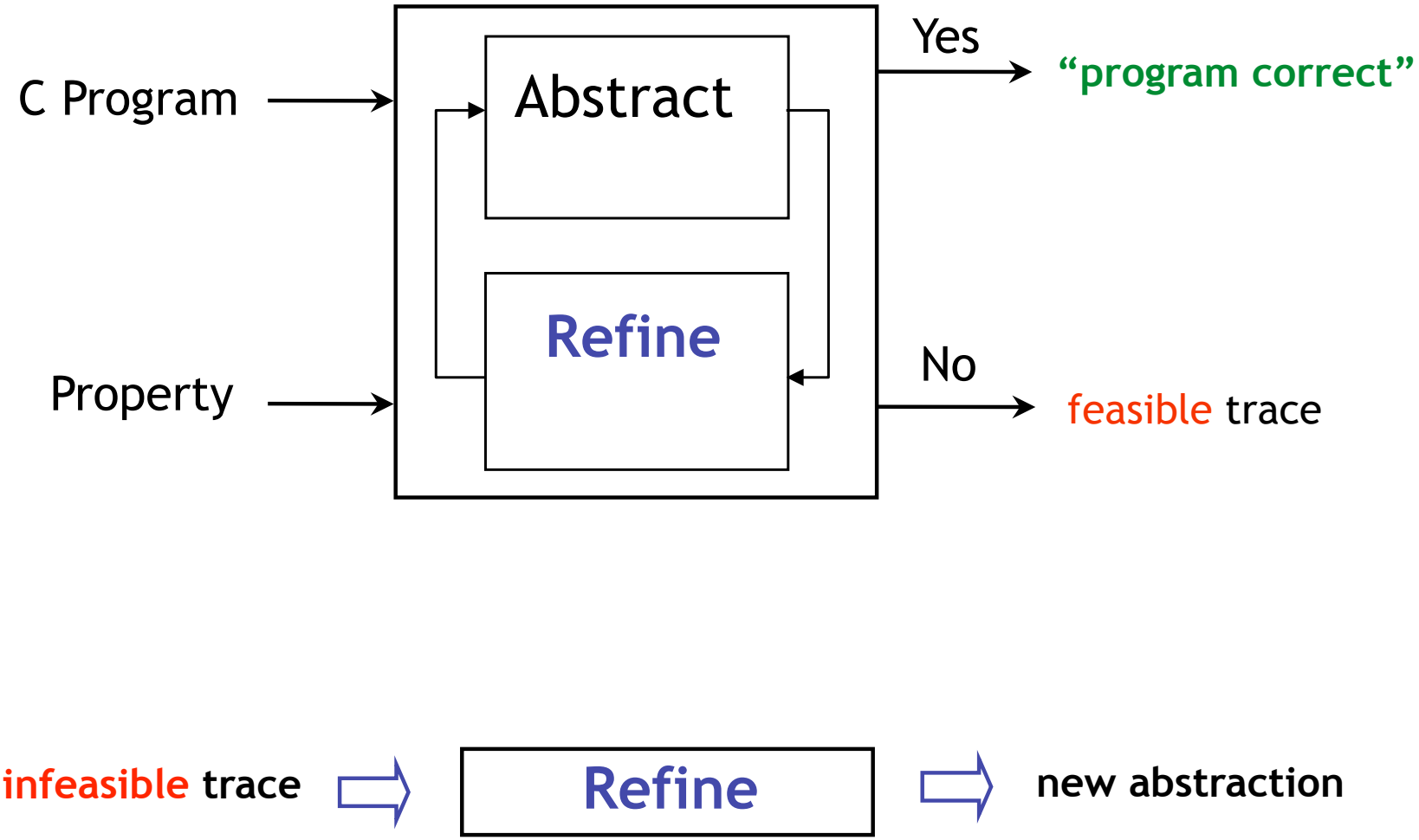

Firefox File Edit View History Bookmarks Tools Window Help
So. 16. Apr 17:10
Uni-Freiburg : SWT - Ulti...
Bundesliga Live-Ticker, T...
https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automizer
Search
Most Visited
Getting Started
Latest Headlines
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
ULTIMATE > Automizer > C
samples </>
execute
1 //#Safe
2 /*
3 * Example that occurs in the following papers which are related to the Goanna
4 * static analysis tool (http://redlizards.com/).
5 * The program is correct (there is no double free) because free(p) is executed
6 * only in the last iteration of the loop.
7 *
8 * 2013ISSE - Ansgar Fehnker, Ralf Huuck - Model Checking Driven Static Analysis for the Real World
9 * (Journal of Innovations in Systems and Software Engineering Springer-Verlag, doi:10.1007/s11334-012-0192-5, pages 1-12, August 2012.)
10 * 2012ICFEM - Maximilian Junker, Ralf Huuck, Ansgar Fehnker, Alexander Knapp - SMT-Based False Positive Elimination in Static Program Analysis
11 * 2012TAPAS - Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, Ralf Huuck - High Performance Static Analysis for Industry
12 *
13 * A simplified version (without pointers) is used in our CAV paper.
14 * 2013CAV - Heizmann,Hoenicke,Podelski - Software Model Checking for People Who Love Automata
15 *
16 * Date: August 2012
17 * Author: heizmann@informatik.uni-freiburg.de
18 *
19 */
20 #include <stdlib.h>
21
22 int main() {
23 int x, *a;
24 int *p = malloc(sizeof(int));
25 for (x = 10; x > 0; x--) {
26 a = p;
27 if (x == 1) {
28 free(p);
29 }
30 }
31 return 0;
32 }
33

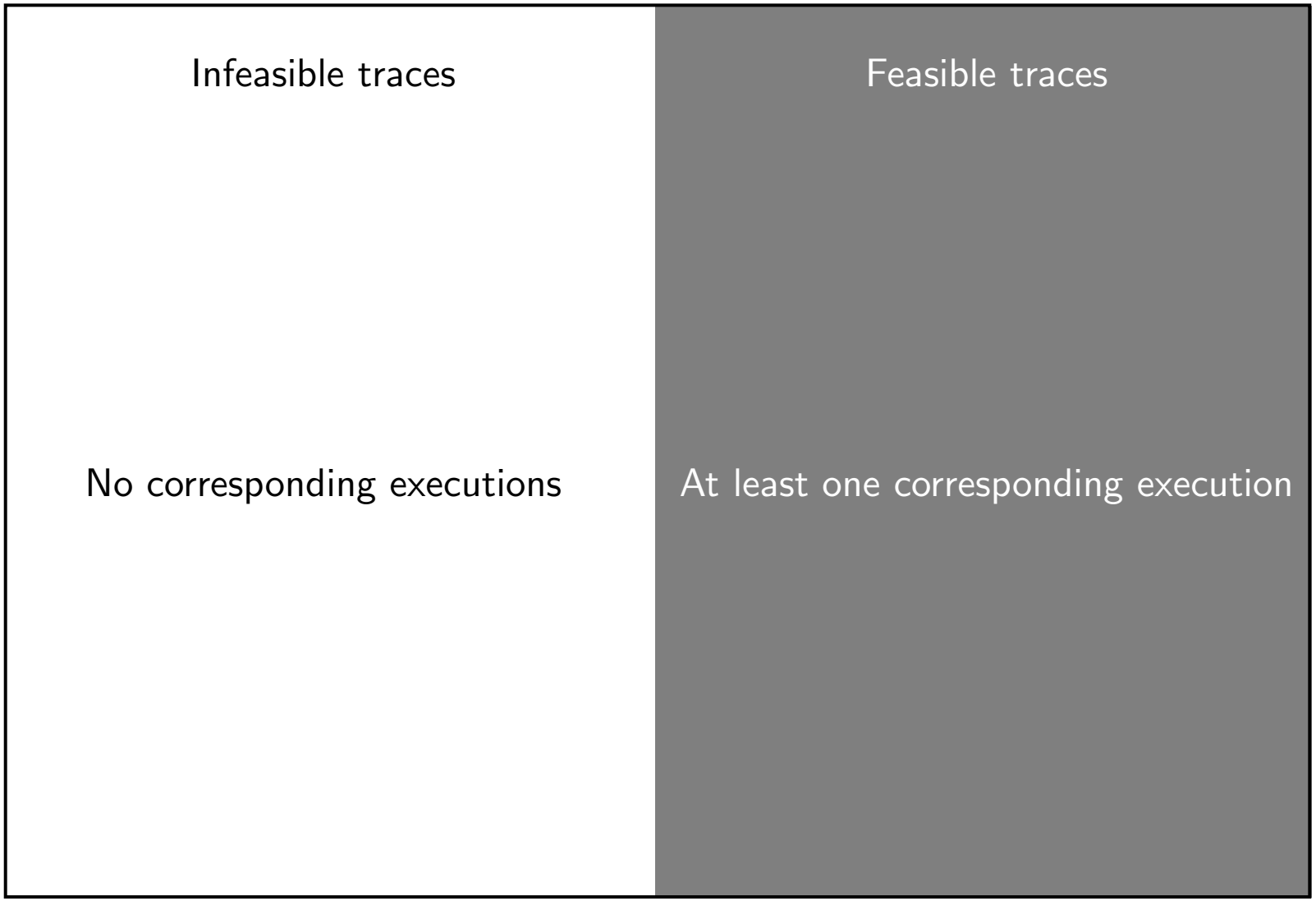

Firefox File Edit View History Bookmarks Tools Window Help
Uni-Freiburg : SWT - Ulti...
Bundesliga Live-Ticker, T...
https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automizer
Search
Most Visited
Getting Started
Latest Headlines
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
ULTIMATE > Automizer > C
samples </>
execute
1 //#Safe
2 /*
3 * Example that occurs in the following papers which are related to the Goanna
4 * static analysis tool (http://redlizards.com/).
5 * The program is correct (there is no double free) because free(p) is executed
6 * only in the last iteration of the loop.
7 *
8 * 2013ISSE - Ansgar Fehnker, Ralf Huuck - Model Checking Driven Static Analysis for the Real World
9 * (Journal of Innovations in Systems and Software Engineering Springer-Verlag, doi:10.1007/s11334-012-0192-5, pages 1-12, August 2012.)
10 * 2012ICFEM - Maximilian Junker, Ralf Huuck, Ansgar Fehnker, Alexander Knapp - SMT-Based False Positive Elimination in Static Program Analysis
11 * 2012TAPAS - Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, Ralf Huuck - High Performance Static Analysis for Industry
12 *
13 * A simplified version (without pointers) is used in our CAV paper.
14 * 2013CAV - Heizmann,Hoenicke,Podelski - Software Model Checking for People Who Love Automata
15 *
16 * Date: August 2012
17 * Author: heizmann@informatik.uni-freiburg.de
18 *
19 */
20 #include <stdlib.h>
21
22 int main() {
23 int x, *a;
24 int *p = malloc(sizeof(int));
25 for (x = 10; x > 0; x--) {
26 a = p;
1 - 32 - free always succeeds
For all program executions holds that free always succeeds at this location
- - All specifications hold
2 specifications checked. All of them hold
25 - 30 - Loop Invariant
Derived loop invariant: (((#valid[p] == 1 && malloc(sizeof(int)) == 0) && p == 0) && #valid[malloc(sizeof(int))] == 1) || ((malloc(sizeof(
&& x <= 0)
22 - 32 - Procedure Contract for main
Derived contract for procedure main: 1

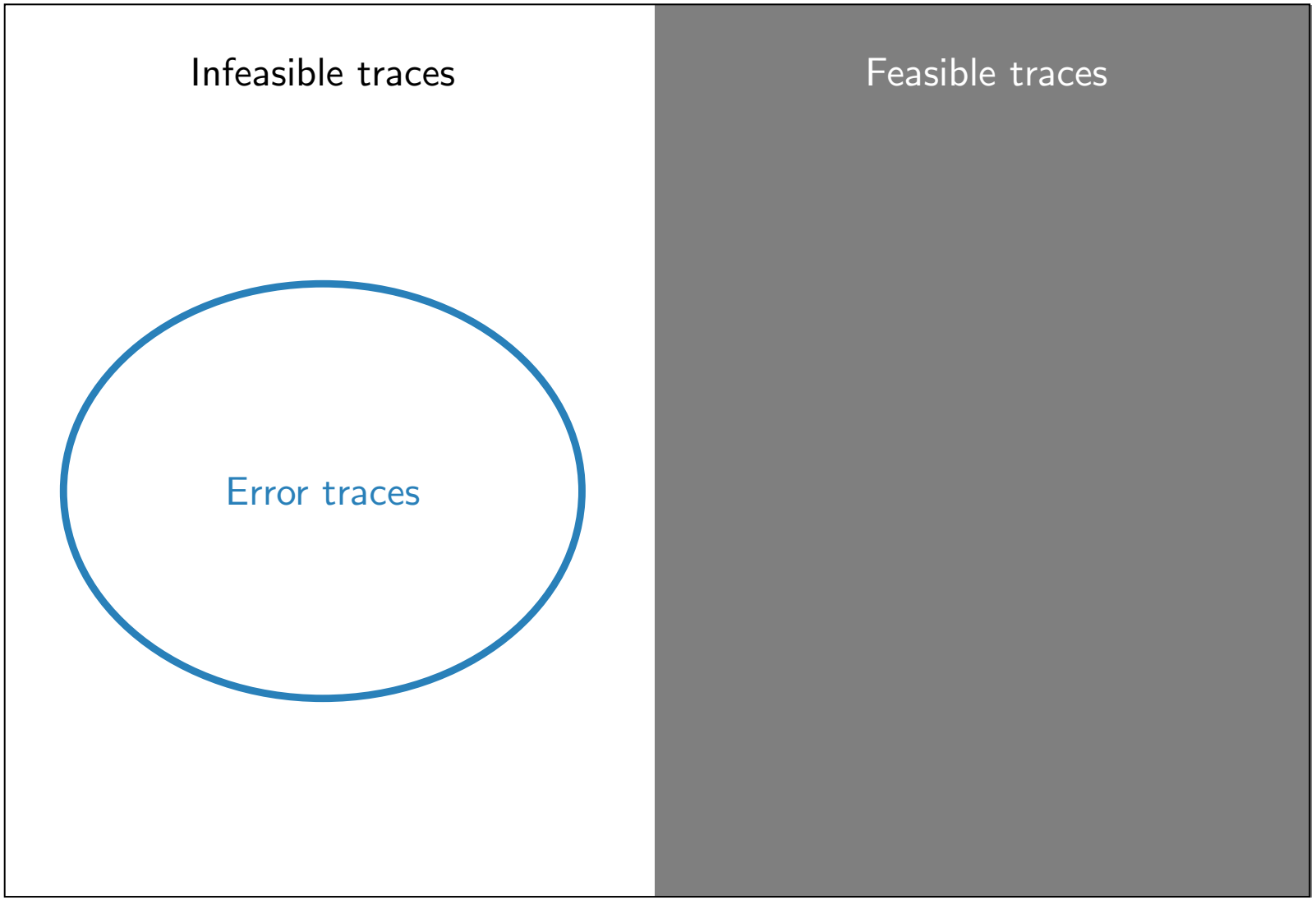
6th Competition on Software Verification (SV-COMP) 2017

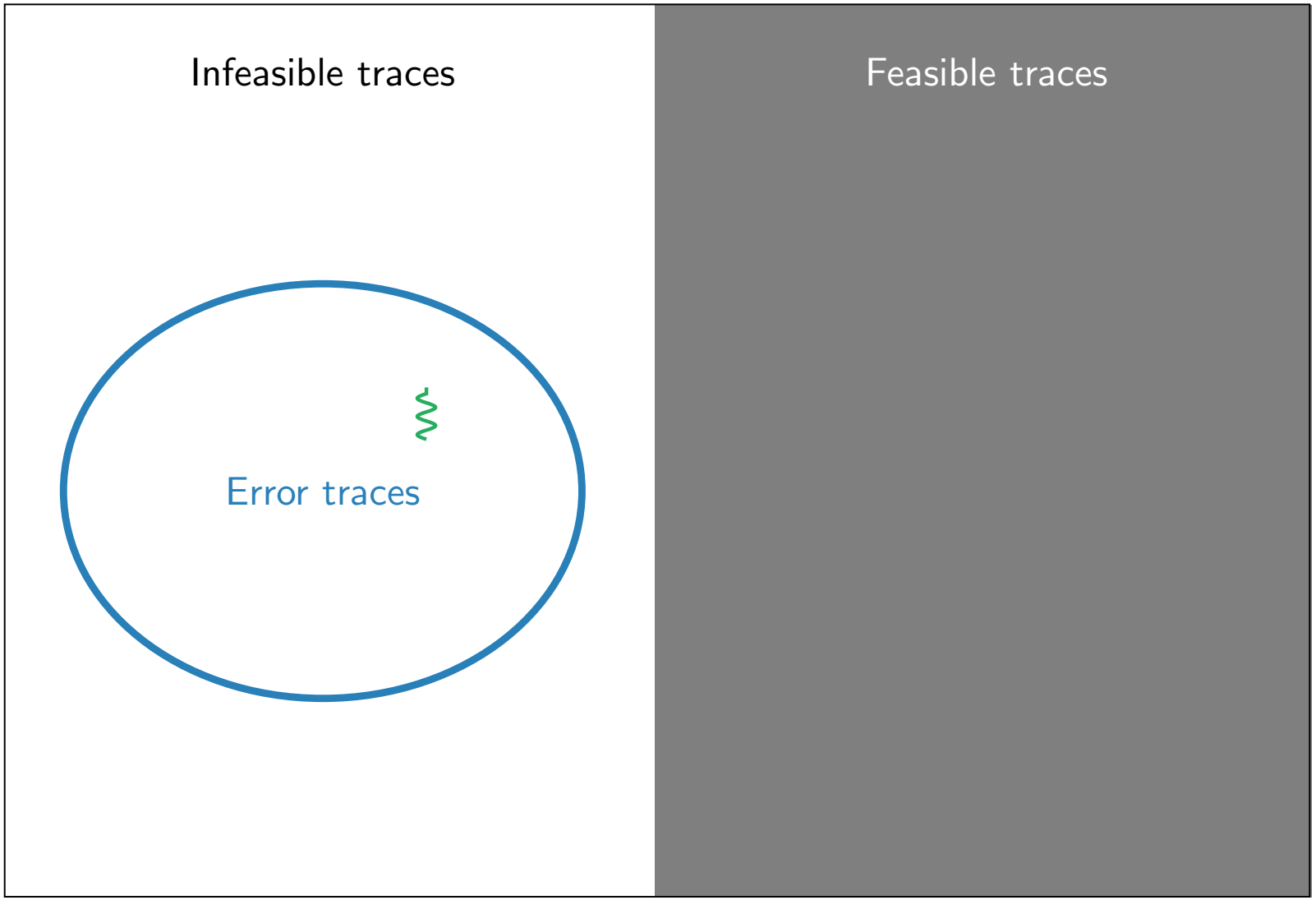
Overall
1. UAutomizer
2. SMACK
3. CPA-Seq

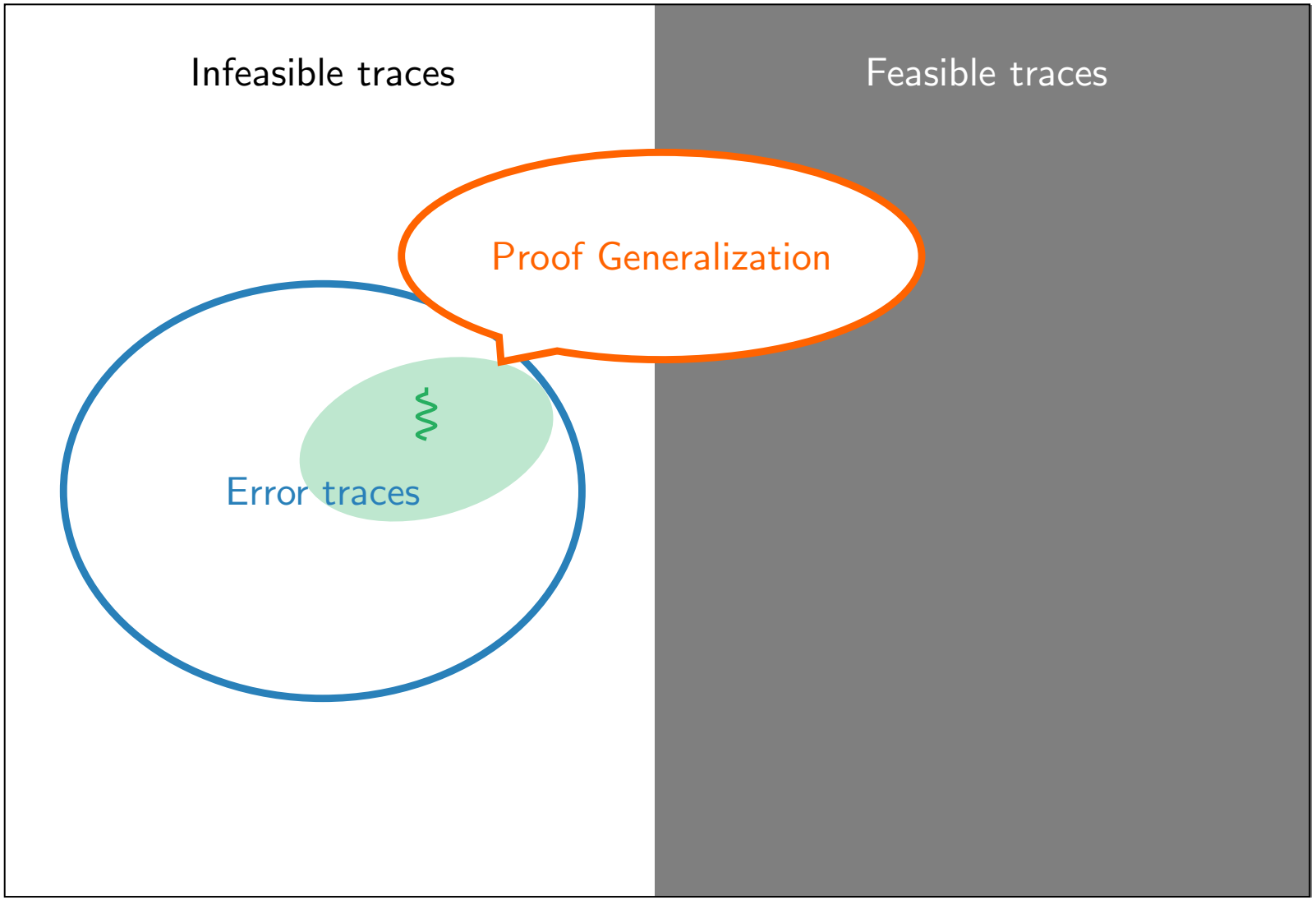


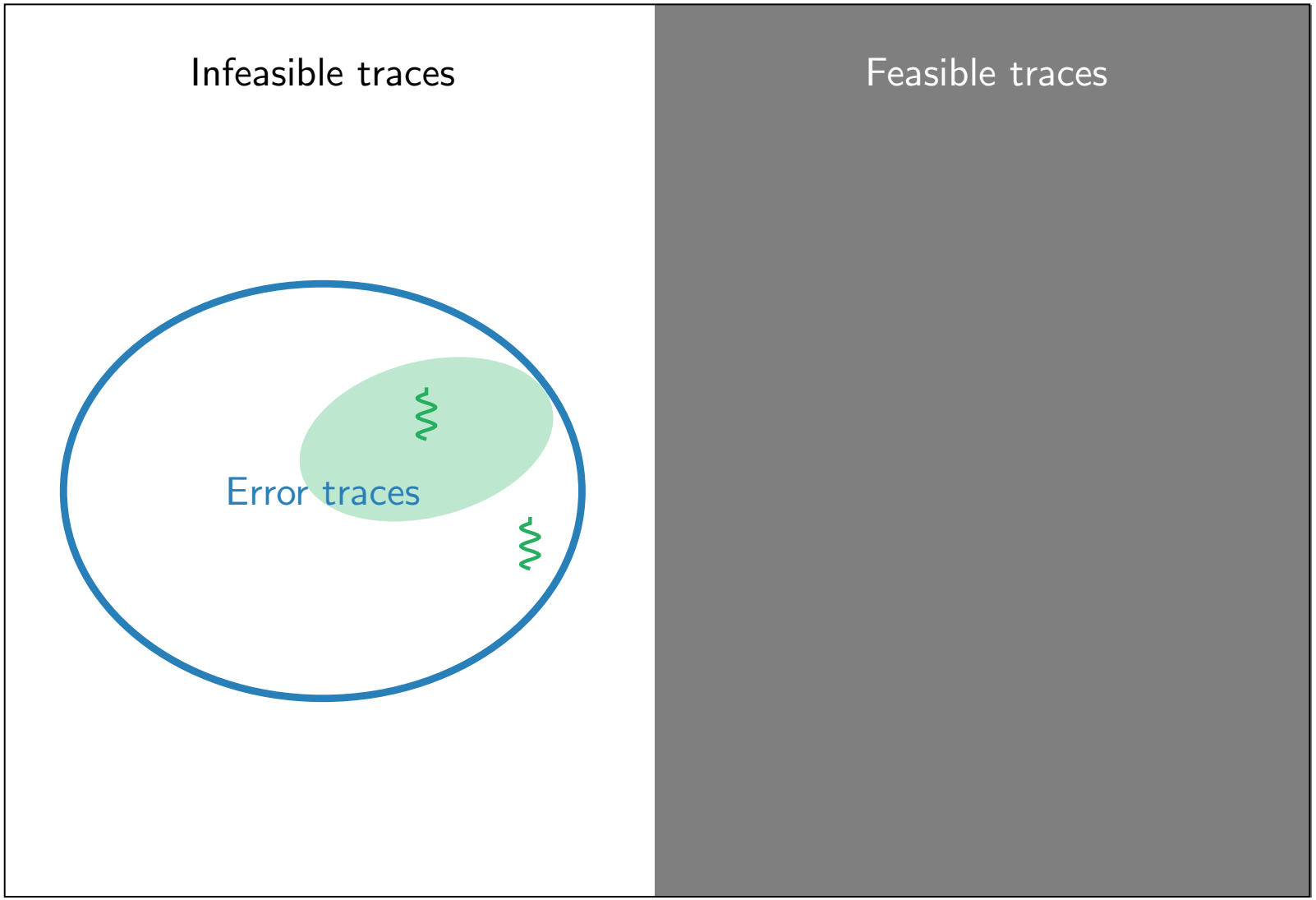

C Program
Property
Abstract
Refine
Yes
"program correct"
No
feasible trace
infeasible trace
Refine
new abstraction

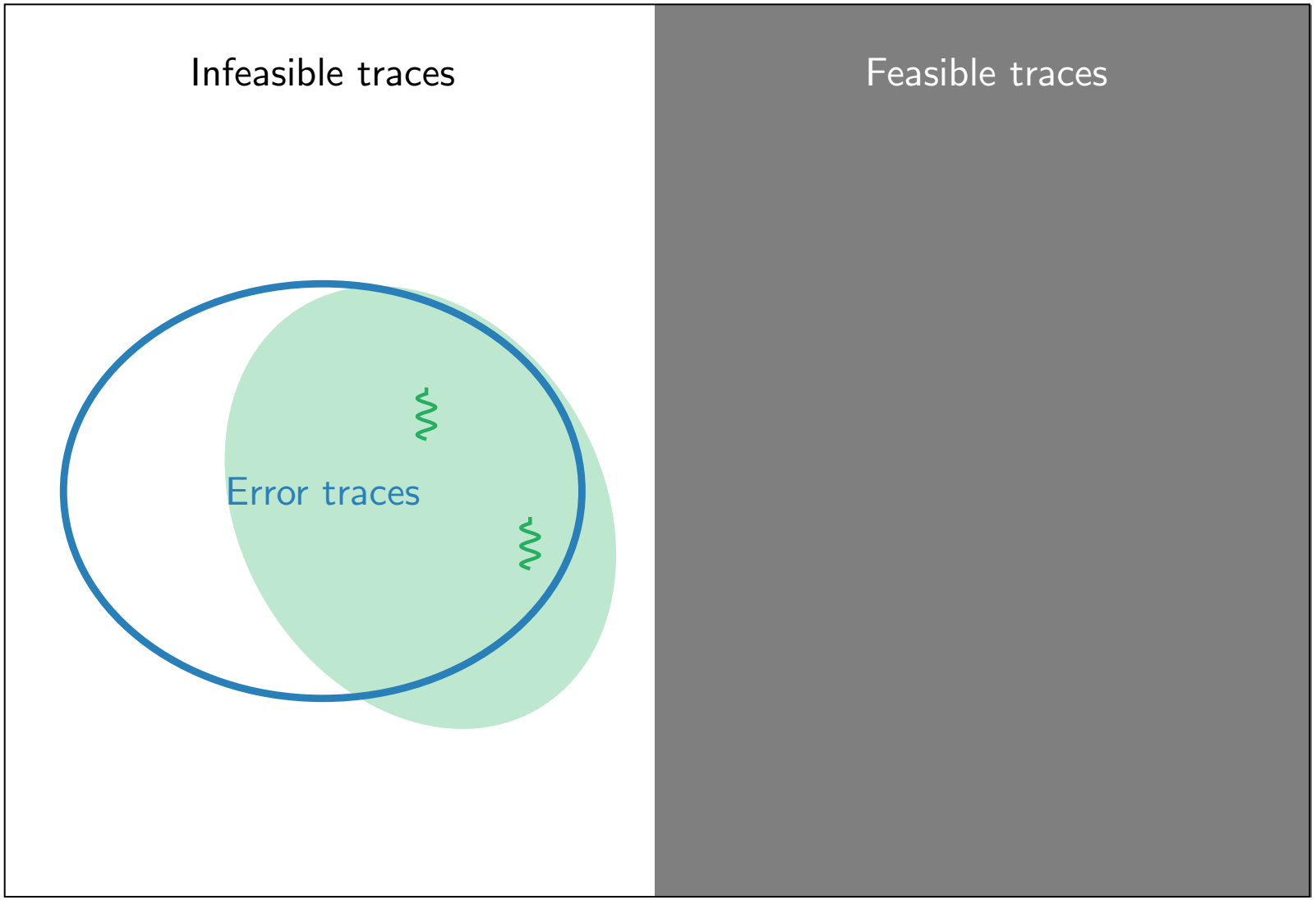

Infeasible traces
Feasible traces
No corresponding executions
At least one corresponding execution

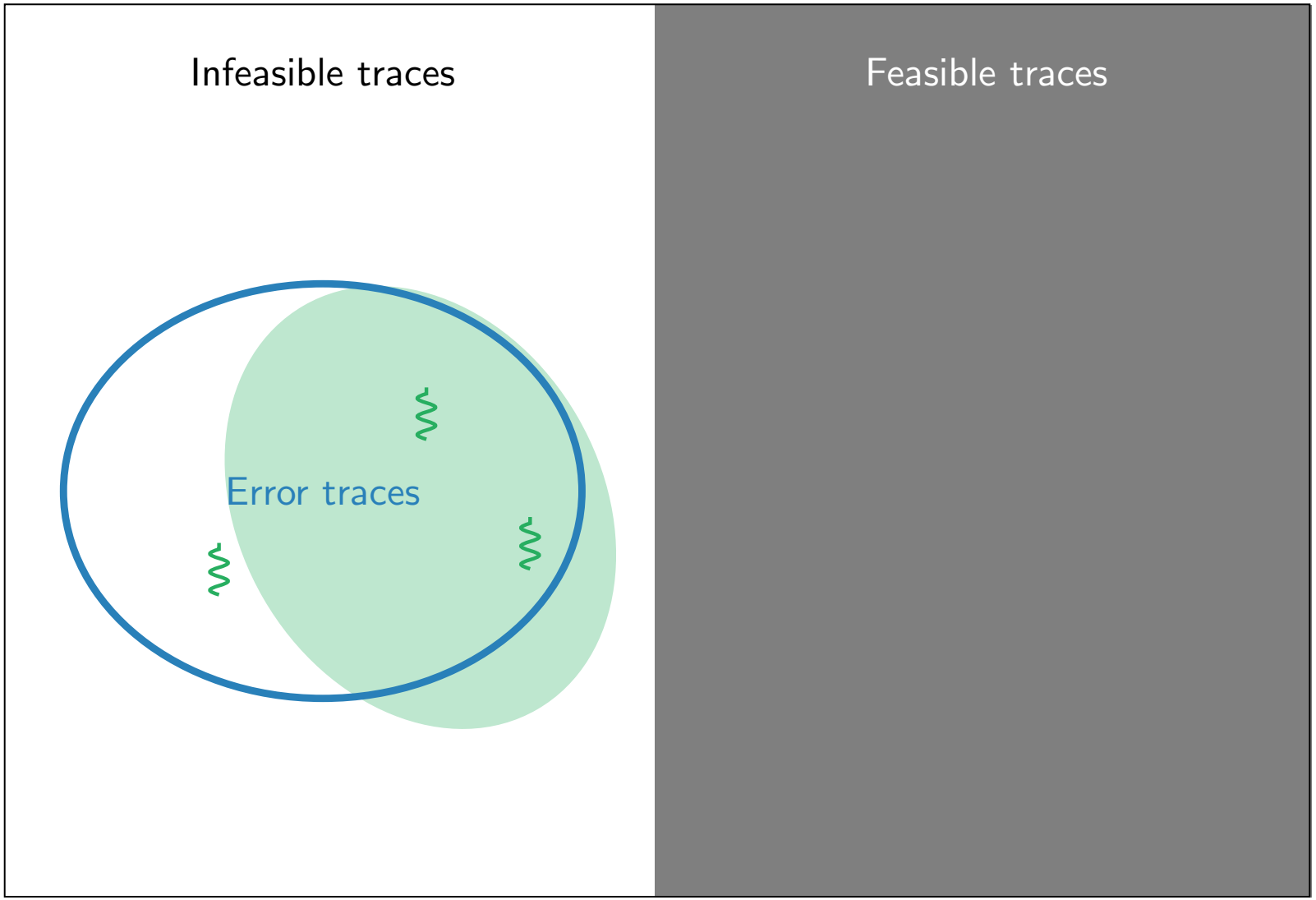

Infeasible traces
Feasible traces
Error traces

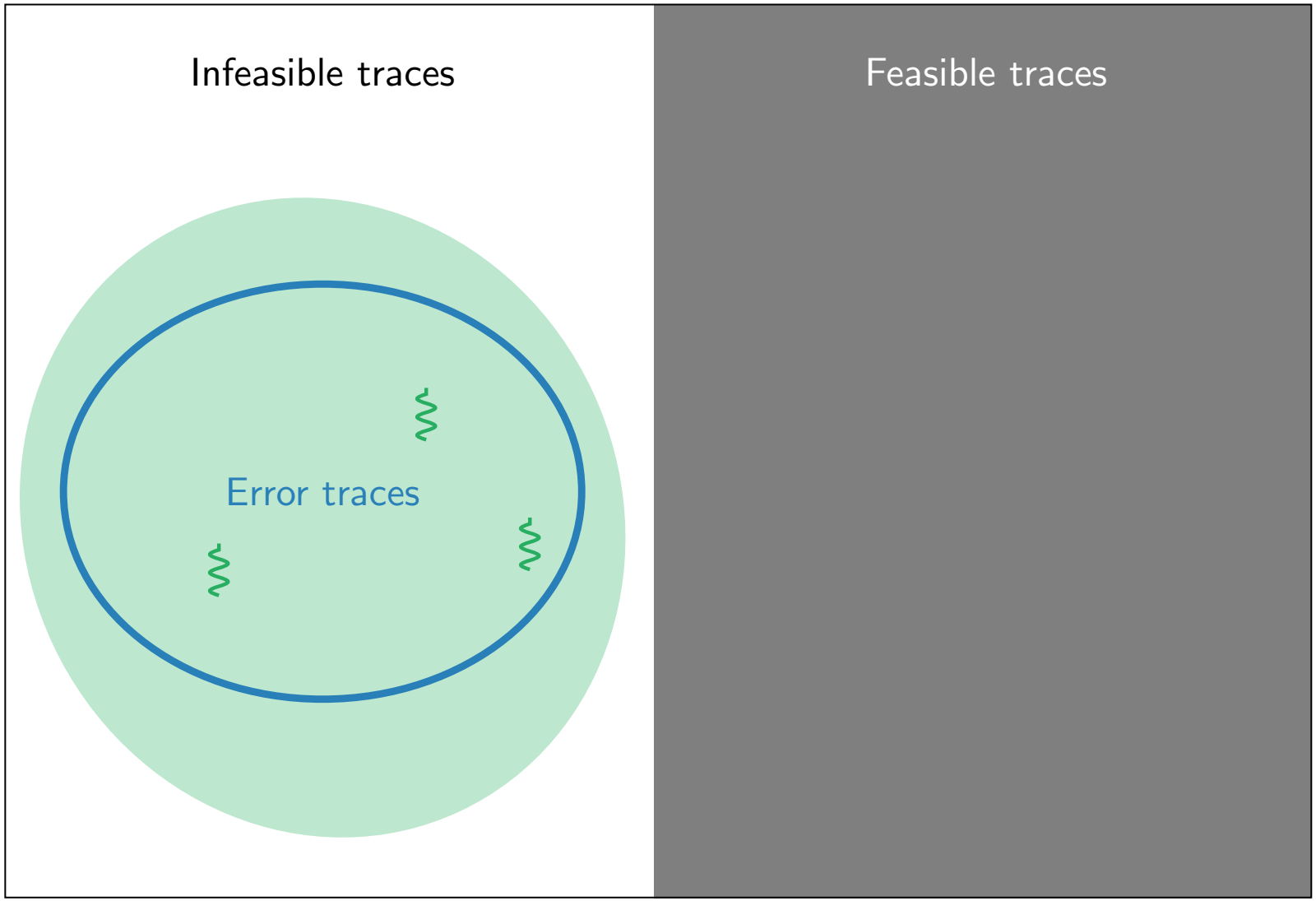

Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Proof Generalization
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces

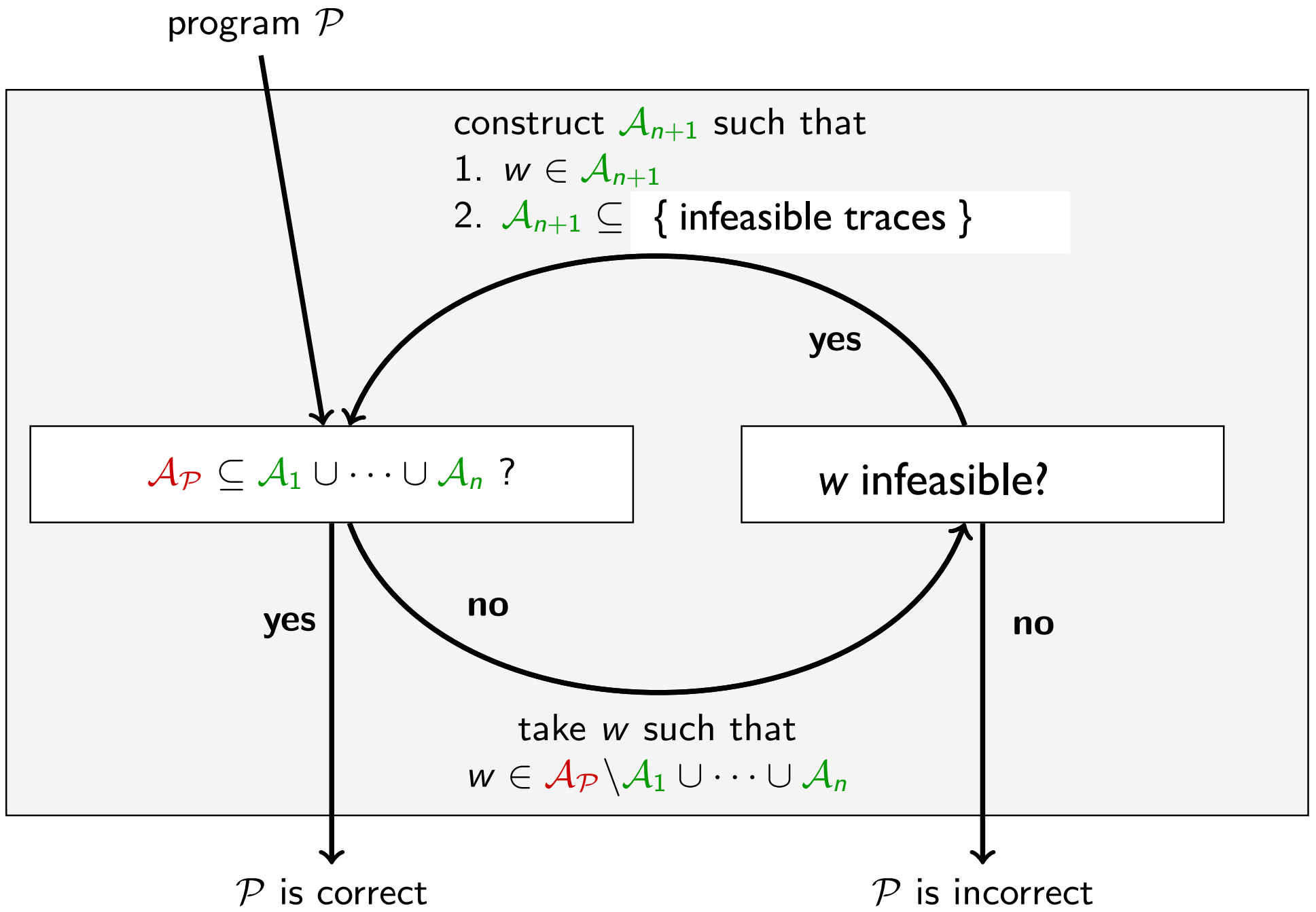
trace abstraction

given a program P,
find a set of correct programs $P_1, \ldots, P_n$
check whether every behavior of P is covered:

$$P \subseteq P_1 \cup \ldots \cup P_n$$

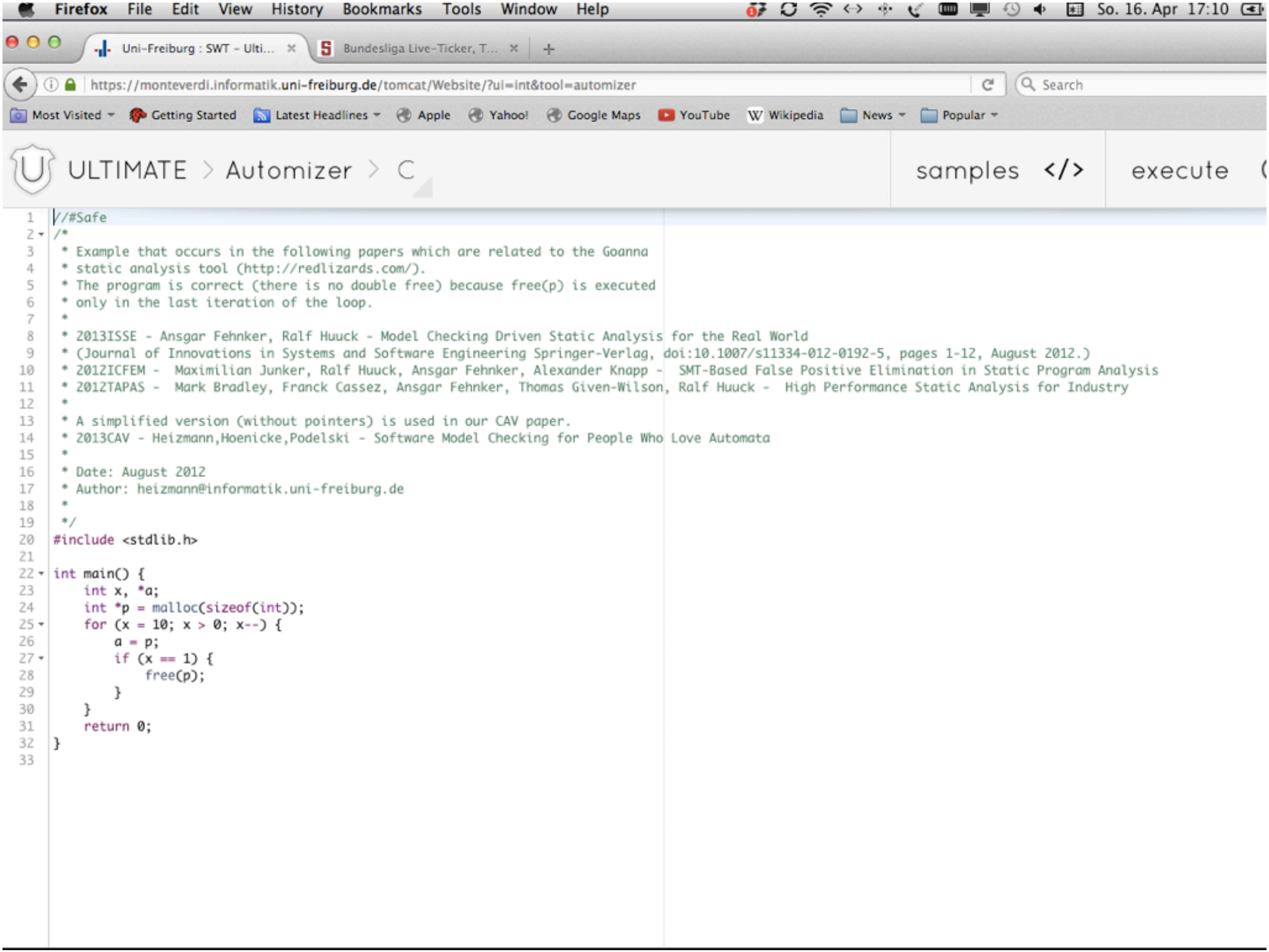
$P_1, \ldots, P_n$ constructed from proofs of traces

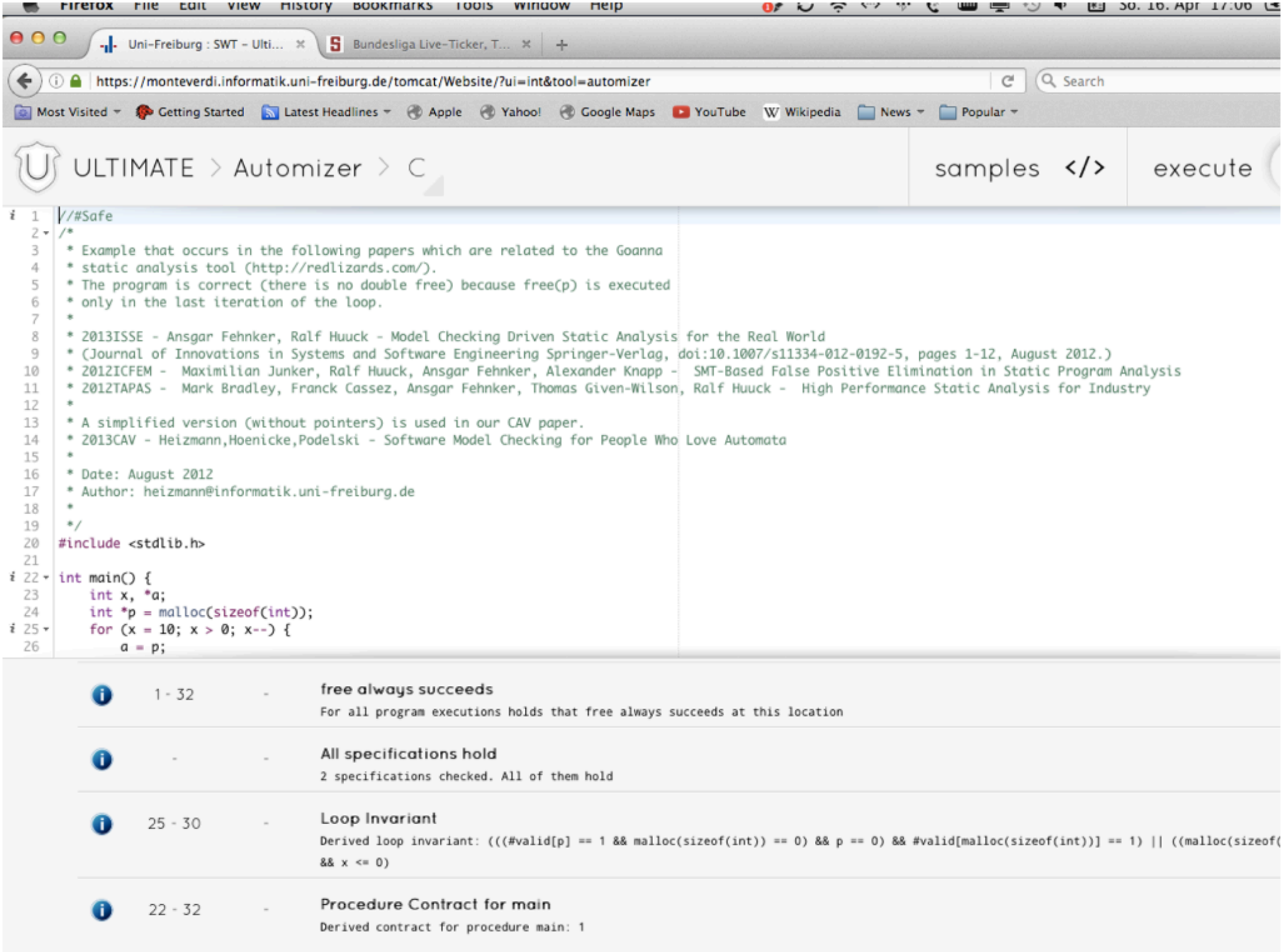
check = inclusion between automata


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

correct?

```
ℓ0: assume p != 0;

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

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

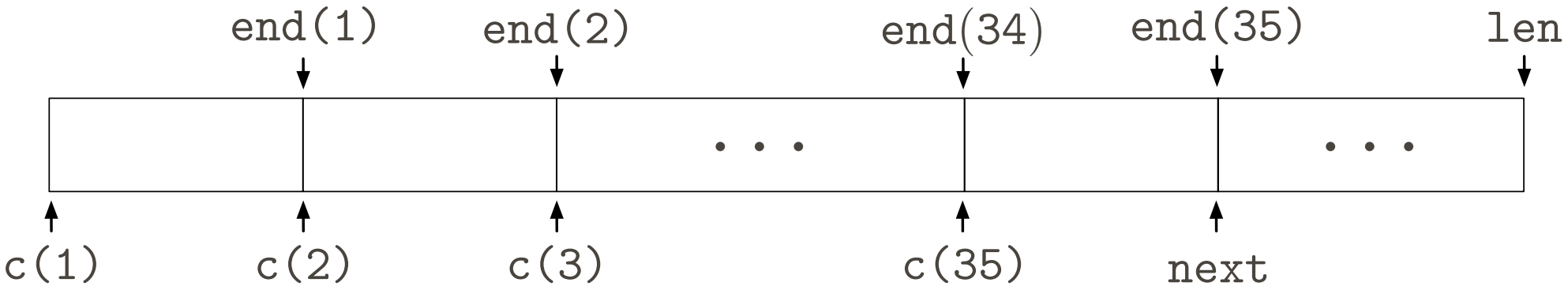

Firefox File Edit View History Bookmarks Tools Window Help
So. 16. Apr 17:10
Uni-Freiburg : SWT - Ulti...
Bundesliga Live-Ticker, T...
https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automizer
Search
Most Visited
Getting Started
Latest Headlines
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
ULTIMATE > Automizer > C
samples </>
execute
1 //#Safe
2 /*
3 * Example that occurs in the following papers which are related to the Goanna
4 * static analysis tool (http://redlizards.com/).
5 * The program is correct (there is no double free) because free(p) is executed
6 * only in the last iteration of the loop.
7 *
8 * 2013ISSE - Ansgar Fehnker, Ralf Huuck - Model Checking Driven Static Analysis for the Real World
9 * (Journal of Innovations in Systems and Software Engineering Springer-Verlag, doi:10.1007/s11334-012-0192-5, pages 1-12, August 2012.)
10 * 2012ICFEM - Maximilian Junker, Ralf Huuck, Ansgar Fehnker, Alexander Knapp - SMT-Based False Positive Elimination in Static Program Analysis
11 * 2012TAPAS - Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, Ralf Huuck - High Performance Static Analysis for Industry
12 *
13 * A simplified version (without pointers) is used in our CAV paper.
14 * 2013CAV - Heizmann,Hoenicke,Podelski - Software Model Checking for People Who Love Automata
15 *
16 * Date: August 2012
17 * Author: heizmann@informatik.uni-freiburg.de
18 *
19 */
20 #include <stdlib.h>
21
22 int main() {
23 int x, *a;
24 int *p = malloc(sizeof(int));
25 for (x = 10; x > 0; x--) {
26 a = p;
27 if (x == 1) {
28 free(p);
29 }
30 }
31 return 0;
32 }
33


Firefox File Edit View History Bookmarks Tools Window Help
Uni-Freiburg : SWT - Ulti...
Bundesliga Live-Ticker, T...
https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automizer
Search
Most Visited
Getting Started
Latest Headlines
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News
Popular
ULTIMATE > Automizer > C
samples </>
execute
1 //#Safe
2 /*
3 * Example that occurs in the following papers which are related to the Goanna
4 * static analysis tool (http://redlizards.com/).
5 * The program is correct (there is no double free) because free(p) is executed
6 * only in the last iteration of the loop.
7 *
8 * 2013ISSE - Ansgar Fehnker, Ralf Huuck - Model Checking Driven Static Analysis for the Real World
9 * (Journal of Innovations in Systems and Software Engineering Springer-Verlag, doi:10.1007/s11334-012-0192-5, pages 1-12, August 2012.)
10 * 2012ICFEM - Maximilian Junker, Ralf Huuck, Ansgar Fehnker, Alexander Knapp - SMT-Based False Positive Elimination in Static Program Analysis
11 * 2012TAPAS - Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, Ralf Huuck - High Performance Static Analysis for Industry
12 *
13 * A simplified version (without pointers) is used in our CAV paper.
14 * 2013CAV - Heizmann,Hoenicke,Podelski - Software Model Checking for People Who Love Automata
15 *
16 * Date: August 2012
17 * Author: heizmann@informatik.uni-freiburg.de
18 *
19 */
20 #include <stdlib.h>
21
22 int main() {
23 int x, *a;
24 int *p = malloc(sizeof(int));
25 for (x = 10; x > 0; x--) {
26 a = p;
1 - 32 - free always succeeds
For all program executions holds that free always succeeds at this location
- - All specifications hold
2 specifications checked. All of them hold
25 - 30 - Loop Invariant
Derived loop invariant: (((#valid[p] == 1 && malloc(sizeof(int)) == 0) && p == 0) && #valid[malloc(sizeof(int))] == 1) || ((malloc(sizeof(
&& x <= 0)
22 - 32 - Procedure Contract for main
Derived contract for procedure main: 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;
```

```
   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;
```

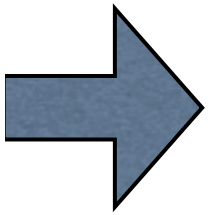
```
thread T:
        local int : c;       //  position of current task
        local int : end; //  position of last task in acquired block
        //  acquire block of tasks
1       lock(m);
2           if(next + 10 <= len)
3               { c := next; next := next + 10; end := next; }
4          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



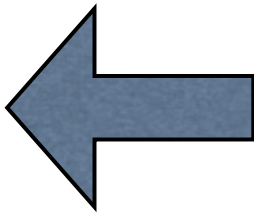
threads $1, 2, \ldots, 35$
have acquired block of tasks
have not yet started working

Next ...

- learn correct programs from unsatisfiability proofs
- learn correct programs from Hoare triple proofs



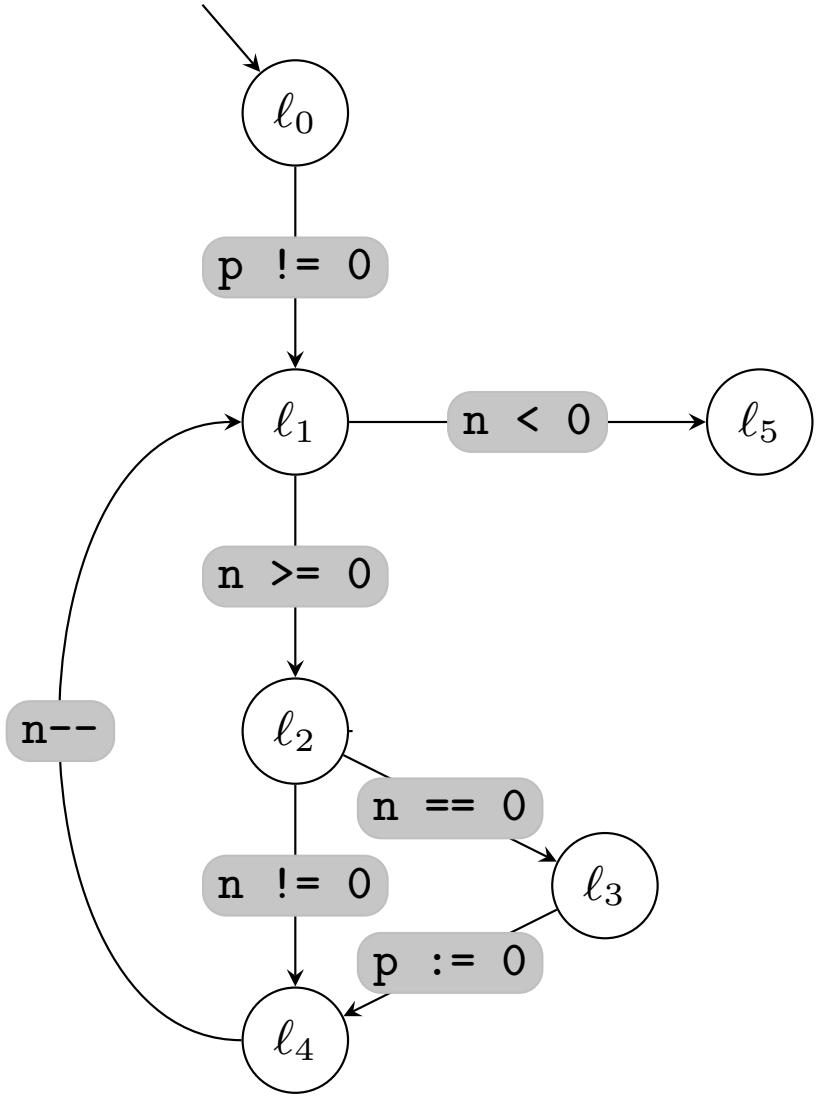
- learn correct programs from unsatisfiability proofs
- learn correct programs from Hoare triple proofs



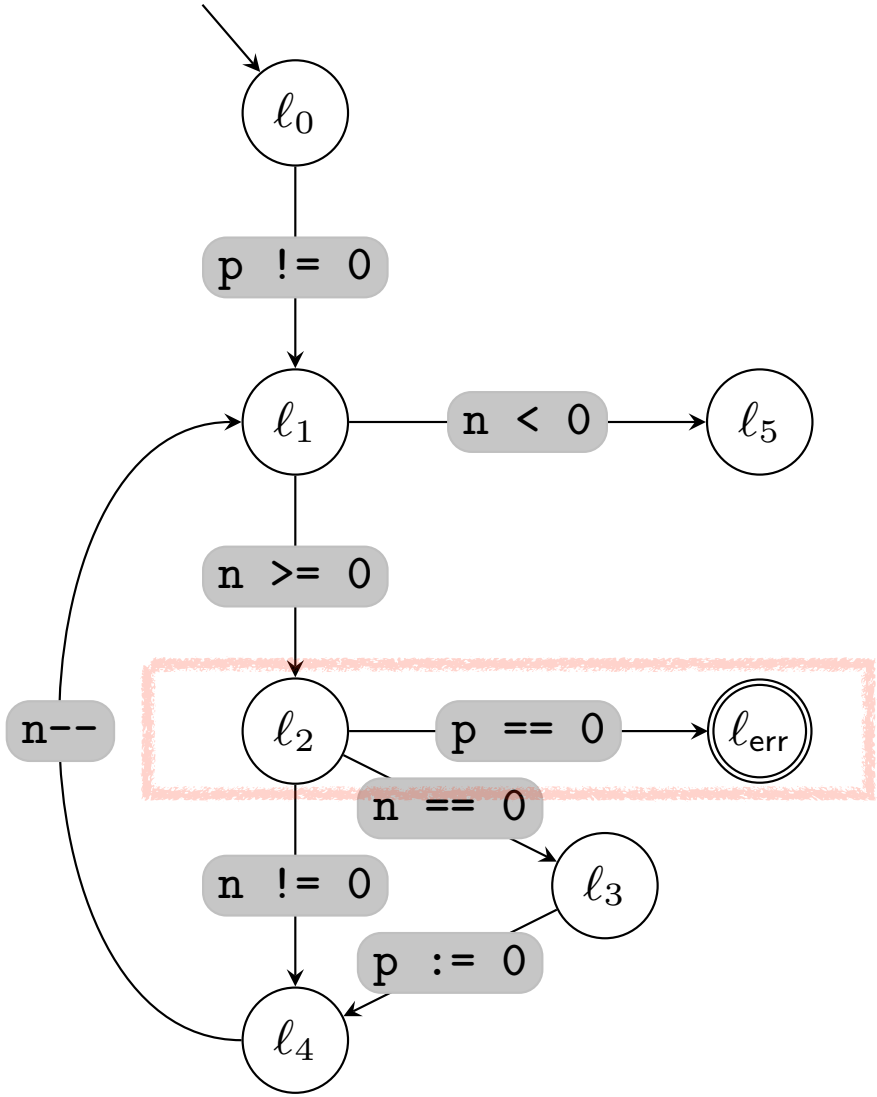
correct?

```
ℓ0: assume p != 0;

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

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

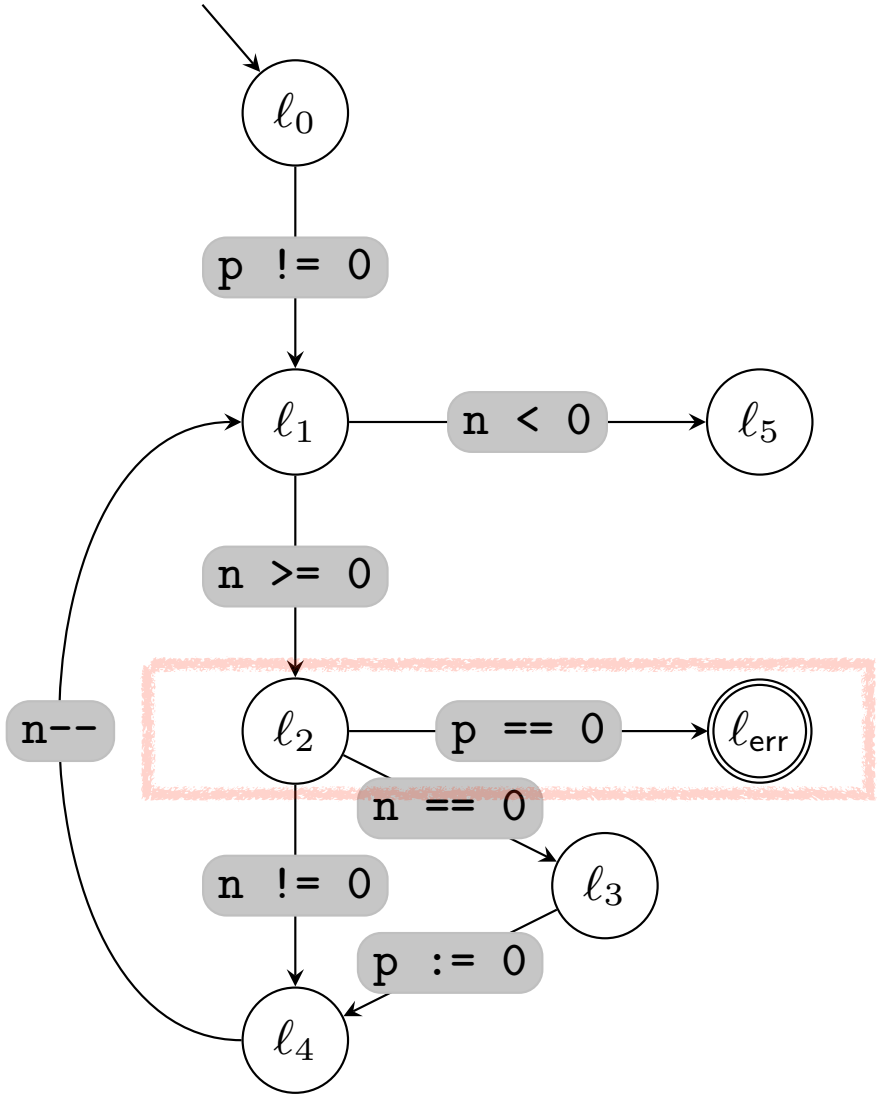
```
ℓ0: assume p != 0;

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

all inter-reducible:

validity of assert statement

non-reachability of error location

validity of safety property

validity of invariant

infeasibility of control flow traces

partial correctness

partial correctness for pre/postcondition (*true*, *false*)

infeasible

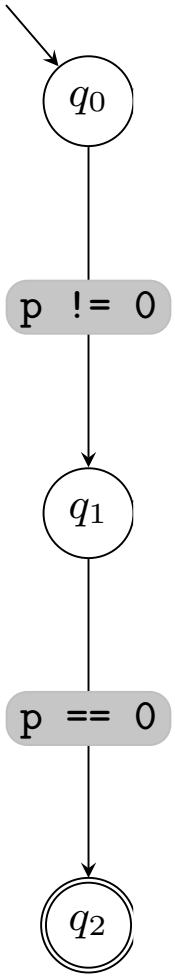
$\Longleftrightarrow$

correct wrt. pre/condition pair *(true, false)*

{ *true* } `x == 1 ; x == -1 ;` { *false* }

{ *true* } `x := 1 ; x == -1 ;` { *false* }

correct program = infeasible trace




```
(p != 0)

(p==0)
```

infeasible trace

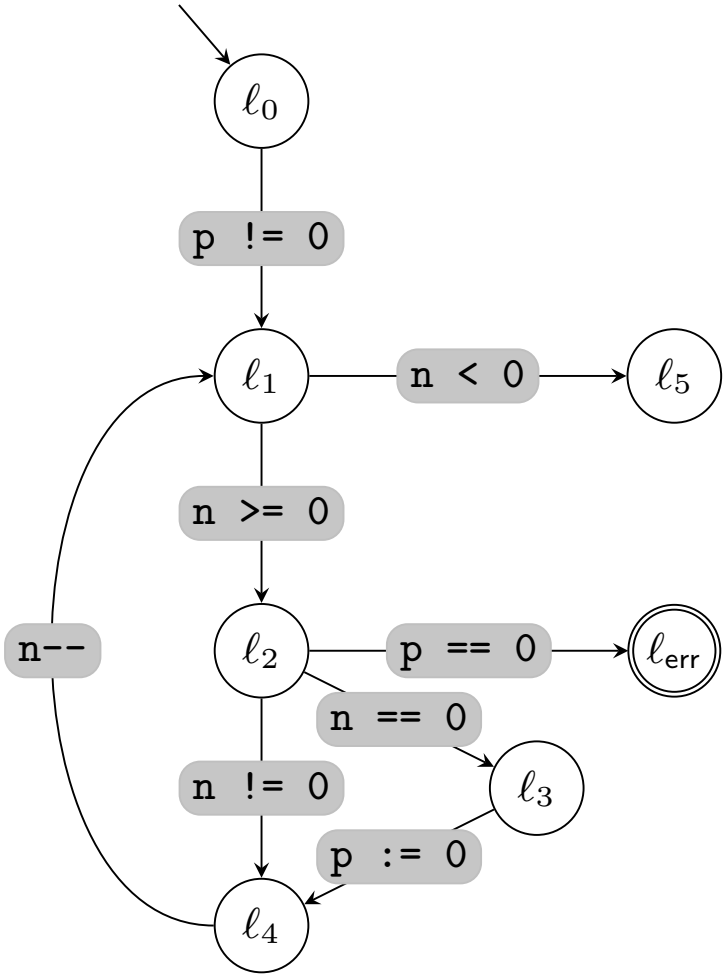
unsatisfiable formula

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

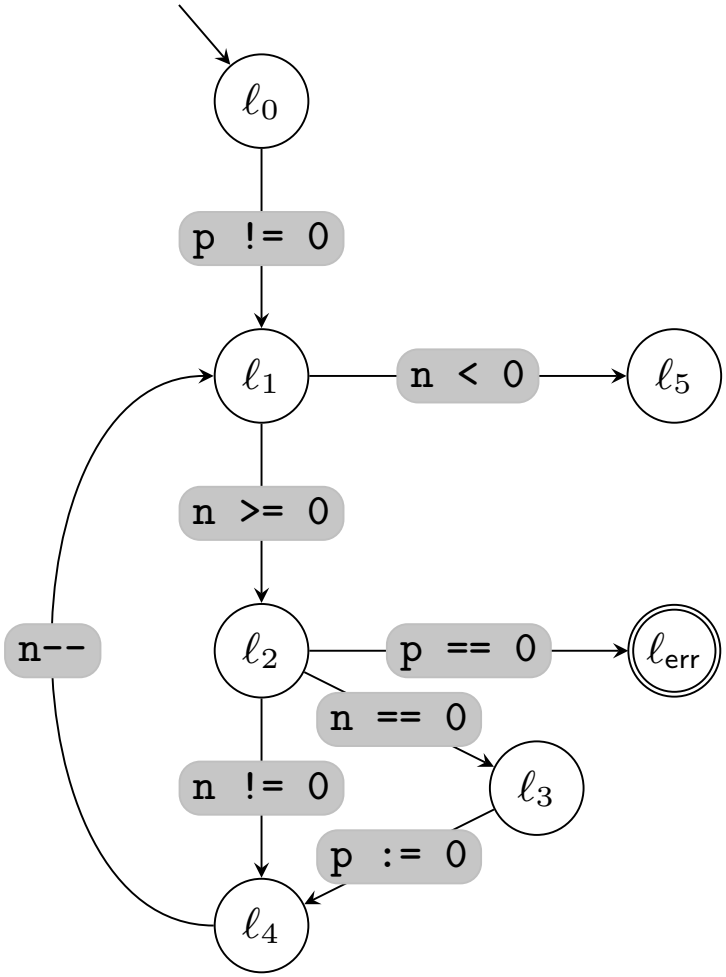
$x = 1 \ \wedge \ x = -1$

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

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



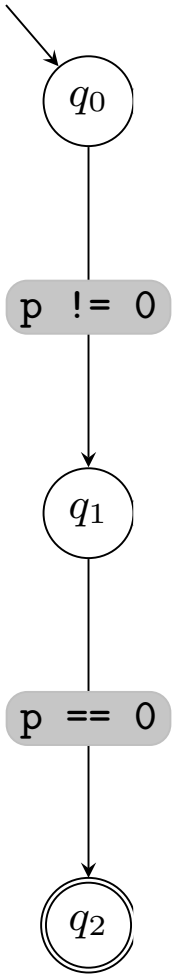

```
(p != 0)
(n >= 0)
(p == 0)
```




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

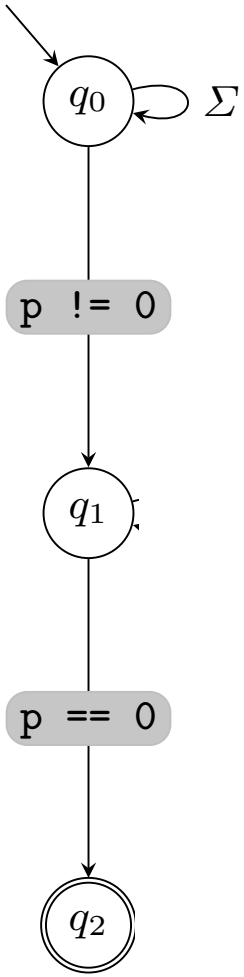
```
(p != 0)

(p==0)
```

```
(p != 0)

(p==0)
```



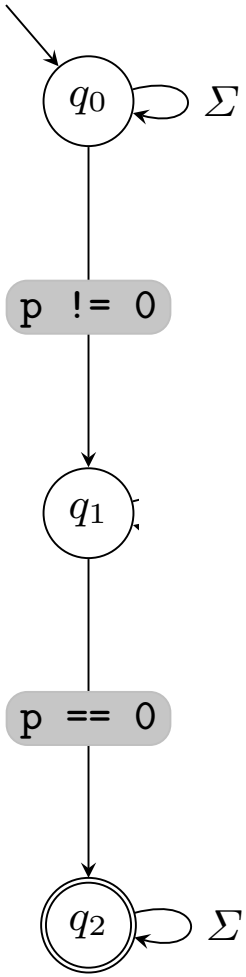
(p != 0)

(p==0)

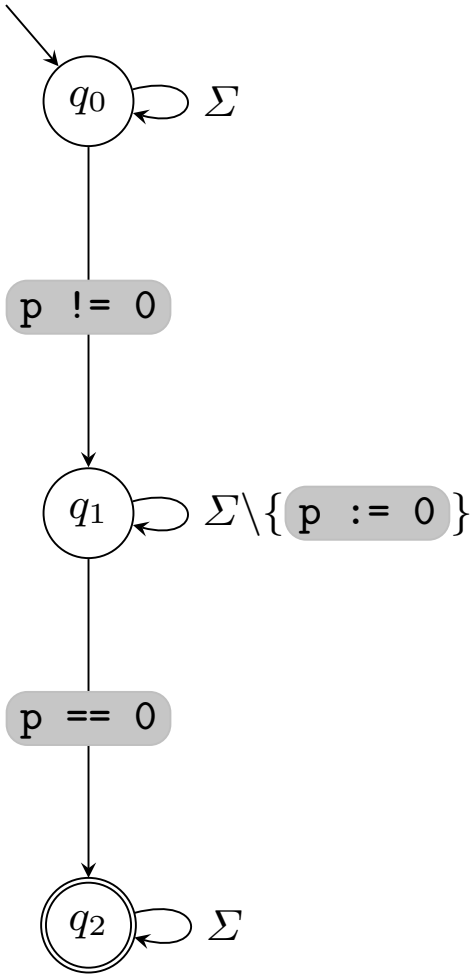


(p != 0)

(p==0)



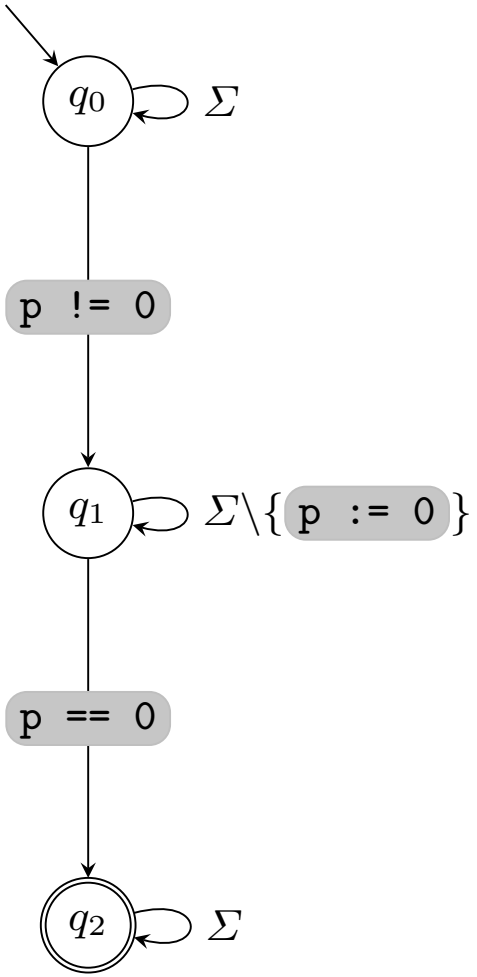

```
(p != 0)

(p==0)
```



(p != 0)

(p==0)

correct program (error location is not reachable)

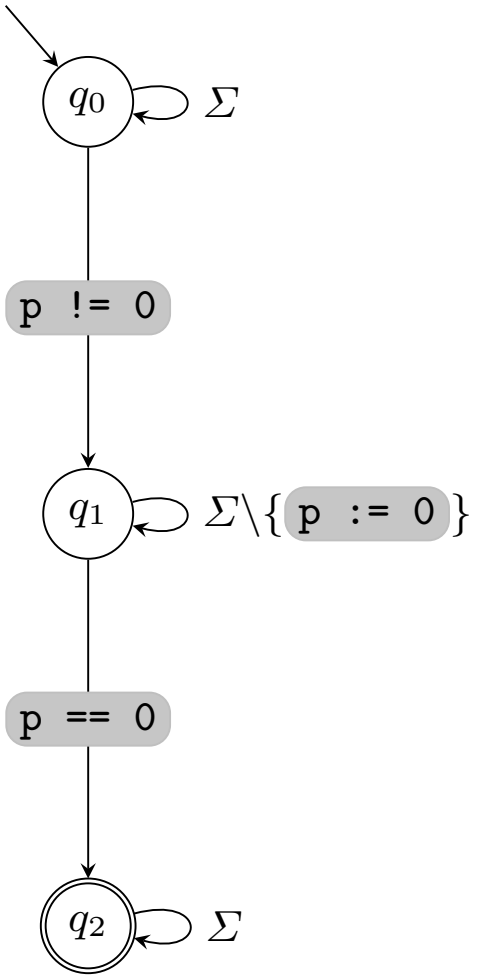



```
(p != 0)
(n >= 0)
(p == 0)
```

all error traces of program have the *same* proof as sample trace
(same unsatisfiable core of unsatisfiability proof)

correct program P_1 constructed from a proof

P_1

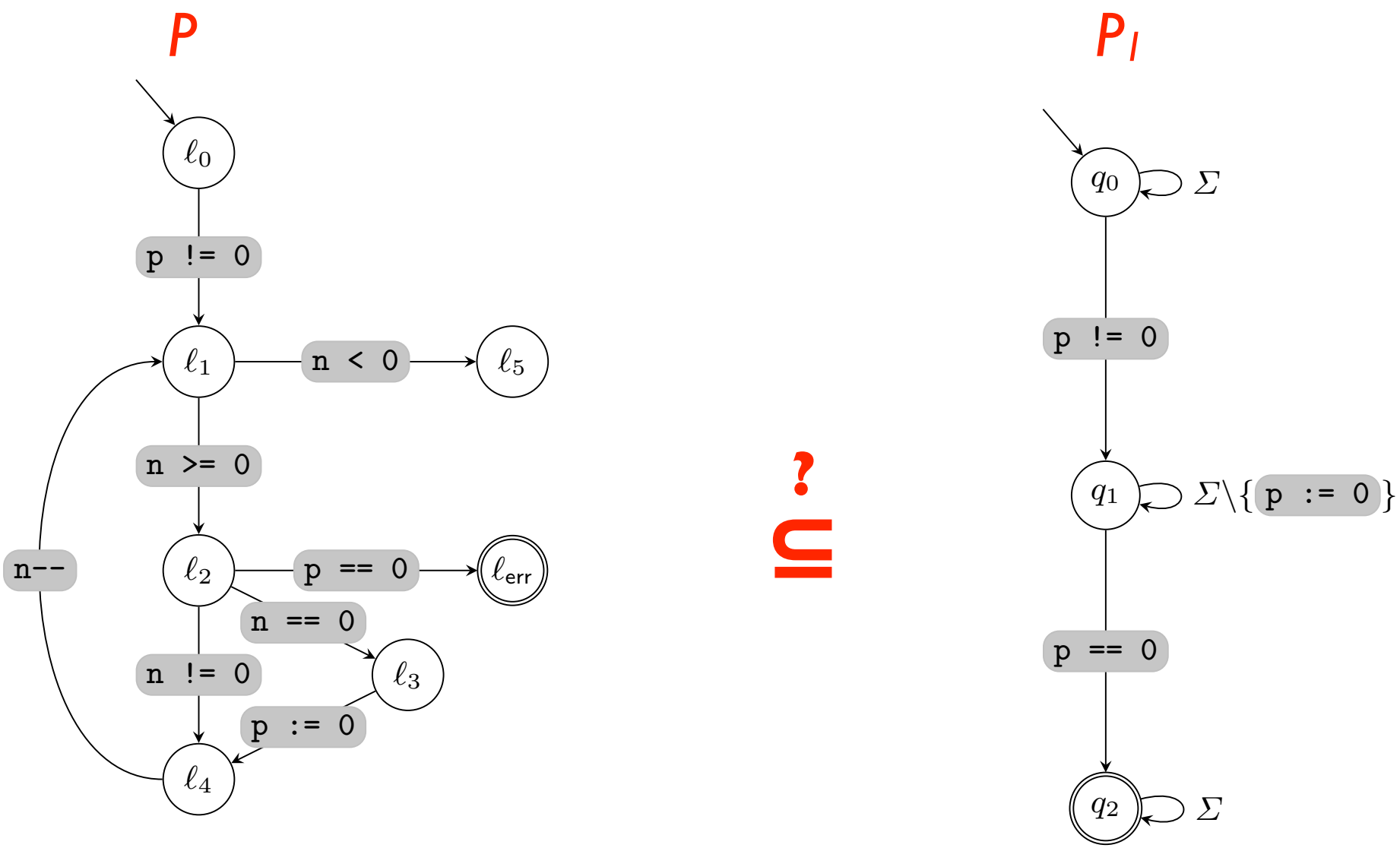


(p != 0)

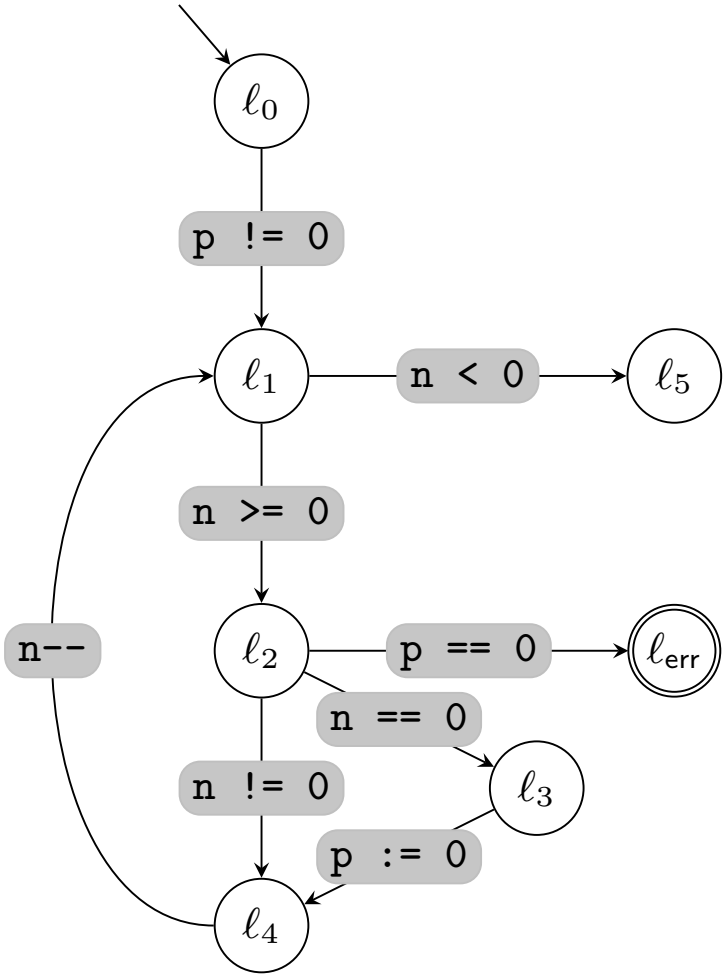
(p==0)

... from unsatisfiable core of unsatisfiability proof for sample trace

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

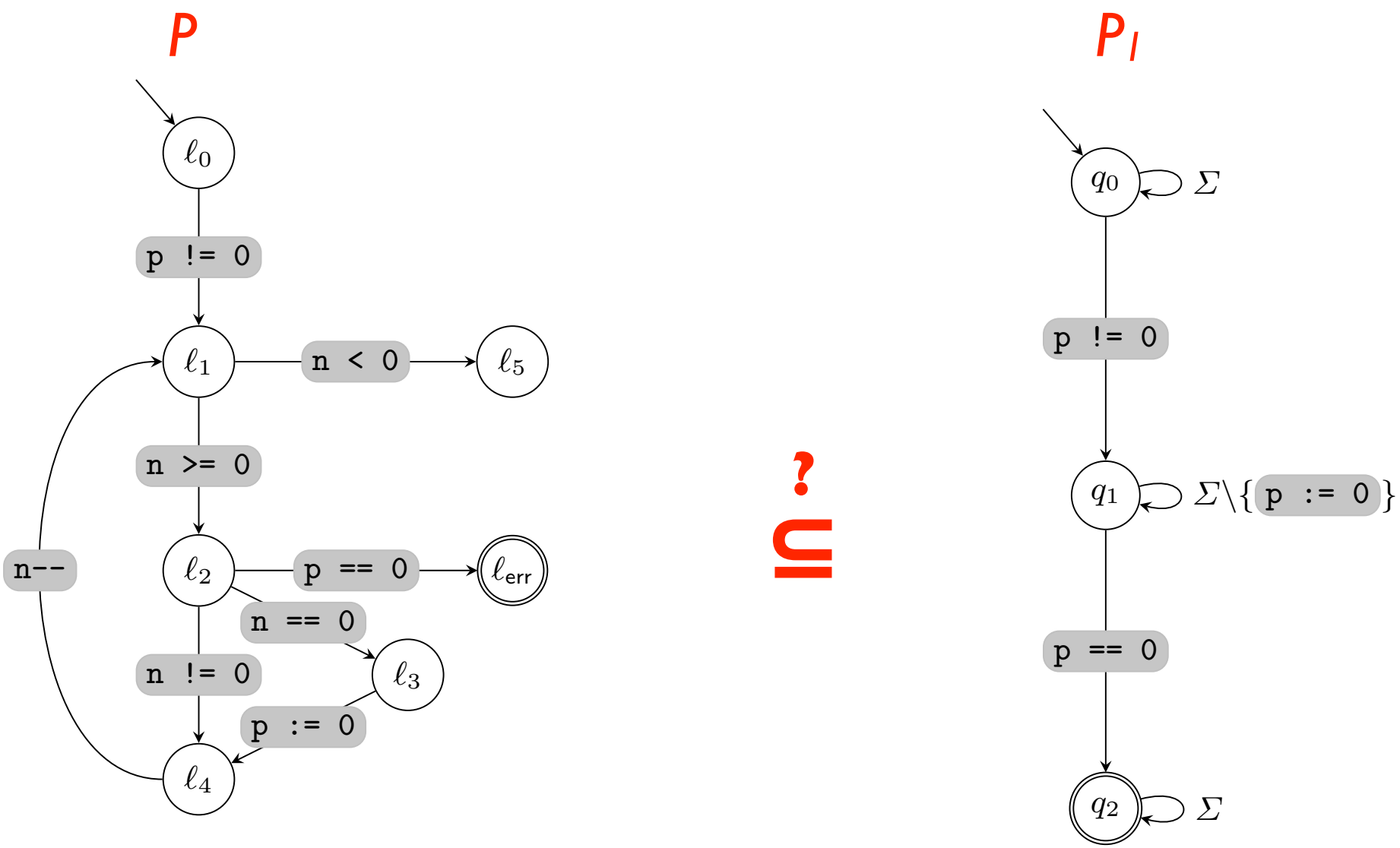


does a proof *exist* for every error trace ?

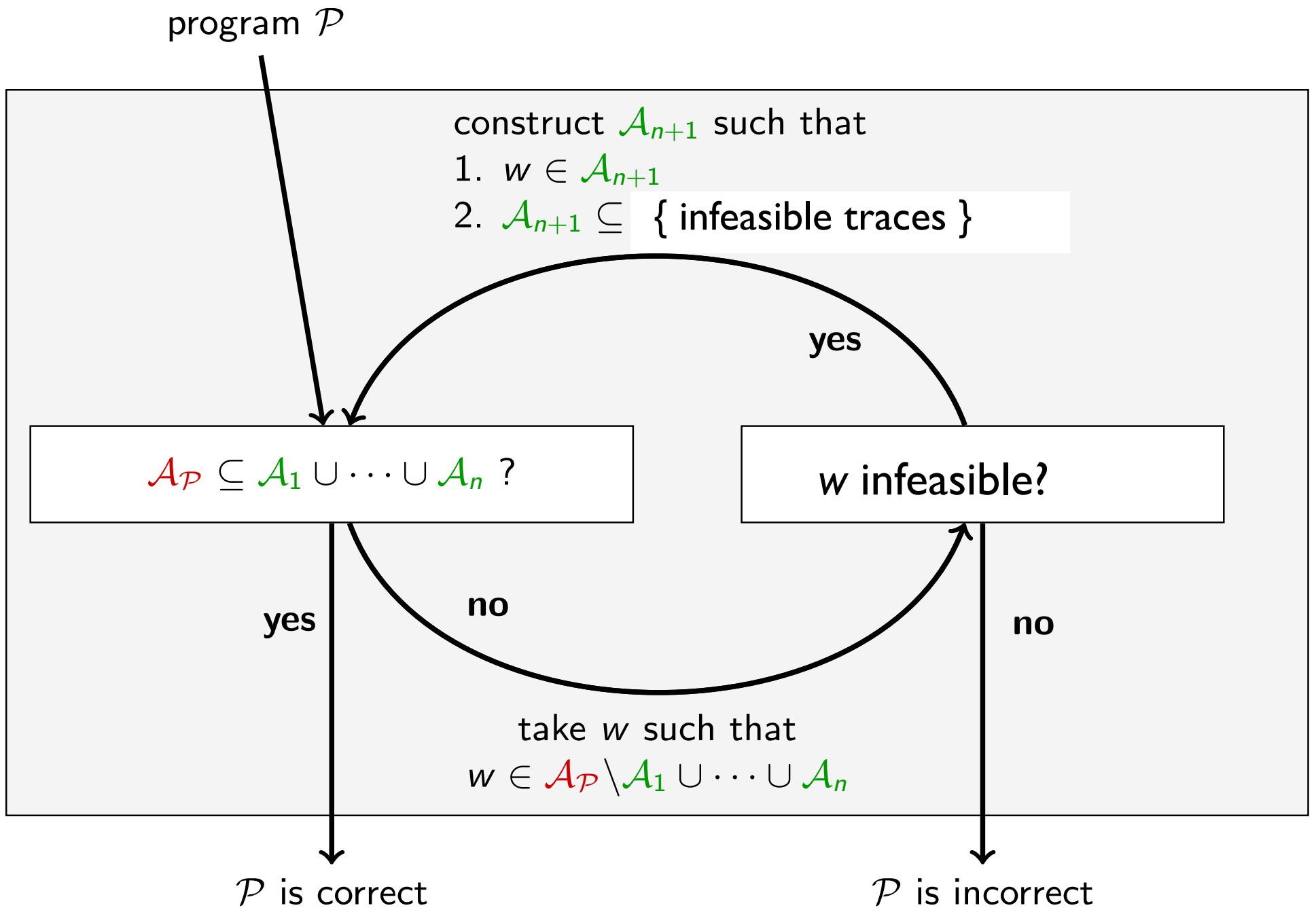


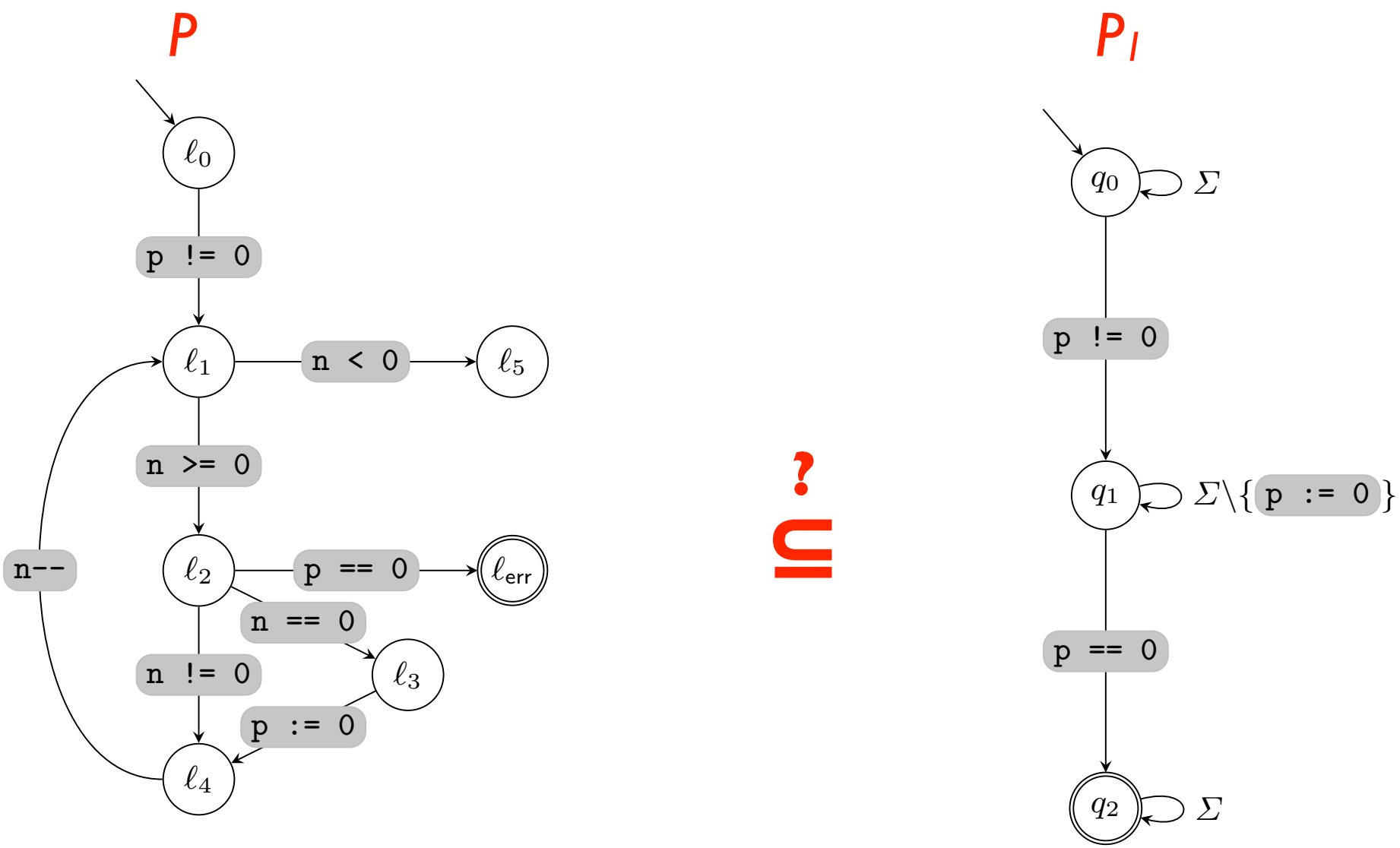
automaton

alphabet: {statements}

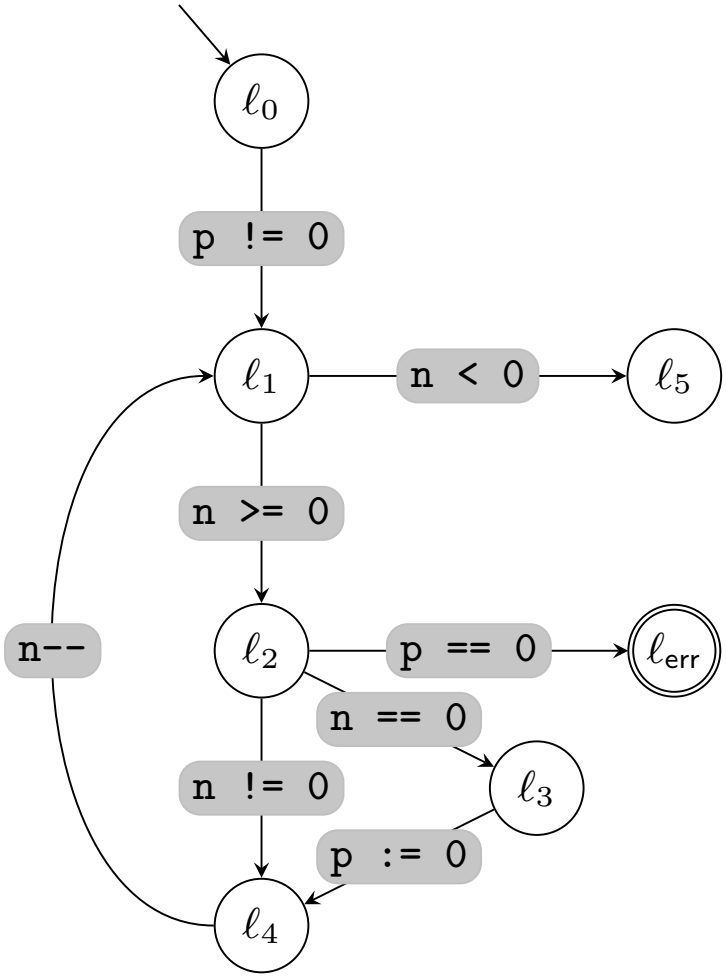


inclusion between automata


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



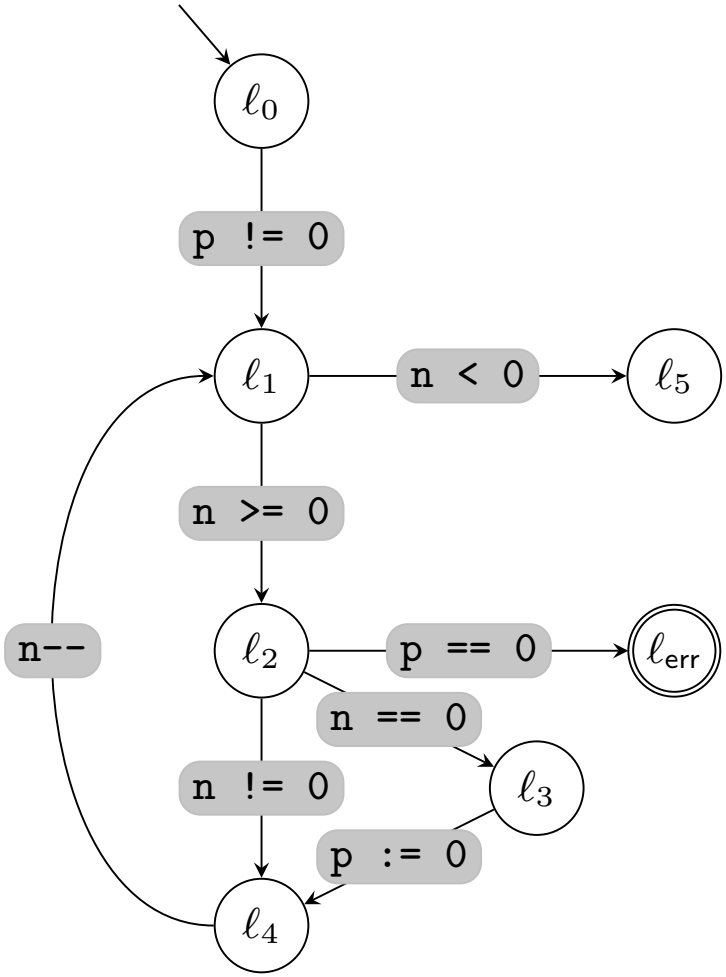
inclusion check fails and returns word in $P \setminus P_I$



new trace:

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

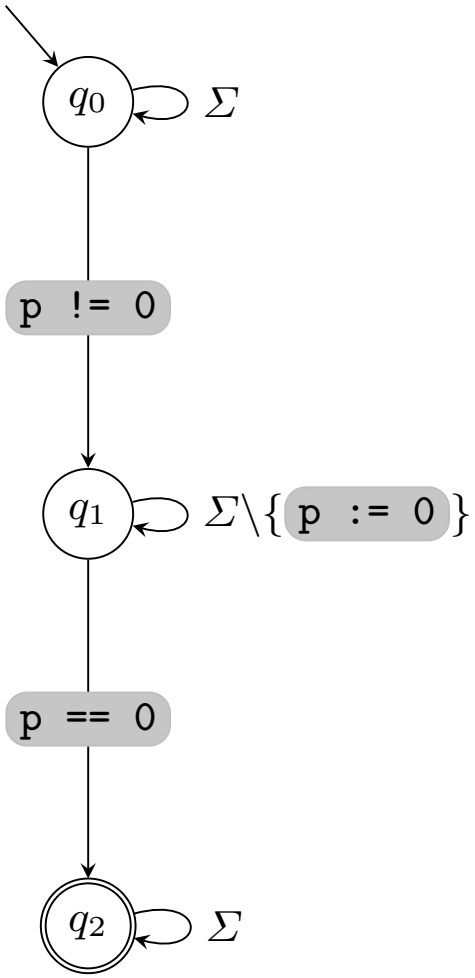
P



word in *P*

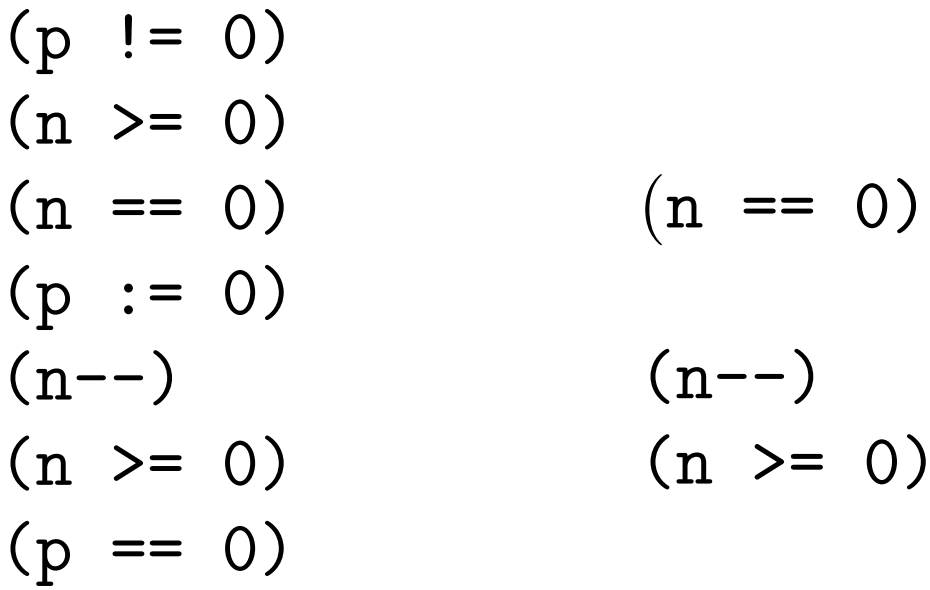
```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```

P_I

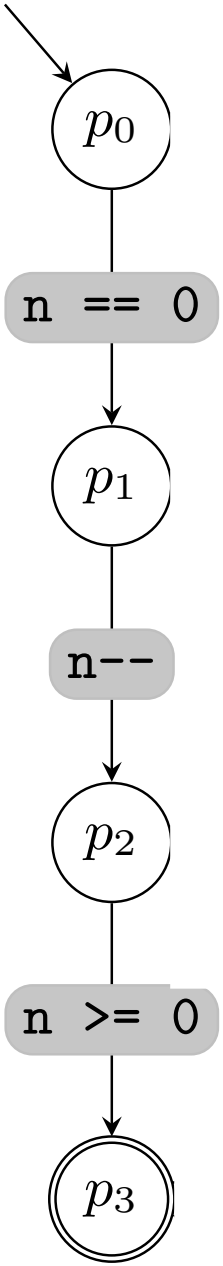


word *not* in P_I

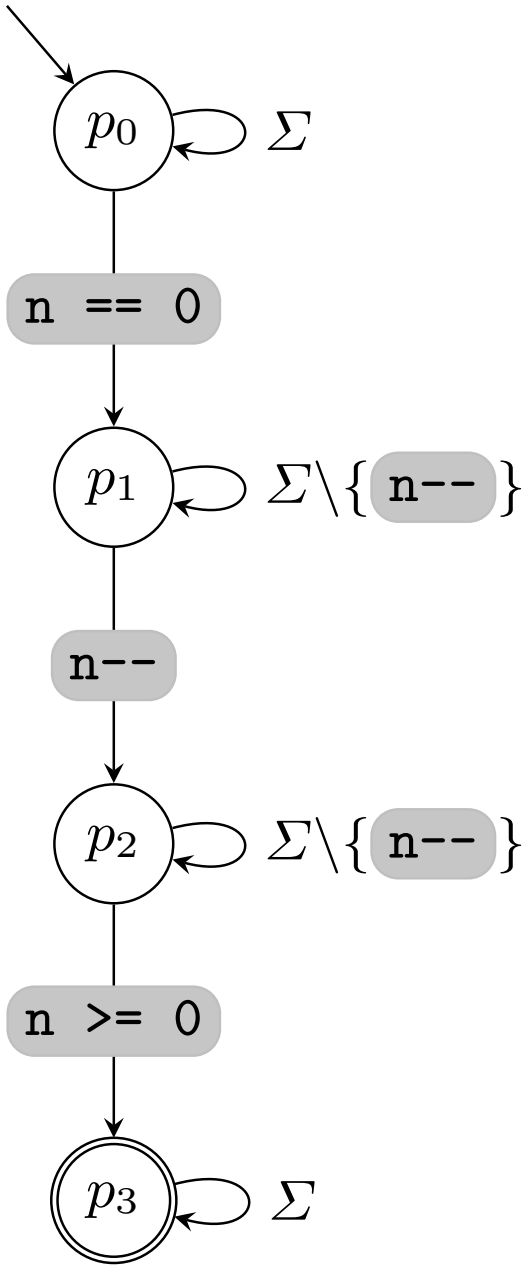
```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```


(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)
```

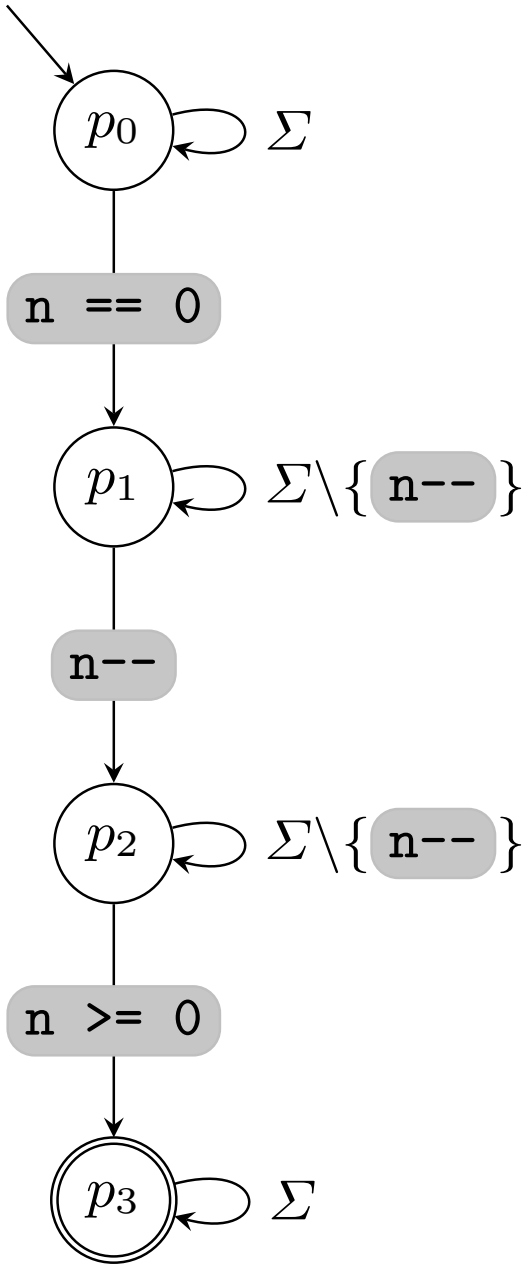

p_0
Σ
n == 0
p_1
$\Sigma\backslash\{$ n-- $\}$
n--
p_2
$\Sigma\backslash\{$ n-- $\}$
n >= 0
p_3
Σ

(n == 0)

(n--)

(n >= 0)

correct program P_2 constructed from a proof

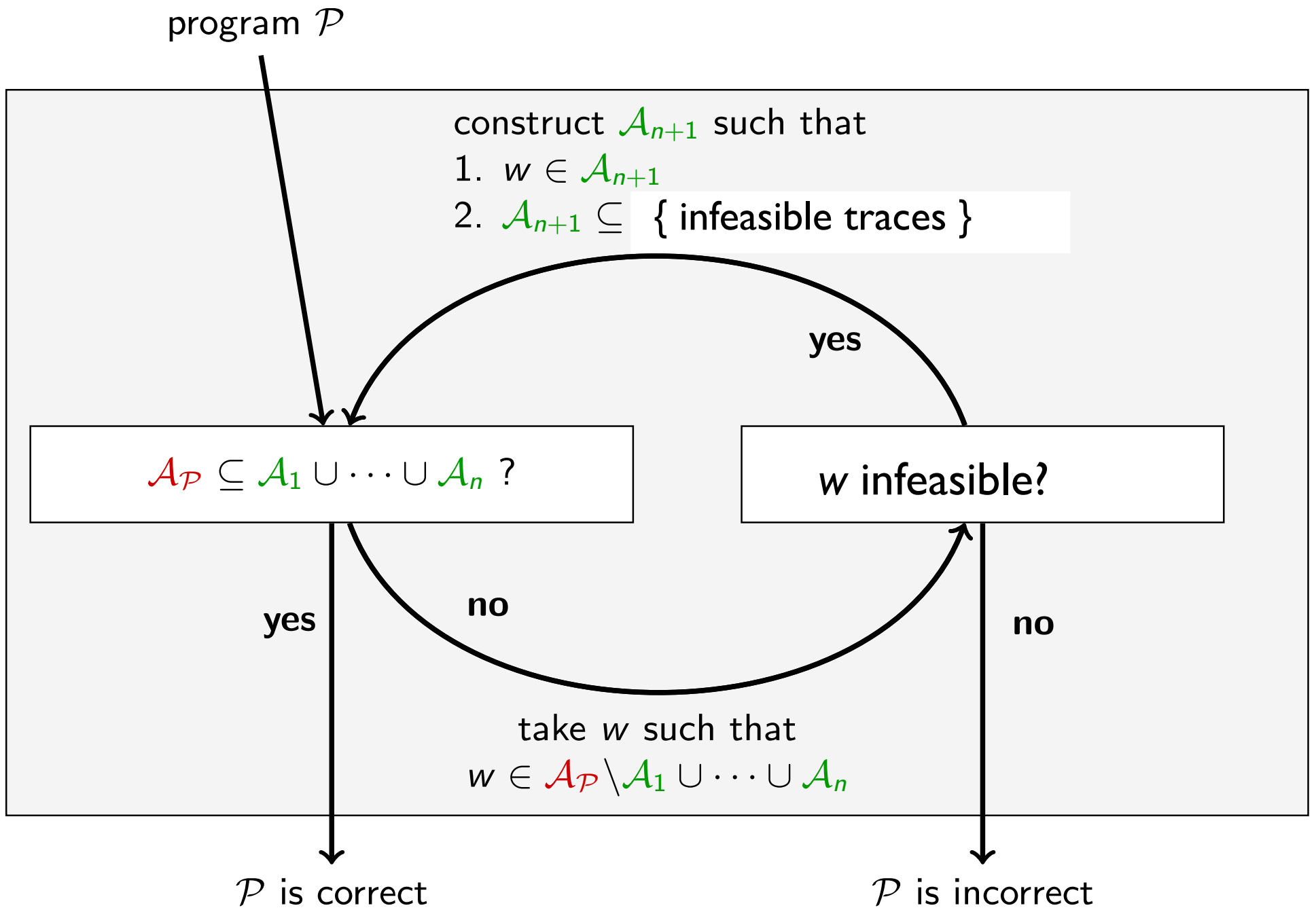


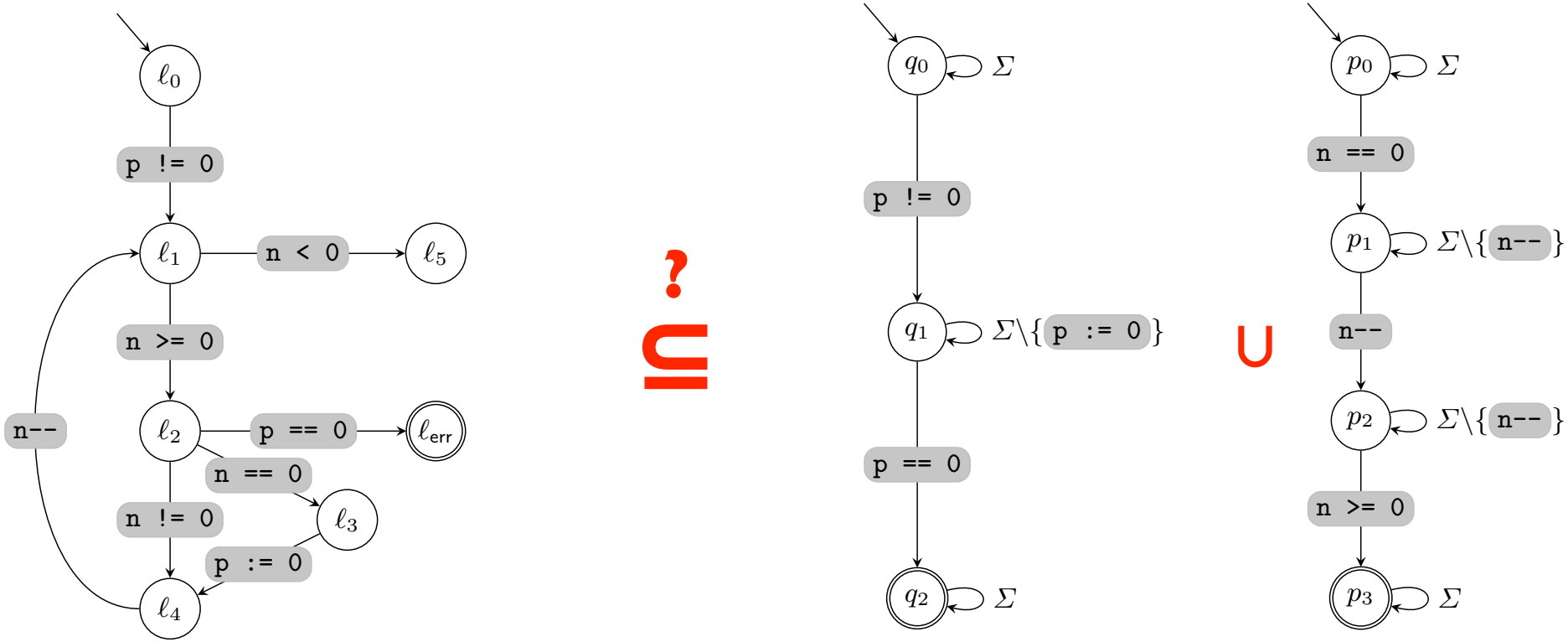
(n == 0)

(n--)
(n >= 0)

... from unsatisfiable core of unsatisfiability proof for sample trace

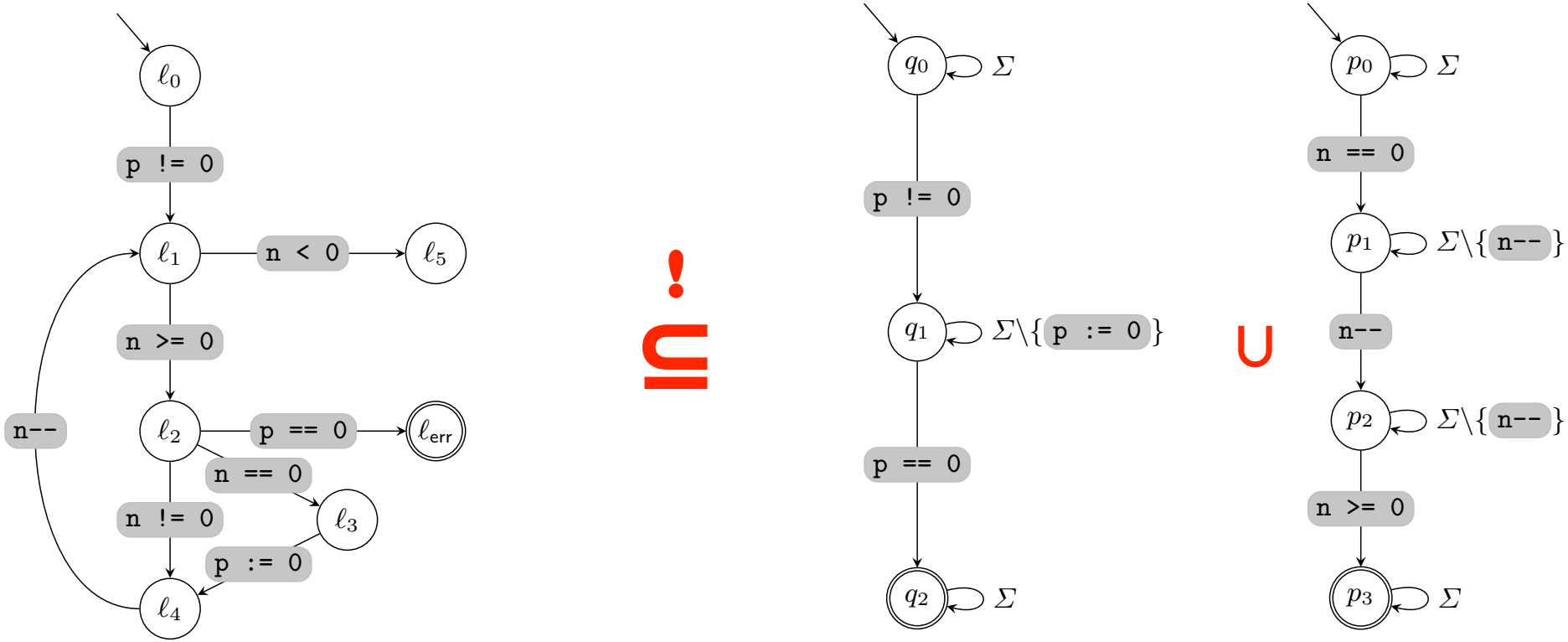
```
(p != 0)
(n >= 0)
(n == 0)
(p := 0)
(n--)
(n >= 0)
(p == 0)
```


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



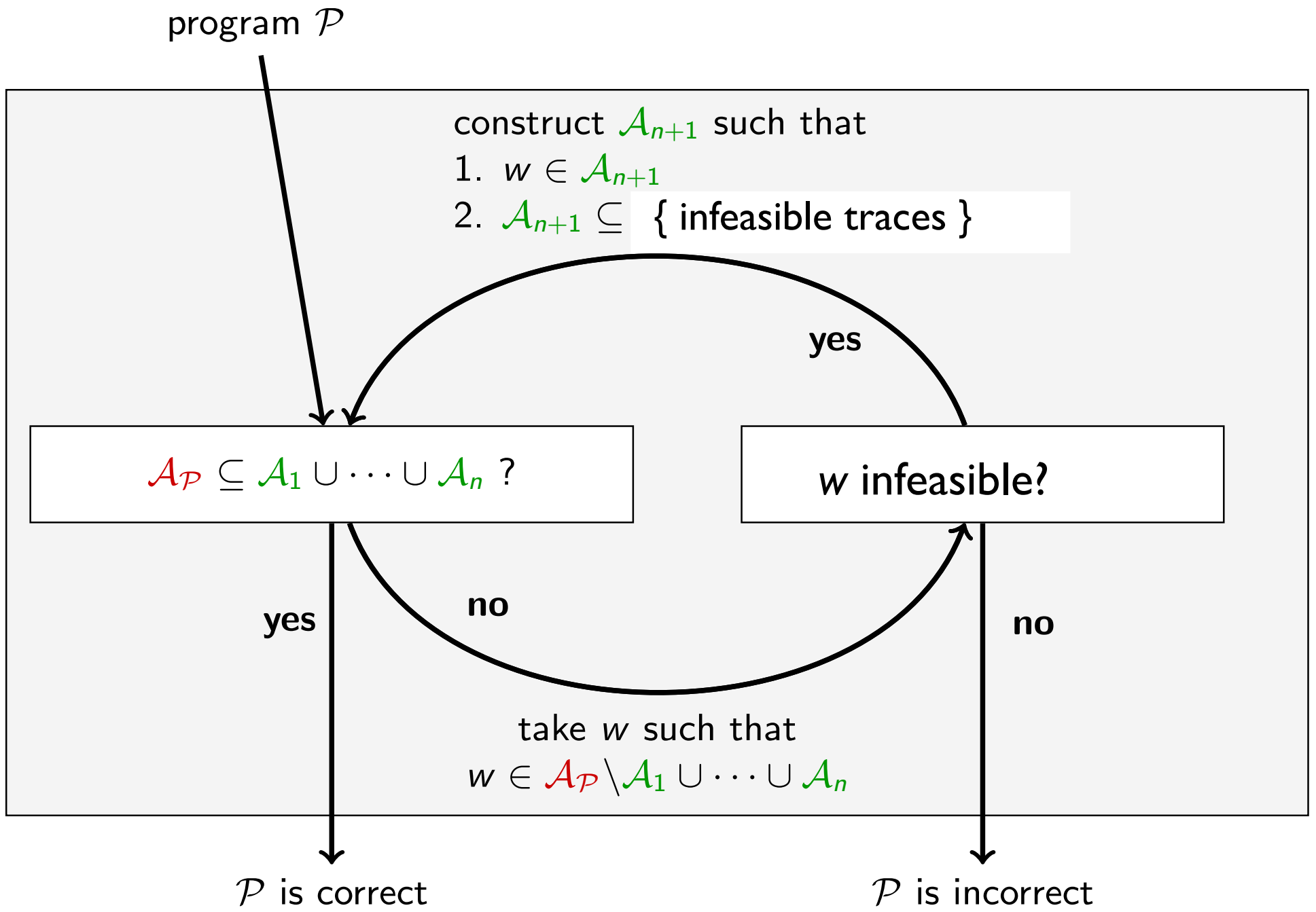
check inclusion between automata

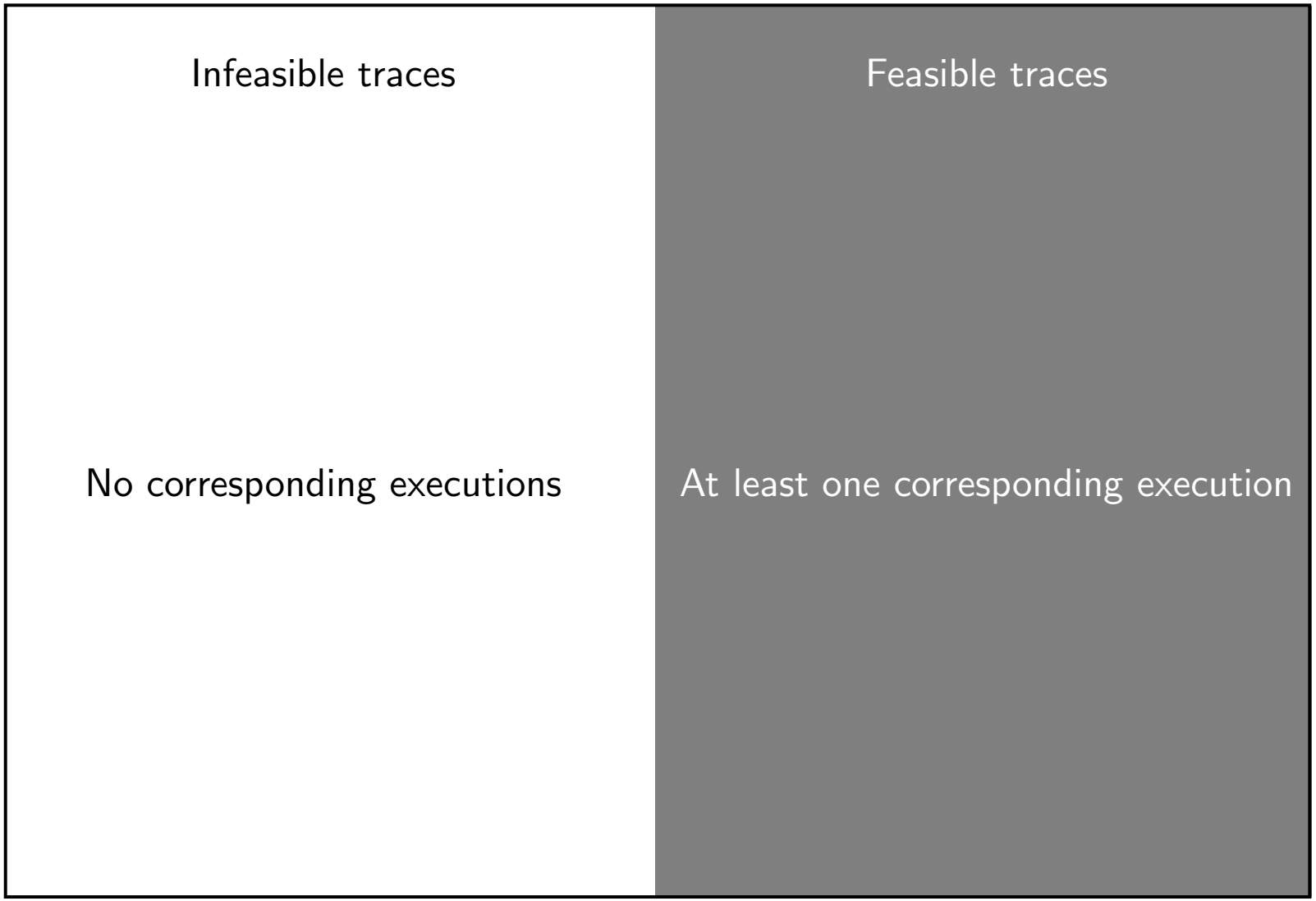
does a proof *exist* for every trace ?

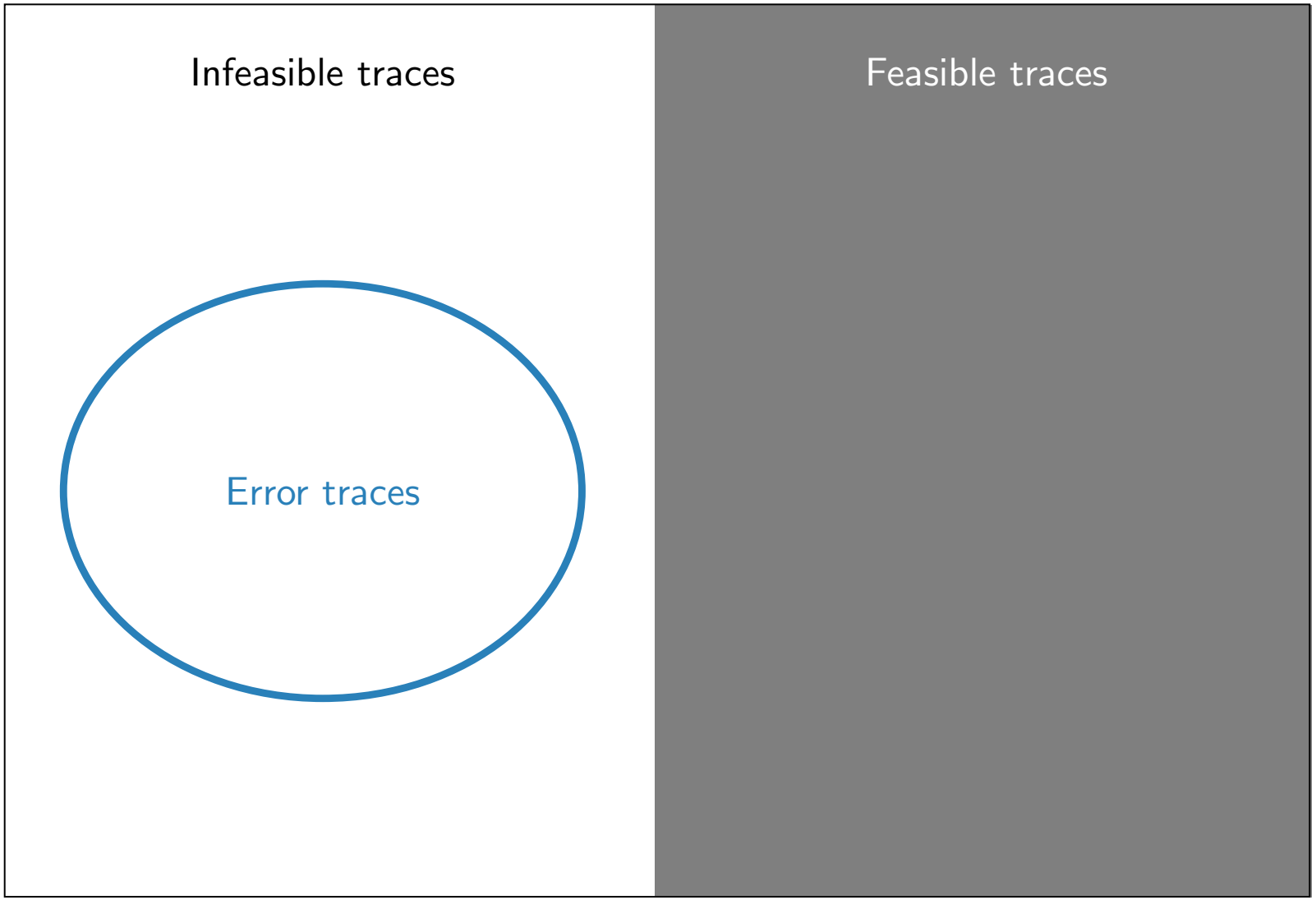


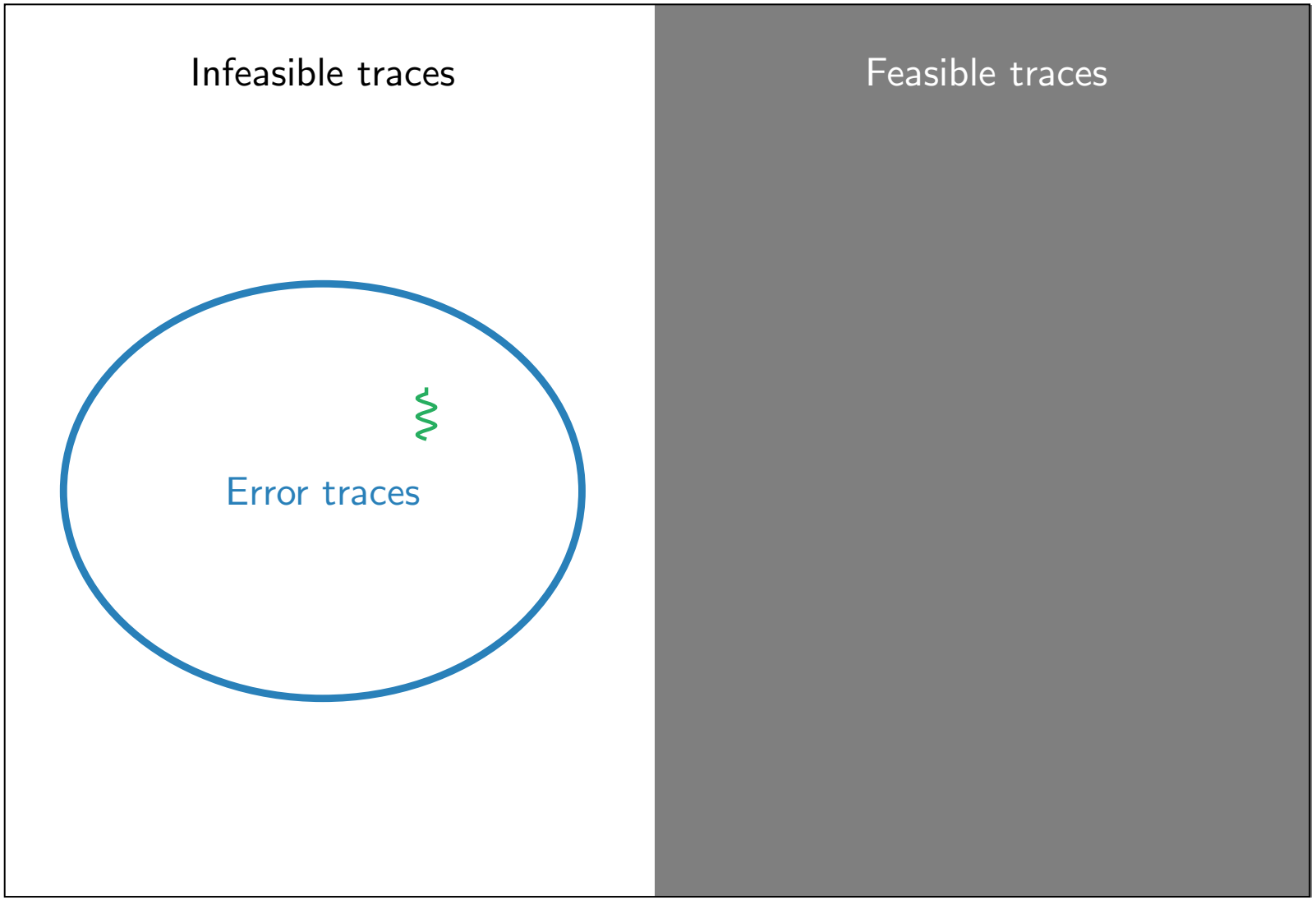
inclusion check succeeds:

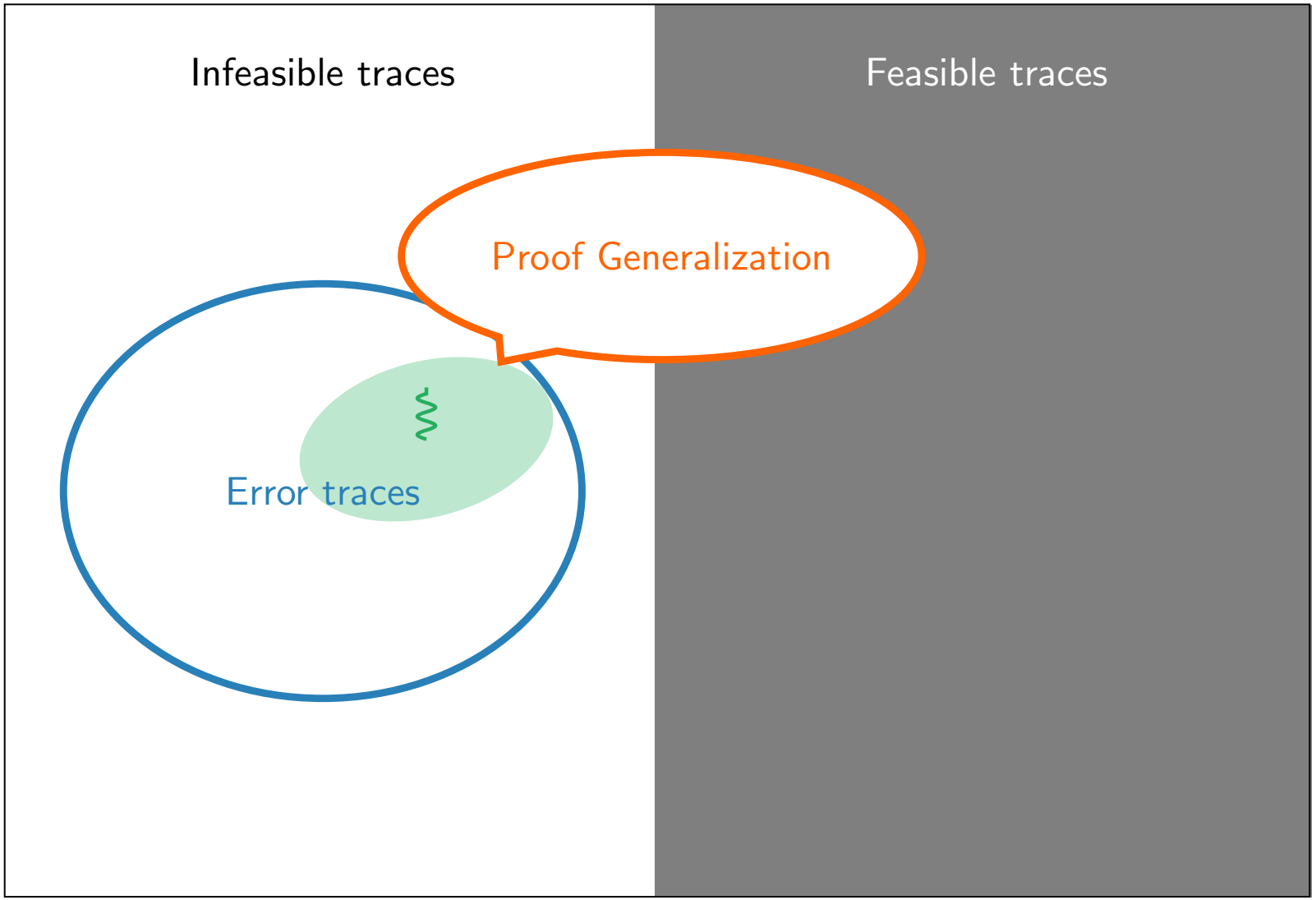
a proof *does* exist for every trace!

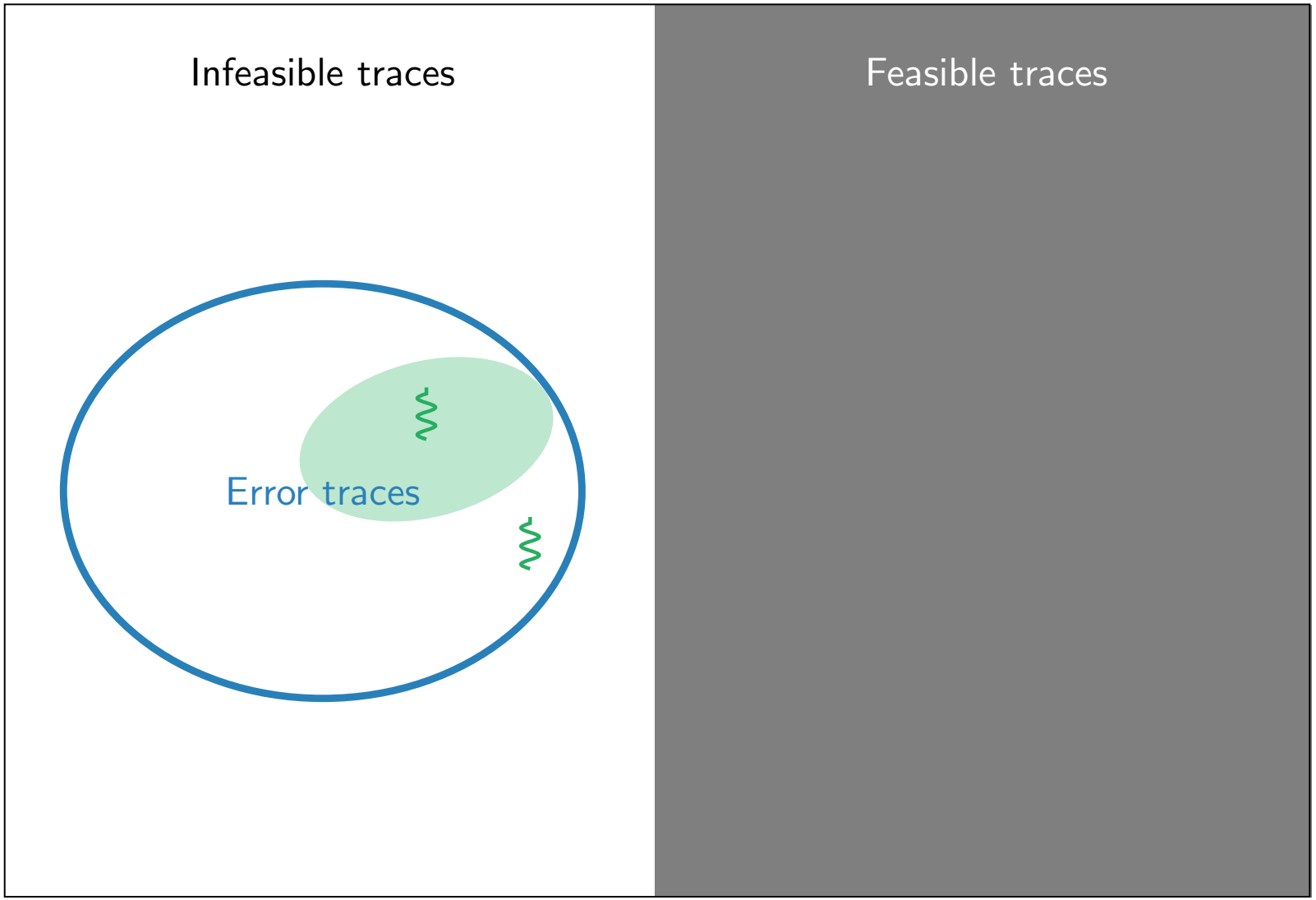

program $\mathcal{P}$
construct $\mathcal{A}_{n+1}$ such that
1. $w \in \mathcal{A}_{n+1}$
2. $\mathcal{A}_{n+1} \subseteq$ { infeasible traces }
yes
$\mathcal{A}_{\mathcal{P}} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$?
w infeasible?
yes
no
no
take w such that
$w \in \mathcal{A}_{\mathcal{P}}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

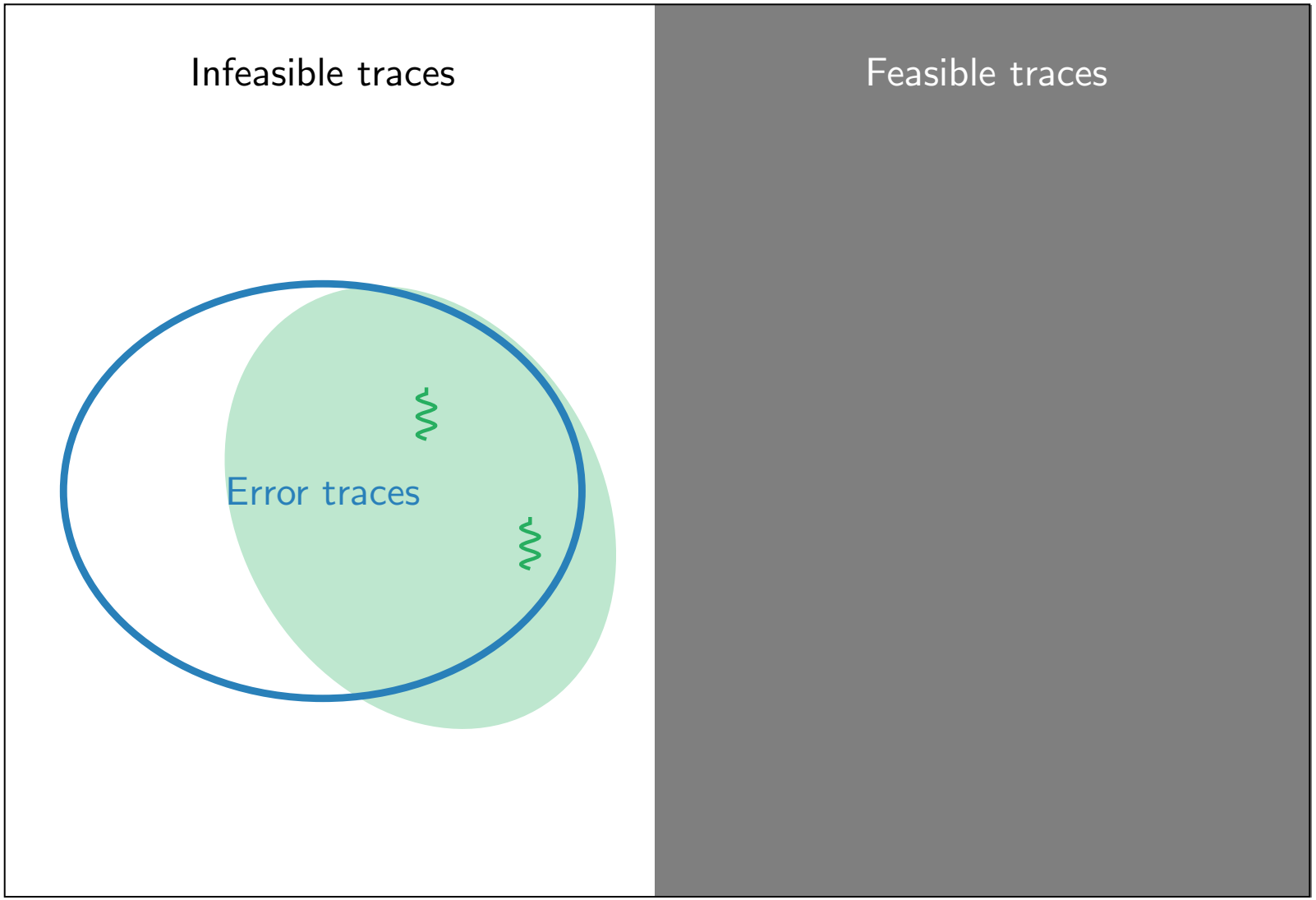

Infeasible traces
Feasible traces
No corresponding executions
At least one corresponding execution

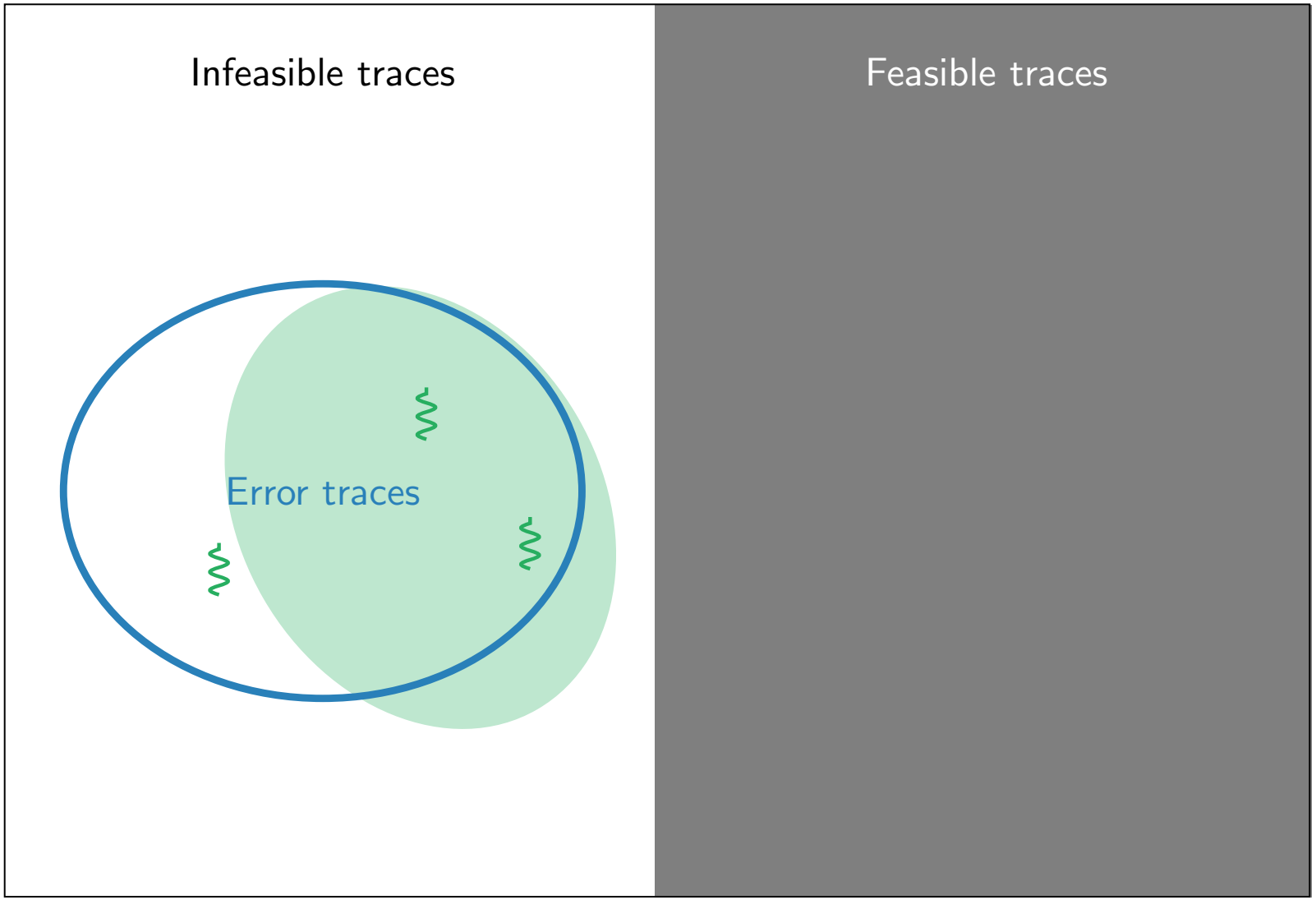

Infeasible traces
Feasible traces
Error traces

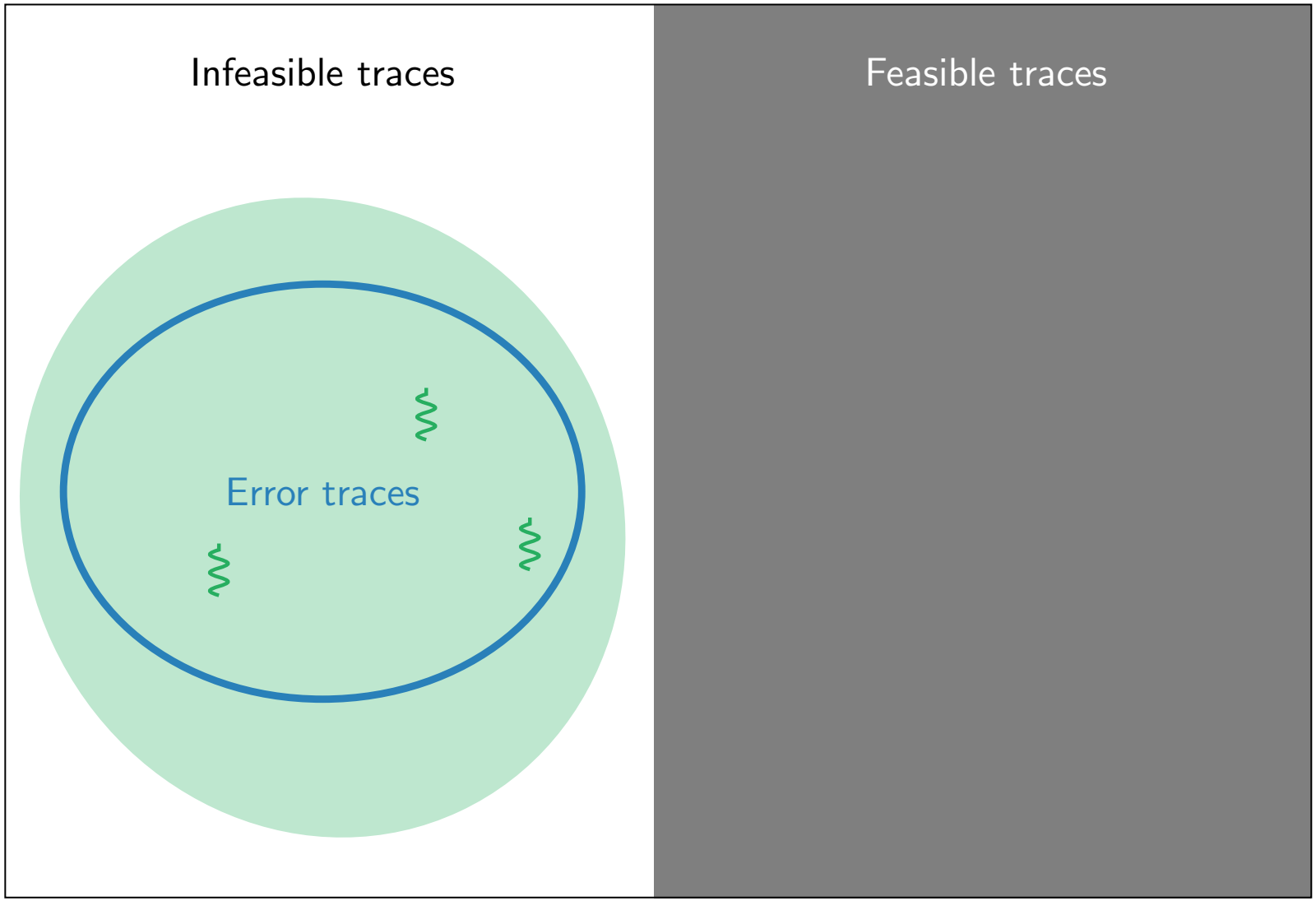

Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Proof Generalization
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces


Infeasible traces
Feasible traces
Error traces

previous example:

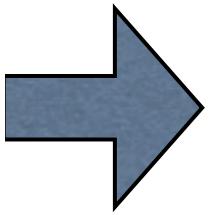
automata from unsatisfiable core
(for proof of infeasibility of error trace)

add self-loop for each irrelevant statement
(does not modify variables in unsatisfiable core)

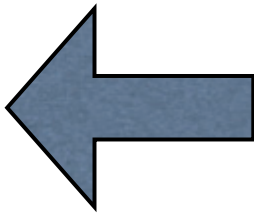
automata constructed from unsatisfiable core

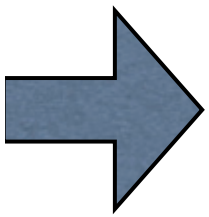
are not sufficient in general

(verification algorithm not complete)

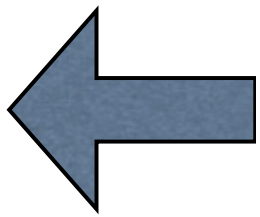


- learn correct programs from unsatisfiability proofs
- learn correct programs from Hoare triples

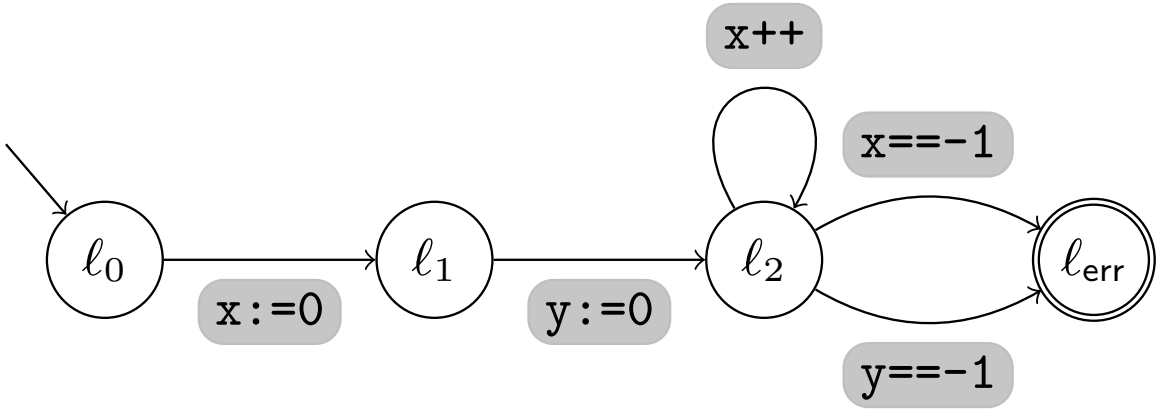




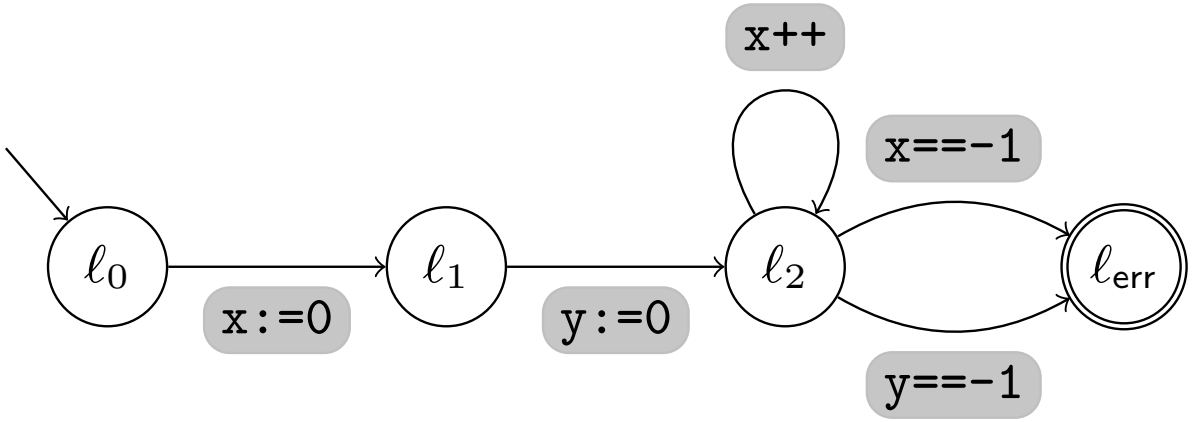
- learn correct programs from unsatisfiability proofs
- learn correct programs from Hoare triples



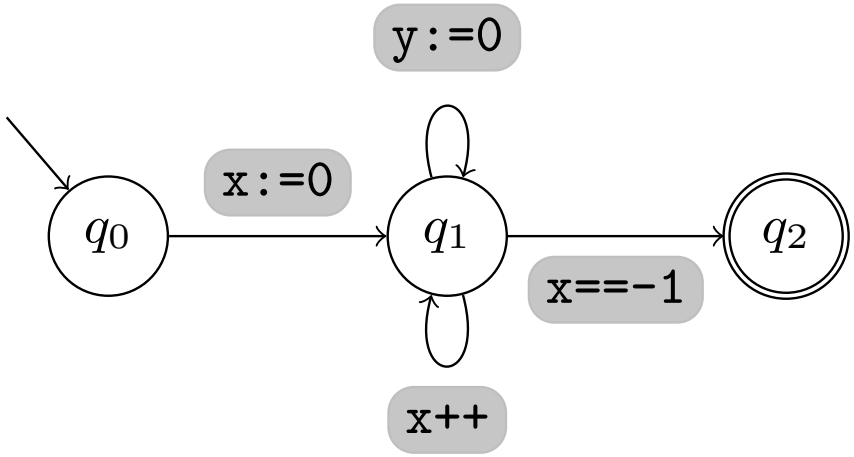
```
ℓ0: x := 0;
ℓ1: y := 0;
ℓ2: while(nondet) {x++;}
    assert(x != -1);
    assert(y != -1);
```

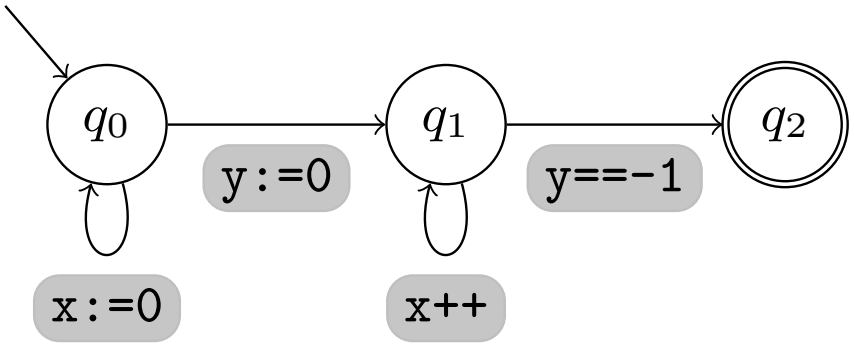


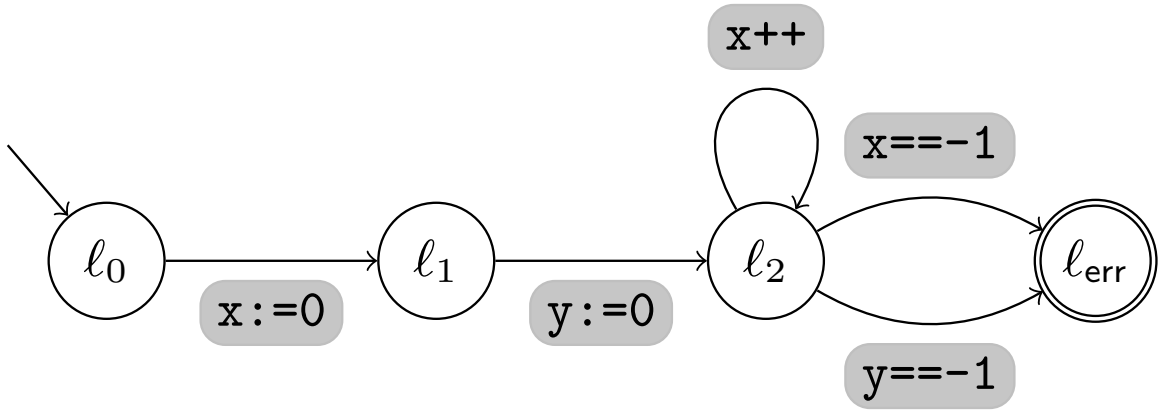
program *P* :



all behaviors of program *P* covered by two programs below:



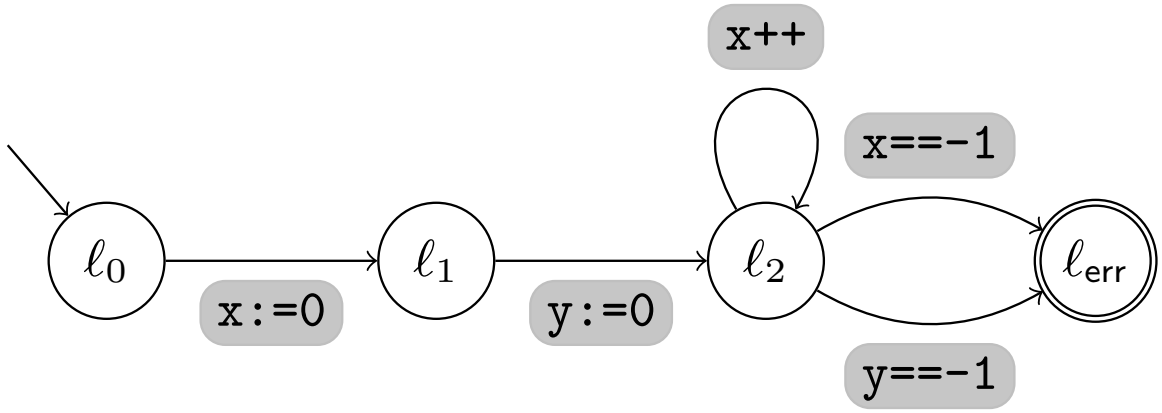





```
x:=0
y:=0
x++
x==-1
```

unsatisfiable core of unsatisfiability proof uses variable *x*

=> program constructed from unsatisfiability proof has no self-loop with statement *x*++ in




```
x:=0
y:=0
x++
x==-1
```

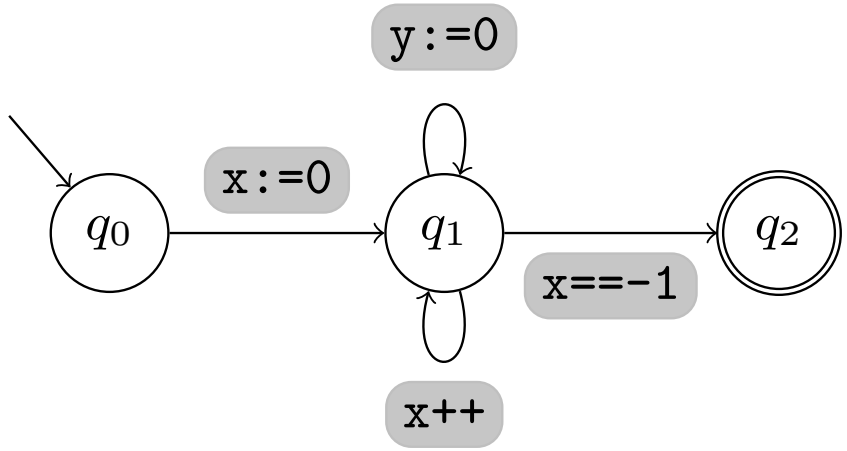
Hoare triples proving infeasibility :

$$\{\ true\ \}\ \texttt{x:=0}\ \{x \geq 0\}$$
$$\{x \geq 0\}\ \texttt{y:=0}\ \{x \geq 0\}$$
$$\{x \geq 0\}\ \texttt{x++}\ \{x \geq 0\}$$
$$\{x \geq 0\}\ \texttt{x==-1}\ \{\ false\ \}$$

infeasibility ⇔ pre/postcondition pair *(true, false)*

Hoare triples $\longmapsto$ correct program

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



correct program

construction of correct program from Floyd-Hoare proof of infeasibility of trace

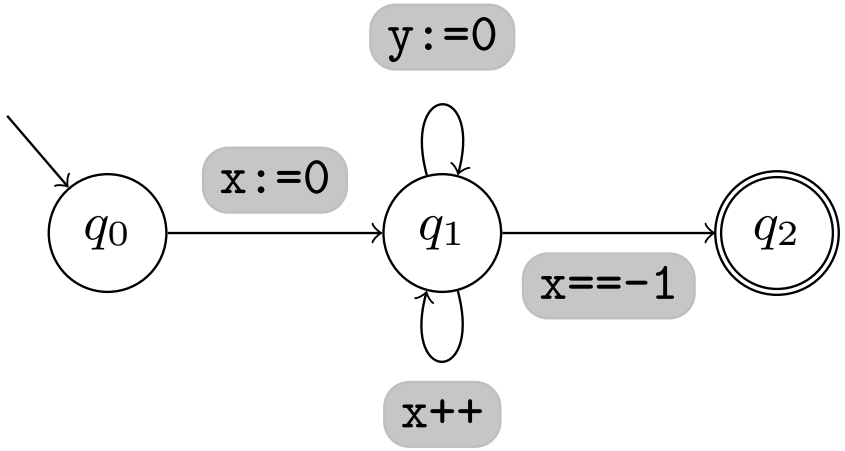
(*remember:* infeasibility ⇔ postcondition *false*)

control flow graph has one node for each assertion, one edge for each Hoare triple

("transition back" = loop, in general not self-loop)

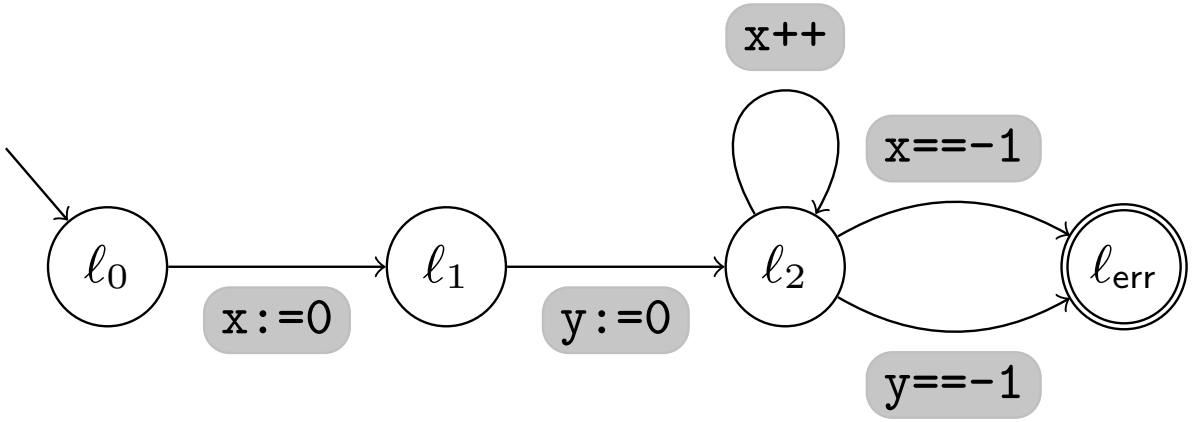
Hoare triples $\longmapsto$ automaton

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

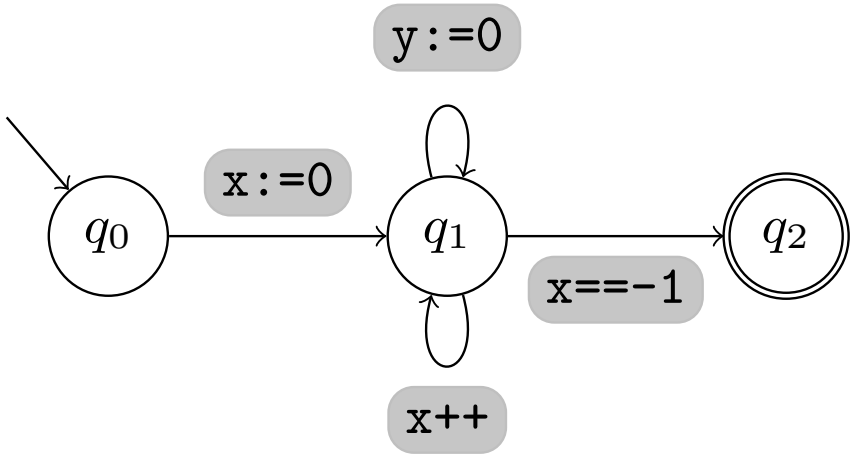


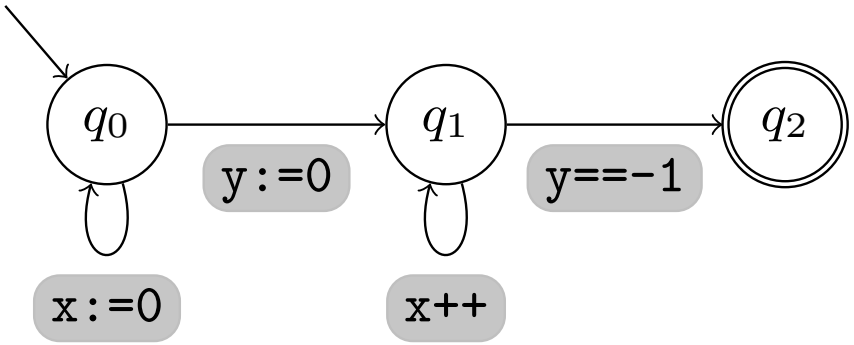
sequencing of Hoare triples $\longmapsto$ run of automaton

program *P* :

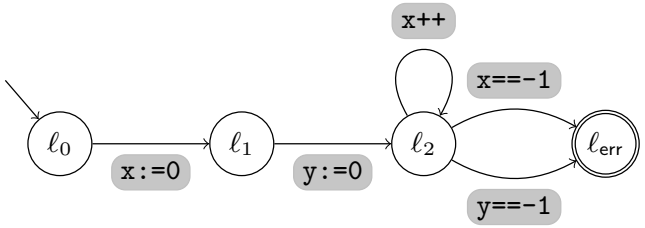


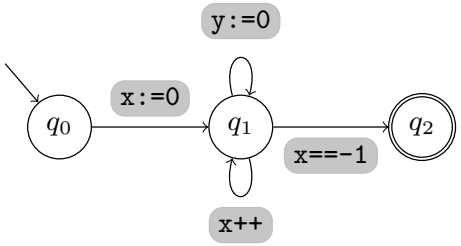
all behaviors of program *P* covered by two programs below:

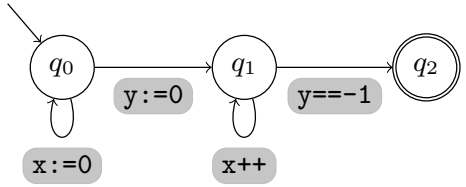




covering check = automata inclusion check

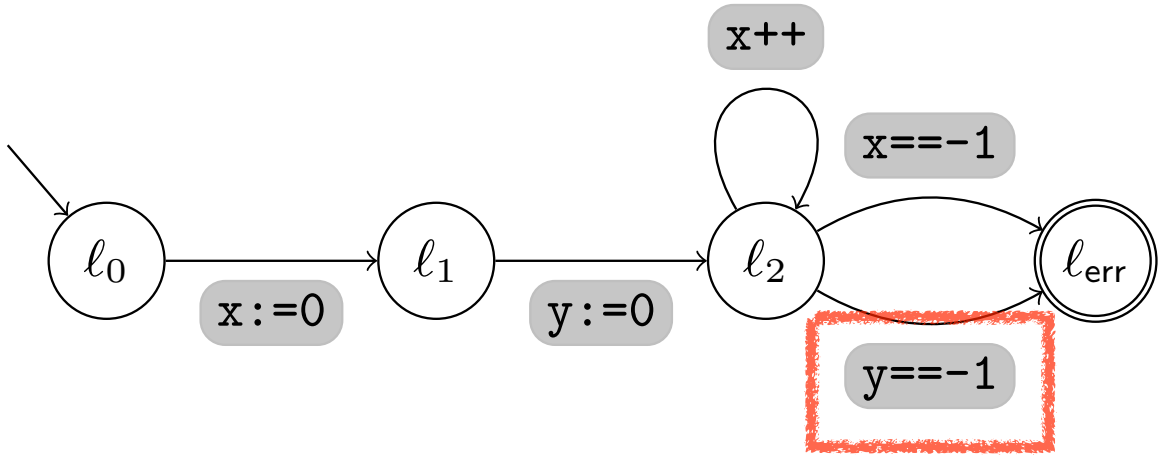






$$\mathcal{P}_{\text{ex2}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$$

second trace



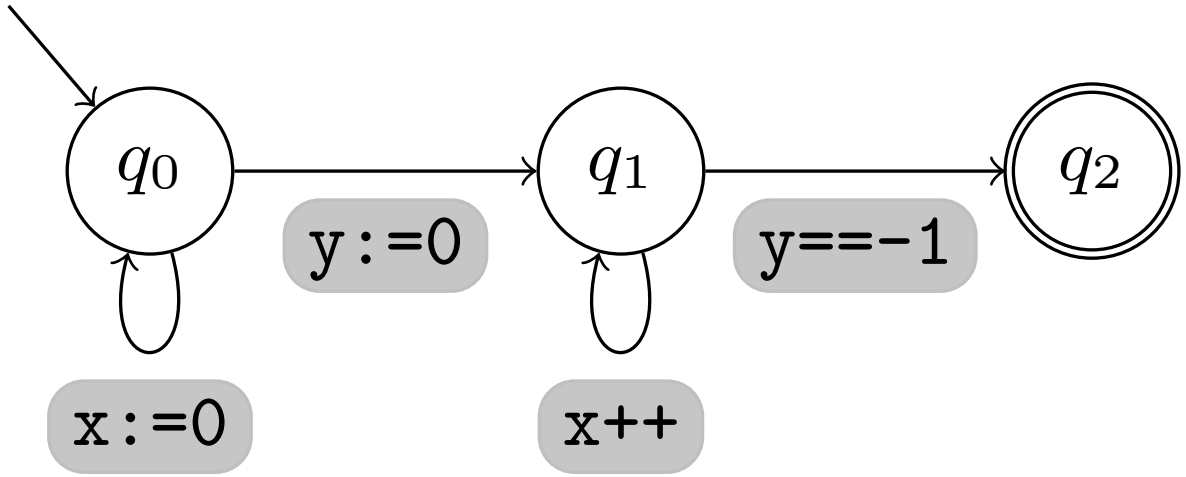
x:=0

y:=0

x++

y==-1

automaton from unsatisfiability core
of infeasibility proof for second trace



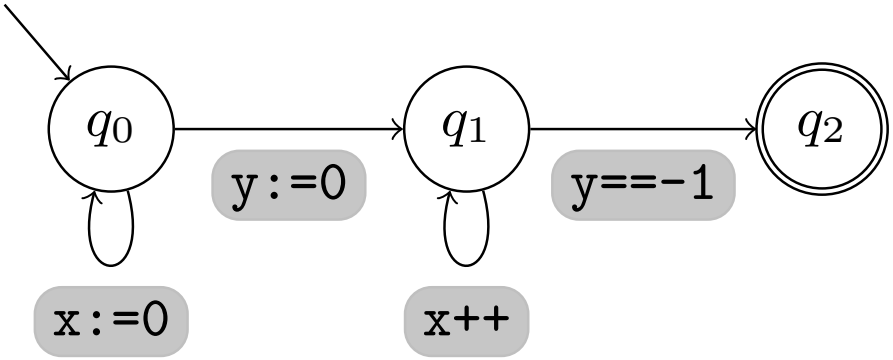
Hoare proof

for infeasibility of second trace

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

automaton from Hoare proof for infeasibility of second trace

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

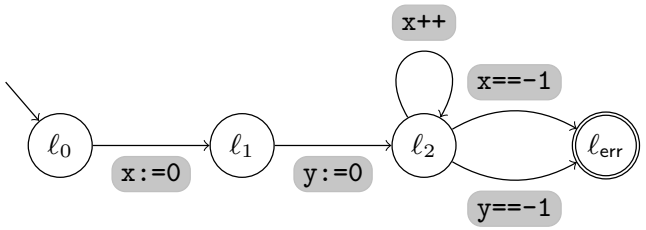


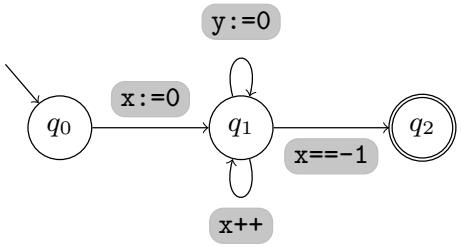
automaton from unsatisfiable core
is a special case of
automaton from Hoare triples
of proof for infeasibility of trace

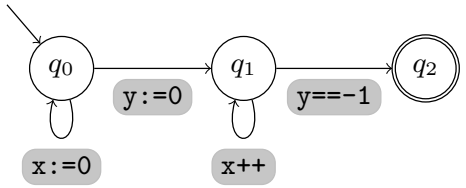
proof for infeasibility of trace
⇒ Hoare triples/assertions exist

"loop invariant: any assertion will do"

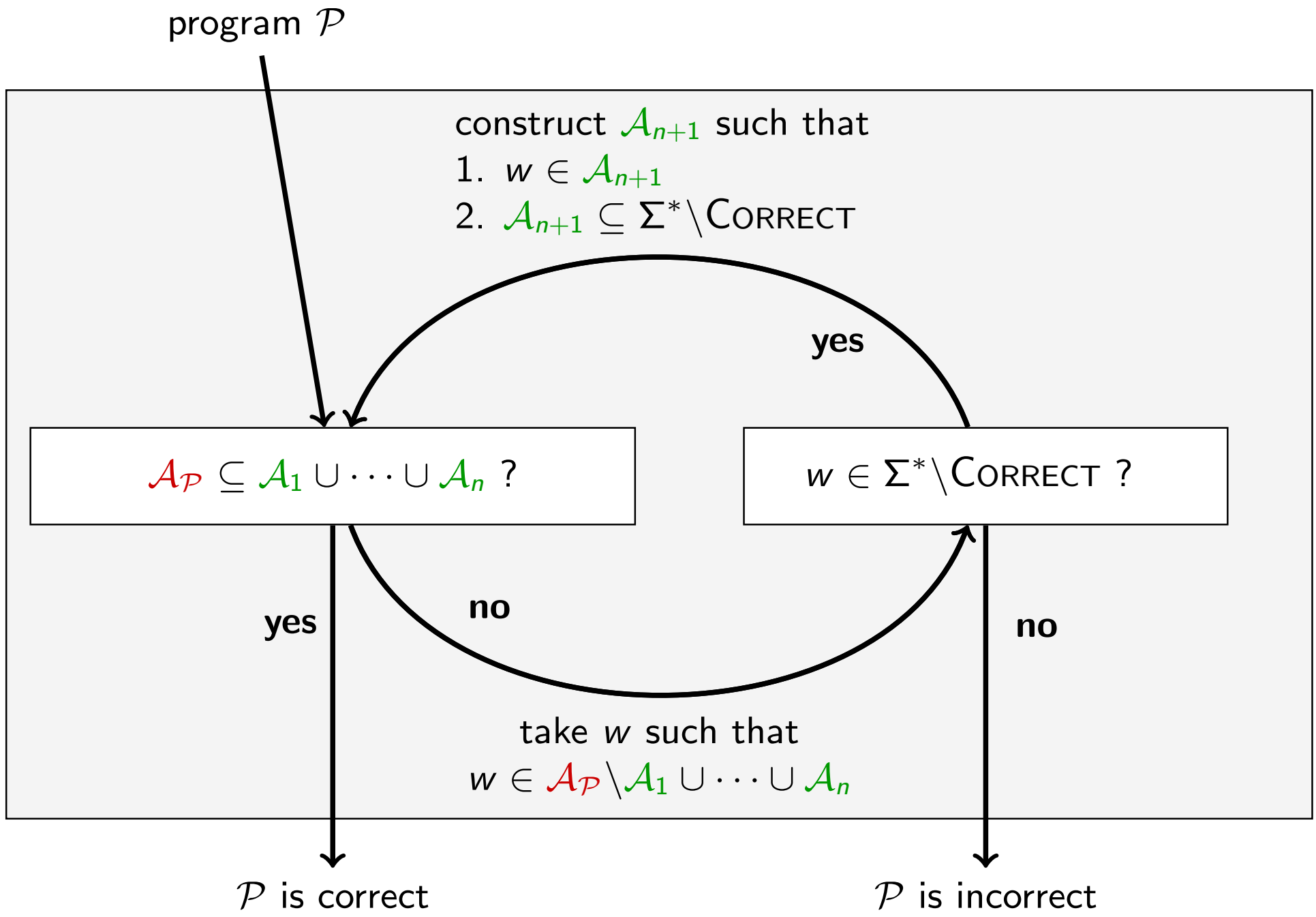
correct programs,
constructed by Hoare proof or by unsatisfiability proof,
sufficient if inclusion check succeeds







$$\mathcal{P}_{\text{ex2}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$$


program $\mathcal{P}$
construct $\mathcal{A}_{n+1}$ such that
1. $w \in \mathcal{A}_{n+1}$
2. $\mathcal{A}_{n+1} \subseteq \Sigma^* \setminus \textsc{Correct}$
yes
$\mathcal{A}_{\mathcal{P}} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$?
$w \in \Sigma^* \setminus \textsc{Correct}$?
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

automated verification

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

- Refinement of Trace Abstraction.
  SAS 2009
- Nested interpolants.
  POPL 2010
- Inductive data flow graphs.
  POPL 2013
- Software Model Checking for People Who Love Automata.
  CAV 2013
- Termination Analysis by Learning Terminating Programs.
  CAV 2014
- Proofs that count.
  POPL 2014
- 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
- Proving Liveness of Parameterized Programs.
  LICS 2016