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
    lock(m);
    if(next + 10 <= len)
    {
        c := next; next := next + 10; end := next;
    }
    else
    {
        c := next; next := next + 10; end := len;
    }
    unlock(m);
    // perform block of tasks
    while (c < end):
        tasks[c] := 0;  // mark task c as started
        ...  // work on the task c
        tasks[c] := 1;  // mark task c as finished
        assert(tasks[c] == 1);  // no other thread has started task c
        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
lock(m);
if(next + 10 <= len)
    { c := next; next := next + 10; end := next; }
else
    { c := next; next := next + 10; end := len; }
unlock(m);

// perform block of tasks
while (c < end):
    tasks[c] := 0; // mark task c as started
    ... // work on the task c
    tasks[c] := 1; // mark task c as finished
    assert(tasks[c] == 1); // no other thread has started task c
    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
lock(m);
if(next + 10 <= len)
    { c := next; next := next + 10; end := next; }  
else
    { c := next; next := next + 10; end := len; }
unlock(m);

threads 1, 2, ..., 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 ⊆ counting automaton
Ultimate Automizer

```
/*@ requires \true;
@ ensures x > 101 || result == 91;
*/

int f91(int x);

int f91(int x) {
    if (x > 100)
        return x - 10;
    else {
        return f91(f91(x+11));
    }
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Ultimate Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>procedure precondition always holds</td>
</tr>
<tr>
<td>21</td>
<td>procedure precondition always holds</td>
</tr>
<tr>
<td>13</td>
<td>procedure postcondition always holds</td>
</tr>
</tbody>
</table>
**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
\ell_0: \text{assume } p \neq 0; \\
\ell_1: \text{while}(n \geq 0) \\
\quad \{ \\
\quad \ell_2: \\
\quad \quad \text{if}(n == 0) \\
\quad \quad \quad \{ \\
\quad \quad \quad \ell_3: \quad p := 0; \\
\quad \quad \quad \} \\
\quad \ell_4: \quad n--; \\
\quad \} \\
\ell_5: 

\text{Fig. 1: Example program}

Example 1: automata from infeasibility proofs

The program \( P_{\text{ex1}} \) in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use \texttt{assert} statements to define the correctness of the program executions. In the example of \( P_{\text{ex1}} \), an incorrect execution would start with a non-zero value for the variable \( p \) and, at some point, enter the body of the while loop when the value of \( p \) is 0 (and the execution of the \texttt{assert} statement fails).

We can argue the correctness of \( P_{\text{ex1}} \) rather directly if we split the executions into two cases, namely according to whether the then branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of \( p \) is never changed and remains non-zero (and the assert statement cannot fail). If the then branch of the conditional is executed, then the value of \( n \) is 0, the statement \( n-- \) decrements the value of \( n \) from 0 to 1, and the while loop will exit directly, without executing the \texttt{assert} statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of \( P_{\text{ex1}} \).

We will describe an execution of \( P_{\text{ex1}} \) through the sequence of statements on the corresponding path in the control flow graph of \( P_{\text{ex1}} \); see Figure 1. The shortest path from \( 0 \) to \( \text{err} \) goes via \( 1 \) and \( 2 \). The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not
\(\ell_0:\) assume \(p \neq 0;\)

\(\ell_1: while(n \geq 0)\)
\{
    \(\ell_2: \) assert \(p \neq 0;\)
    if \(n == 0\)
    \{
        \(\ell_3: \) \(p := 0;\)
    \}
    \(\ell_4: \) \(n--;\)
\}

\(\ell_5:\)
$\ell_0$: assume $p \neq 0$;

$\ell_1$: while($n \geq 0$)
{
  $\ell_2$: assert $p \neq 0$;
  if($n == 0$)
  {
    $\ell_3$: $p := 0$;
  }
  $\ell_4$: $n --$;
}

$\ell_5$: 

no execution violates assertion  =  no execution reaches error location
Example 1: automata from infeasibility proofs

The program $P_{ex1}$ in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use `assert` statements to define the correctness of the program executions. In the example of $P_{ex1}$, an incorrect execution would start with a non-zero value for the variable $p$ and, at some point, enter the body of the while loop when the value of $p$ is 0 (and the execution of the `assert` statement fails).

We can argue the correctness of $P_{ex1}$ rather directly if we split the executions into two cases, namely according to whether the `then` branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of $p$ is never changed and remains non-zero (and the assert statement cannot fail). If the `then` branch of the conditional is executed, then the value of $n$ is 0, the statement $n--$ decrements the value of $n$ from 0 to 1, and the while loop will exit directly, without executing the `assert` statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $P_{ex1}$.

We will describe an execution of $P_{ex1}$ through the sequence of statements on the corresponding path in the control flow graph of $P_{ex1}$; see Figure 1. The shortest path from 0 to `err` goes via 1 and 2. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not automaton

alphabet: {statements}
Example 1: automata from infeasibility proofs

The program \( P_{\text{ex1}} \) in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use \textit{assert} statements to define the correctness of the program executions. In the example of \( P_{\text{ex1}} \), an incorrect execution would start with a non-zero value for the variable \( p \) and, at some point, enter the body of the while loop when the value of \( p \) is 0 (and the execution of the \textit{assert} statement fails).

We can argue the correctness of \( P_{\text{ex1}} \) rather directly if we split the executions into two cases, namely according to whether the then branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of \( p \) is never changed and remains non-zero (and the assert statement cannot fail). If the then branch of the conditional is executed, then the value of \( n \) is 0, the statement \( n-- \) decrements the value of \( n \) from 0 to 1, and the while loop will exit directly, without executing the \textit{assert} statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of \( P_{\text{ex1}} \).

We will describe an execution of \( P_{\text{ex1}} \) through the sequence of statements on the corresponding path in the control flow graph of \( P_{\text{ex1}} \); see Figure 1. The shortest path from \( l_0 \) to \( l_{\text{err}} \) goes via \( l_1 \) and \( l_5 \). The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not

\[
(p \neq 0) \\
(n \geq 0) \\
(p == 0)
\]
infeasible trace

\[ x := 1 \ ; \ x == -1 \ ; \]

unsatisfiable formula

\[ x = 1 \land x = -1 \]

\[ x' = 1 \land x' = -1 \]
Example 1: automata from infeasibility proofs

The program $P_{ex1}$ in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use \texttt{assert} statements to define the correctness of the program executions. In the example of $P_{ex1}$, an incorrect execution would start with a non-zero value for the variable $p$ and, at some point, enter the body of the while loop when the value of $p$ is 0 (and the execution of the \texttt{assert} statement fails).

We can argue the correctness of $P_{ex1}$ rather directly if we split the executions into two cases, namely according to whether the \texttt{then} branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of $p$ is never changed and remains non-zero (and the \texttt{assert} statement cannot fail). If the \texttt{then} branch of the conditional is executed, then the value of $n$ is 0, the statement $n--$ decrements the value of $n$ from 0 to 1, and the while loop will exit directly, without executing the \texttt{assert} statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $P_{ex1}$.

We will describe an execution of $P_{ex1}$ through the sequence of statements on the corresponding path in the control flow graph of $P_{ex1}$; see Figure 1. The shortest path from $l_0$ to $l_{err}$ goes via $l_1$ and $l_2$. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not $(p \neq 0)$ $(n \geq 0)$ $(p == 0)$ $(p \neq 0)$ $(p==0)$.
(p \neq 0)

(p==0)
\(q_0\) with

\[p \neq 0\]

\[p = 0\]

\(q_1\)

\(q_2\)

\(\text{(p != 0)}\)

\(\text{(p==0)}\)
We construct the automaton $A_1$ in Figure 2 which recognizes the set of all sequences of statements that contain $p \neq 0$ and $p = 0$ without an update of $p$ in between (and with any statements before or after). I.e., $A_1$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of $p \neq 0$ and $p = 0$).

A sequence of statements is not accepted by $A_1$ if it contains $p \neq 0$ and $p = 0$ with an update of $p$ in between. The shortest path from `0` to `err` with such a sequence of statements goes from `2` to `err` after it has gone from `2` to `3` once before. The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the `assume` statement $n = 0$, the update statement $n --$, and the `assume` statement $n \geq 0$ unless there is an (other) update of $n$ between $n = 0$ and $n --$ or between $n --$ and $n \geq 0$.

We construct the automaton $A_2$ depicted in Figure 2 which recognizes the set of all sequences of statements that contain the statements $n = 0$, $n --$, and $n \geq 0$ without an update of $n$ in between (and with any statements before or after). I.e., $A_2$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts $n = 0$, $n = n_0 = n_1$, and $n_0 = 0$).

To summarize, we have twice taken a path from `0` to `err`, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set $(p \neq 0)$ and $(p = 0)$.
automaton constructed from unsatisfiability proof

![Automaton Diagram]

accepts all traces with the same unsatisfiability proof
The program

\[
\begin{align*}
\text{while}(n\geq 0) & \quad \\
\text{assume} & \quad p \neq 0; \\
\text{assert} & \quad p \neq 0; \\
\text{n--} & \quad \\
\end{align*}
\]

We will describe an execution of \( \text{ex1} \) that starts with a non-zero value for the variable \( p \) and, at some point, enters the body of the while loop when the value of \( n \) is 0, the statement

\[
\text{assert} p \neq 0;
\]

is executed at least once during the execution or it does not. If not, then

\[
\text{assert} p \neq 0;
\]

the inconsistency of \( n \) fails.

In our setting we use automata as an expressive means to characterize di
erent cases of execution paths. For another, instead of first fixing

\[
\text{case split and then constructing the corresponding correctness arguments},
\]

we can construct an automaton for a given correctness argument so that the shortest path from

\[
\text{on the corresponding path in the automaton characterizes the case of exactly the executions for which the cor-}
\]

we can construct an automaton for a given correctness argument so that the shortest path from

does a proof exist for every trace ?
Incremental Construction

A₁,..., Aⁿ ` a la CEGAR

Program P

Construct A_{n+1} such that
1. \( w \in A_{n+1} \)
2. \( A_{n+1} \subseteq \{ \text{infeasible traces} \} \)

\( A_P \subseteq A₁ \cup \cdots \cup Aₙ \)?

Yes

No

\( w \) infeasible?

Yes

No

Take \( w \) such that
\( w \in A_P \setminus A₁ \cup \cdots \cup Aₙ \)

\( P \) is correct

\( P \) is incorrect
Example 1: automata from infeasibility proofs

The program $P_{ex1}$ in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use assert statements to define the correctness of the program executions. In the example of $P_{ex1}$, an incorrect execution would start with a non-zero value for the variable $p$ and, at some point, enter the body of the while loop when the value of $p$ is 0 (and the execution of the assert statement fails).

We can argue the correctness of $P_{ex1}$ rather directly if we split the executions into two cases, namely according to whether the then branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of $p$ is never changed and remains non-zero (and the assert statement cannot fail). If the then branch of the conditional is executed, then the value of $n$ is 0, the statement $n--$ decrements the value of $n$ from 0 to $1$, and the while loop will exit directly, without executing the assert statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $P_{ex1}$.

We will describe an execution of $P_{ex1}$ through the sequence of statements on the corresponding path in the control flow graph of $P_{ex1}$; see Figure 1. The shortest path from $l_0$ to $l_{err}$ goes via $l_1$ and $l_5$. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not $(p != 0)$ $(n >= 0)$ $(n == 0)$ $(p := 0)$ $(n--)$ $(n >= 0)$ $(p == 0)$

new trace:
Example 1: automata from infeasibility proofs

The program \( P_{ex1} \) in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use `assert` statements to define the correctness of the program executions. In the example of \( P_{ex1} \), an incorrect execution would start with a non-zero value for the variable \( p \) and, at some point, enter the body of the while loop when the value of \( p \) is 0 (and the execution of the `assert` statement fails).

We can argue the correctness of \( P_{ex1} \) rather directly if we split the executions into two cases, namely according to whether the `then` branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of \( p \) is never changed and remains non-zero (and the `assert` statement cannot fail). If the `then` branch of the conditional is executed, then the value of \( n \) is 0, the statement \( n-- \) decrements the value of \( n \) from 0 to 1, and the while loop will exit directly, without executing the `assert` statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of \( P_{ex1} \).

We will describe an execution of \( P_{ex1} \) through the sequence of statements on the corresponding path in the control flow graph of \( P_{ex1} \); see Figure 1. The shortest path from \( l_0 \) to \( l_{err} \) goes via \( l_1 \) and \( l_2 \). The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not

\[
\begin{align*}
(p \neq 0) \\
(n \geq 0) \\
(n = 0) \\
(p := 0) \\
(n--) \\
(n \geq 0) \\
(p = 0)
\end{align*}
\]
(n == 0)

(n--)

(n >= 0)
this path is shortest path from on the corresponding path in the rectness argument applies. We will next illustrate this in the example of automaton characterizes the case of exactly the executions for which the cor-
we can construct an automaton for a given correctness argument so that the case split and then constructing the corresponding correctness arguments, characterize di
use automata. For one thing, we can use automata as an expressive means to n
gets executed at least once during the execution or it does not. If not, then while loop when the value of with a non-zero value for the variable program executions. In the example of setting. In our setting we use The program Example 1: automata from infeasibility proofs `3
`2
`1
`0
is 0, the statement:
assert p != 0;
We will describe an execution of We can infer a case split like the one above automatically. The key is to We can argue the correctness of different cases of execution paths. For another, instead of first fixing ex1 branch of the conditional is executed, then the value of n--
and, at some point, enter the body of the possible to execute the p
statements to define the correctness of the p
error of sequences of statements that are infeasible for the same reason as above (i.e., an edge labelled with (an edge labelled with
p := 0
err
n==0
n--
assert p != 0;
p: =0 ;

p
is never changed and remains non-zero (and the assert statement after). I.e.,
ex1
assert
p

p
3
2
1
0

{n >= 0} while(n >= 0) {p != 0;

{n < 0} p := 0

{n == 0} n--

{n > 0} n >= 0

{n != 0} while(n != 0) {p != 0;
 while loop when the value of p

{n==0} n--

{n!}= 0

{n >= 0}

{n <= 0}

{n != 0}

{n == 0}
this path is the shortest path from on the corresponding path in the rectness argument applies. We will next illustrate this in the example of we can construct an automaton for a given correctness argument so that the case split and then constructing the corresponding correctness arguments, characterize di…

Example 1: automata from infeasibility proofs

\[ \begin{align*}
\text{let } p_0 := \text{true}; \\
\text{while}(n \geq 0) \\
\text{assume } p \neq 0; \\
\text{if}(n == 0) \\
\text{assert } p \neq 0; \\
\text{n--} \\
\text{n >= 0} \\
\text{p := 0;}
\end{align*} \]

We can infer a case split like the one above automatically. The key is to We can argue the correctness of)

4
3
1
0

We construct the automaton

\[ \begin{align*}
A &\text{ recognizes the set of all sequences of statements that contain the statements to define the correctness of the } P \text{ infeasible} \\
&\text{set of all sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts before). I.e., } n \geq 0 \\
&\text{an update of } n \neq 0 \\
&\text{p \neq 0} \\
&\text{and the } n==0 \\
&\text{with such a } p=0 \\
&\text{and } n-- \\
&\text{in between (and with any statements before or after). I.e., } n \geq 0 \\
&\text{sequences of statements that contain } p \neq 0 \\
&\text{and the } p==0 \\
&\text{in } P \text{ex1 after). I.e., } n \geq 0 \\
&\text{sequences of statements that are a proof of correctness for } P \text{ex1} \\
&\text{rather directly if we split the executions through the sequence of statements } P \text{ex1} \\
&\text{for the same reason as above (i.e., the inconsistency of the three conjuncts }
\]

\[ \begin{align*}
&\text{p: =0 ;} \\
&\text{p --} \\
&\text{p := 0} \\
&\text{p != 0} \\
&\text{p == 0} \\
&\text{p == 0} \\
\end{align*} \]

\[ \begin{align*}
&\text{impossible. If the value of } p \text{ gets executed at least once during the execution or it does not. If not, then } \\
&\text{the value of } p \text{ is never changed and remains non-zero (and the assert statement after). I.e., } n \geq 0 \\
&\text{wrong are between (and with any statements before or after). I.e., } n \geq 0 \\
&\text{various cases of execution paths. For another, instead of first fixing } n \geq 0 \\
&\text{to } n \geq 0 \\
&\text{rather directly if we split the executions through the sequence of statements } P \text{ex1} \\
&\text{for the same reason as above (i.e., the inconsistency of the three conjuncts }
\]

\[ \begin{align*}
&\text{ex1} \\
&\text{infeasible} \\
&\text{goes via } P \text{ex1 and, at some point, enter the body of the } P \text{ex1}
\end{align*} \]

\[ \begin{align*}
&\text{Fig. 1: Example program} \\
&\text{Fig. 2: Automata} \\
\end{align*} \]
automaton constructed from unsatisfiability proof

\[
\begin{array}{c}
p_0 \xrightarrow{\Sigma} p_1 \xrightarrow{\Sigma \setminus \{n--\}} p_2 \xrightarrow{\Sigma \setminus \{n--\}} p_3 \xrightarrow{\Sigma} \\
\end{array}
\]

\[
\begin{align*}
&n == 0 \\
&n-- \\
&n >= 0 \\
\end{align*}
\]

\[
\begin{align*}
&(p \neq 0) \\
&(n \geq 0) \\
&(n == 0) \\
&(p := 0) \\
&(n--) \\
&(n \geq 0) \\
&(p == 0)
\end{align*}
\]

accepts all traces with the same unsatisfiability proof
does a proof exist for every trace?
Incremental Construction

A_1, ..., A_n ` a la CEGAR

program P

\begin{itemize}
  \item P is correct
  \item P is incorrect
\end{itemize}

A_P \subseteq A_1 \cup \cdots \cup A_n \ ?

w infeasible?

\begin{itemize}
  \item yes
  \item no
\end{itemize}

\begin{itemize}
  \item yes
  \item no
\end{itemize}

\begin{itemize}
  \item take w such that
  \item w \in A_P \setminus A_1 \cup \cdots \cup A_n
\end{itemize}

\begin{itemize}
  \item \{ infeasible traces \}
\end{itemize}

\begin{itemize}
  \item \{ infeasible traces \}
\end{itemize}
automata constructed from unsatisfiable core

are not sufficient in general

(verification algorithm not complete)
proof spaces

- automata from unsatisfiability proofs
- proof spaces
\( \ell_0: \ x := 0; \)
\( \ell_1: \ y := 0; \)
\( \ell_2: \ \text{while(\text{nondet}) \{x++;\}} \)
  \hspace{1em} \text{assert}(x \neq -1);
  \hspace{1em} \text{assert}(y \neq -1);
We use them in the same way as above in order to construct the automaton paths that reach the error location via the edge labeled with proof. The automaton can have arbitrary loops. In contrast, an automaton constructed as Floyd-Hoare automaton has only self-loops.

In our implementation, the set of Hoare triples comes from an interpolating SMT solver which generates the assertion corresponding to one path from the initial state. Where does the set of Hoare triples come from? In this example, it may come from taking the edge labeled \( x = -1 \) to \( x = 0 \), the (only) final state.

The automaton accepts a word exactly if the word labels a path from the initial state to a final state. Thus, the check amounts to checking the inclusion between automata, which is in general undecidable. To rephrase our summary in the terminology of automata, we have twice taken a word accepted by an automaton which we also call a word accepted by the automaton.

The example of the program in Figure 3 shows that sometimes a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion \( x \neq -1 \) holds after the update \( x := 0 \), and determine that \( x = -1 \) is not reachable.

It has three states, one for each assertion: the initial state \( q_0 \) for \( x = 0 \), the (only) final state \( q_2 \) for \( x = 1 \), and \( q_1 \) for \( y = 0 \). The only transitions are labeled with \( x = 0 \), \( y = 0 \), \( x = -1 \), and \( y = -1 \). The set of such sequences is the language recognized by an automaton which we also call a word accepted by the automaton.

We must base the construction of the automaton on a more powerful form of reasoning: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion \( x = 0 \) holds after the update \( x := 0 \), and determine that \( x = -1 \) is not reachable.

The example of the program in Figure 3 shows that sometimes a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion \( x = 0 \) holds after the update \( x := 0 \), and determine that \( x = -1 \) is not reachable.

The view of a program as an automaton over the alphabet of statements may generalize to any set of Hoare triples. The resulting automaton can, however, check a condition which is stronger, namely that all sequences of statements on paths from \( q_0 \) to \( q_2 \) fall into one of the two cases? – The corresponding decision problem is undecidable. We have constructed an automaton which recognizes the set of all words for which the statements in the sequence \( x := 0 \), \( x := 0 \), \( y := 0 \), \( x := 0 \) hold after the update \( x := 0 \) and \( y := 0 \). The automaton thus characterizes a case of executions in the sense discussed above.

The example of the program in Figure 3 shows that sometimes a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion \( x = 0 \) holds after the update \( x := 0 \), and determine that \( x = -1 \) is not reachable.

The view of a program as an automaton over the alphabet of statements may generalize to any set of Hoare triples. The resulting automaton can, however, check a condition which is stronger, namely that all sequences of statements on paths from \( q_0 \) to \( q_2 \) fall into one of the two cases? – The corresponding decision problem is undecidable. We have constructed an automaton which recognizes the set of all words for which the statements in the sequence \( x := 0 \), \( x := 0 \), \( y := 0 \), \( x := 0 \) hold after the update \( x := 0 \) and \( y := 0 \). The automaton thus characterizes a case of executions in the sense discussed above.
of sequences of statements that are infeasible for the specific reason. The two automata thus characterize a case of executions in the sense discussed above. Can one automatically check that every possible execution of $P_{ex1}$ falls into one of the two cases? – The corresponding decision problem is undecidable. We can, however, check a condition which is stronger, namely that all sequences of statements on paths from $`0$ to $`err$ in the control flow graph of $P_{ex1}$ fall into one of the two cases (the condition is stronger because not every such path corresponds to a possible execution). The set of such sequences is the language recognized by an automaton which we also call $P_{ex1}$ (recall that an automaton accepts a word exactly if the word labels a path from the initial state to a final state). Thus, the check amounts to checking the inclusion between automata, namely $P_{ex1} \sqsubseteq A_1 \sqcup A_2$.

To rephrase our summary in the terminology of automata, we have twice taken a word accepted by the automaton $P_{ex1}$, we have analyzed the reason of the infeasibility of the word (i.e., the corresponding sequence of statements), and we have constructed an automaton which recognizes the set of all words for which the same reason applies.

The view of a program as an automaton over the alphabet of statements may take some time to get used to because the view ignores the operational meaning of the program.

Example 2: automata from sets of Hoare triples

It is “easy” to justify the construction of the automata $A_1$ and $A_2$ in Example 1: the infeasibility of a sequence of statements (such as the sequence $p\neq 0 \ p==0$) is preserved if one adds statements that do not modify any of the variables of the statements in the sequence (here, the variable $p$).

The example of the program $P_{ex2}$ in Figure 3 shows that sometimes a more involved justification is required. The sequence of the two statements $x:=0$ and $x==-1$ (which labels a path from $`0$ to $`err$) is infeasible. However, the statement $x++$ does modify the variable that appears in the two statements. So how can we account for the paths that loop in $`2$ taking the edge labeled $x++$ one or more times? We need to construct an automaton that covers the case of those paths, but we cannot base the construction solely on infeasibility (as we did in Example 1).

We must base the construction of the automaton on a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion $x\geq 0$ holds after the update $x:=0$, that it is invariant under the updates $y:=0$ and $x++$, and that it blocks the execution of the assume statement $x==-1$.

\[
\{ true \} \ x:=0 \ { x \geq 0 } \\
\{ x \geq 0 \} \ y:=0 \ { x \geq 0 } \\
\{ x \geq 0 \} \ x++ \ { x \geq 0 } \\
\{ x \geq 0 \} \ x==\!-\!1 \ { false } \\
\]

infeasibility $\iff$ pre/postcondition pair $(true, false)$
inference rule for sequencing

\[
\{p\} \ s \ \{q'\} \\
\{q'\} \ s' \ \{q\} \\
\hline \\
\{p\} \ s \ ; \ s' \ \{q\}
\]
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}”
of sequences of statements that are infeasible for the specific reason. The two automata thus characterize a case of executions in the sense discussed above. Can one automatically check that every possible execution of $P_{\text{ex1}}$ falls into one of the two cases? – The corresponding decision problem is undecidable. We can, however, check a condition which is stronger, namely that all sequences of statements on paths from $`0$ to $`\text{err}$ in the control flow graph of $P_{\text{ex1}}$ fall into one of the two cases (the condition is stronger because not every such path corresponds to a possible execution). The set of such sequences is the language recognized by an automaton which we also call $P_{\text{ex1}}$ (recall that an automaton accepts a word exactly if the word labels a path from the initial state to a final state). Thus, the check amounts to checking the inclusion between automata, namely $P_{\text{ex1}} \sqsubset A_1 \sqcup A_2$.

To rephrase our summary in the terminology of automata, we have twice taken a word accepted by the automaton $P_{\text{ex1}}$, we have analyzed the reason of the infeasibility of the word (i.e., the corresponding sequence of statements), and we have constructed an automaton which recognizes the set of all words for which the same reason applies.

The view of a program as an automaton over the alphabet of statements may take some time to get used to because the view ignores the operational meaning of the program.

Example 2: automata from sets of Hoare triples

It is "easy" to justify the construction of the automata $A_1$ and $A_2$ in Example 1: the infeasibility of a sequence of statements (such as the sequence $p!=0 \ p==0$) is preserved if one adds statements that do not modify any of the variables of the statements in the sequence (here, the variable $p$).

The example of the program $P_{\text{ex2}}$ in Figure 3 shows that sometimes a more involved justification is required. The sequence of the two statements $x:=0$ and $x==-1$ (which labels a path from $`0$ to $`\text{err}$) is infeasible. However, the statement $x++$ does modify the variable that appears in the two statements. So how can we account for the paths that loop in $`2$ taking the edge labeled $x++$ one or more times? We need to construct an automaton that covers the case of those paths, but we cannot base the construction solely on infeasibility (as we did in Example 1).

We must base the construction of the automaton on a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion $x = 0$ holds after the update $x:=0$, that it is invariant under the updates $y:=0$ and $x++$, and that it blocks the execution of the assume statement $x==-1$.

$$
\begin{align*}
\{ \text{true} \} & \ x:=0 \ \{ x \geq 0 \} \\
\{ x \geq 0 \} & \ y:=0 \ \{ x \geq 0 \} \\
\{ x \geq 0 \} & \ x++ \ \{ x \geq 0 \} \\
\{ x \geq 0 \} & \ x==\!\!\!\!-1 \ \{ \text{false} \} 
\end{align*}
$$

proof of sample trace:
finite basis of Hoare triples "\{pre\} stmt \{post\}"
can be obtained from proofs of sample traces
finite basis of Hoare triples \( \{pre\) stmt \{post\} \) \( \rightarrow \) automaton

\[
\begin{align*}
\{ \text{true} \} & \quad x := 0 \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad y := 0 \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad x++ \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad x == -1 \quad \{ \text{false} \}
\end{align*}
\]

sequencing of Hoare triples in basis \( \rightarrow \) run of automaton
proof space contains “\{true\} trace \{false\}”
if
exists sequencing of Hoare triples in basis
if
exists accepting run of automaton
paradigm:
- construct proof space
- check proof space
inference rule for \textit{sequencing}

\[
\begin{align*}
\{p\} & \quad s \quad \{q'\} \\
\{q'\} & \quad s' \quad \{q\} \\
\hline \\
\{p\} & \quad s ; s' \quad \{q\}
\end{align*}
\]
inference rule for parallelism

\[
\begin{align*}
\{p\} & \quad s \quad \{q\} \\
\{p'\} & \quad s \quad \{q'\} \\
\hline
\{p \land p'\} & \quad s \quad \{q \land q'\}
\end{align*}
\]

“interference freedom”
inference rule for **unbounded number of threads**

\[
\{p\} s \{q\} \\
\text{----------} \text{ renaming of thread id's} \\
\{p'\} s' \{q'\}
\]

“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
    lock(m);
    if(next + 10 <= len)
        { c := next; next := next + 10; end := next; }
    else
        { c := next; next := next + 10; end := len; }
    unlock(m);

    // perform block of tasks
    while (c < end):
        tasks[c] := 0; // mark task c as started
        ... // work on the task c
        tasks[c] := 1; // mark task c as finished
        assert(tasks[c] == 1); // no other thread has started task c
        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

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

    lock(m);
    if(next + 10 <= len)
       { c := next; next := next + 10; end := next; }
    else
       { c := next; next := next + 10; end := len; }
    unlock(m);

threads 1, 2, \ldots, 35 have acquired block of tasks
have not yet started working
for given trace $\pi$ (fixed set of threads),
proof
can be
computed
automatically
by SMT solver
\{true\} \; \pi \; \{\text{end}(2) \leq c(9)\}

for given trace $\pi$

(fixed set of threads),

proof

can be

computed

automatically

by SMT solver
\{true\} \pi \{end(2) \leq c(9)\}

for given trace \(\pi\) (fixed set of threads), proof can be assembled automatically from a basis of atomic Hoare triples.
basis for Thread Pooling

\[
\begin{align*}
\{\text{true}\} & \quad \{c(1) \leq \text{next}\} & \quad \{\text{true}\} \\
\langle c := \text{next} : 1 \rangle & \quad \langle \text{next} := \text{next} + 10 : 1 \rangle & \quad \langle \text{end} := \text{next} : 1 \rangle \\
\{c(1) \leq \text{next}\} & \quad \{c(1) < \text{next}\} & \quad \{\text{end}(1) \leq \text{next}\}
\end{align*}
\]

\[
\begin{align*}
\{\text{end}(1) \leq \text{next}\} & \quad \{\text{true}\} & \quad \{\text{len} \leq \text{next}\} \\
\langle c := \text{next} : 2 \rangle & \quad \langle \text{assume}(\text{next} + 10 > \text{len}) : 1 \rangle & \quad \langle \text{end} := \text{len} : 1 \rangle \\
\{\text{end}(1) \leq c(2)\} & \quad \{\text{len} \leq \text{next}\} & \quad \{\text{end}(1) \leq \text{next}\}
\end{align*}
\]
\{true\}
lock(m)
\{true\}
assume(next + 10 <= len)
\{true\}
c := next
\{true\}
next := next + 10
\{true\}
end := next
\begin{align*}
\text{tasks}[c] &= 0 \\
\text{tasks}[c] &= 1
\end{align*}
\{end(2) \leq next\}
unlock(m);
\{end(2) \leq next\}
basis for Thread Pooling

\[
\begin{array}{ccc}
\{true\} & \{c(1) \leq \text{next}\} & \{true\} \\
\langle c := \text{next} : 1 \rangle & \langle \text{next} := \text{next} + 10 : 1 \rangle & \langle \text{end} := \text{next} : 1 \rangle \\
\{c(1) \leq \text{next}\} & \{c(1) < \text{next}\} & \{\text{end}(1) \leq \text{next}\}
\end{array}
\]

\[
\begin{array}{ccc}
\{\text{end}(1) \leq \text{next}\} & \{true\} & \{\text{len} \leq \text{next}\} \\
\langle c := \text{next} : 2 \rangle & \langle \text{assume}(\text{next} + 10 > \text{len}) : 1 \rangle & \langle \text{end} := \text{len} : 1 \rangle \\
\{\text{end}(1) \leq c(2)\} & \{\text{len} \leq \text{next}\} & \{\text{end}(1) \leq \text{next}\}
\end{array}
\]
inference by symmetry

```latex
\{\text{end}(2) \leq \text{next}\}
lock(m)
\{\text{end}(2) \leq \text{next}\}
\text{assume}(\text{next} + 10 \leq \text{len})
\{\text{end}(2) \leq \text{next}\}
c := \text{next}
\{\text{end}(2) \leq c(9)\}
\text{next} := \text{next} + 10
\{\text{end}(2) \leq c(9)\}
\text{end} := \text{next}
\{\text{end}(2) \leq c(9)\}
\text{unlock}(m)
\{\text{end}(2) \leq c(9)\}
```

("rename 2/1 and 9/2")
### Basis for Thread Pooling

<table>
<thead>
<tr>
<th>Hoare Triple</th>
<th>Invariant</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>{true}</td>
<td>(c := \text{next} : 1)</td>
<td>{c(1) \leq \text{next}}</td>
</tr>
<tr>
<td>{c(1) \leq \text{next}}</td>
<td>(\text{next} := \text{next} + 10 : 1)</td>
<td>{c(1) &lt; \text{next}}</td>
</tr>
<tr>
<td>{true}</td>
<td>(\text{end} := \text{next} : 1)</td>
<td>{\text{end}(1) \leq \text{next}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hoare Triple</th>
<th>Invariant</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>{\text{end}(1) \leq \text{next}}</td>
<td>(c := \text{next} : 2)</td>
<td>{\text{true}}</td>
</tr>
<tr>
<td>{\text{end}(1) \leq \text{c}(2)}</td>
<td>(\text{assume}(\text{next} + 10 &gt; \text{len}) : 1)</td>
<td>{\text{len} \leq \text{next}}</td>
</tr>
<tr>
<td>{\text{true}}</td>
<td></td>
<td>{\text{end} := \text{len} : 1}</td>
</tr>
<tr>
<td>{\text{true}}</td>
<td>{\text{len} \leq \text{next}}</td>
<td>{\text{end}(1) \leq \text{next}}</td>
</tr>
</tbody>
</table>
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**