\[ \ell_0: \text{assume } p \neq 0; \]
\[ \ell_1: \text{while}(n \geq 0) \]
\[ \{ \]
\[ \ell_2: \quad \text{assert } p \neq 0; \]
\[ \quad \text{if}(n == 0) \]
\[ \qquad \{ \]
\[ \ell_3: \quad p := 0; \]
\[ \qquad \} \]
\[ \ell_4: \quad n--; \]
\[ \} \]
\( \ell_0: \) assume \( p \neq 0; \)

\( \ell_1: \) while \( n \geq 0 \)

\[
\begin{align*}
  \ell_2: & \quad \text{if} (n == 0) \\
             & \quad \{ \\
             \ell_3: & \quad p := 0; \\
             \ell_4: & \quad n--; \\
            \}
\end{align*}
\]

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

\( \ell_1: \) while (\( n \geq 0 \))

\{ \hfill \}

\( \ell_2: \) assert \( p \neq 0; \)

\( \) if (\( n == 0 \))

\{ \hfill \}

\( \ell_3: \) \( p := 0; \)

\} \hfill \}

\( \ell_4: \) \( n--; \)

\}

\( \ell_5: \)
\( \ell_0 \): assume \( p \neq 0 \);
\( \ell_1 \): while(\( n \geq 0 \))
\{
\( \ell_2 \): assert \( p \neq 0 \);
\ 
\( \text{if}(n == 0)
\)
\{
\( \ell_3 \): \( p := 0 \);
\}
\( \ell_4 \): \( n-- \);
\}
\( \ell_5 \):

\[
\begin{align*}
\ell_0 & : \text{assume } p \neq 0; \\
\ell_1 & : \text{while}(n \geq 0) \\
\quad \{ \\
\quad \ell_2 & : \text{assert } p \neq 0; \\
\quad \quad \text{if}(n == 0) \\
\quad \quad \quad \{ \\
\quad \quad \quad \ell_3 & : p := 0; \\
\quad \quad \} \\
\quad \ell_4 & : n--; \\
\quad \} \\
\ell_5 & :
\end{align*}
\]

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

\[ \iff \]

correct \ wrt. \ pre/condition \ pair \ ( \text{true, false} )

\[
\{ \text{true} \} \ x == 1 \ ; \ x == -1 \ ; \ \{ \text{false} \}
\]

\[
\{ \text{true} \} \ x := 1 \ ; \ x == -1 \ ; \ \{ \text{false} \}
\]
correct program = infeasible trace

\( p \neq 0 \)

\( p = 0 \)

\( p = 0 \)
infeasible trace

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

unsatisfiable formula

\[ x' = 1 \land x' = -1 \]
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 `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 $(p \neq 0)$ $(n \geq 0)$ $(p == 0)$.

\[(p \neq 0)\]  
\[(n \geq 0)\]  
\[(p == 0)\]
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 \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 assert statement cannot fail). If the \texttt{then} branch of the conditional is executed, then the value of \( n \) is 0, the statement \texttt{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 \( \ell_0 \) to \( \ell_{\text{err}} \) goes via \( \ell_1 \) and \( \ell_2 \). The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not \((p \neq 0) \land (n \geq 0) \land (p = 0)\) \((p \neq 0) \land (p=0)\).
\( (p \neq 0) \)

\( (p == 0) \)
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\nabla$ means a transition reading any letter, an edge labeled with $\nabla \cap \{p := 0\}$ means a transition reading any letter except for $p := 0$, etc.).

We construct the automaton $A_1$ depicted 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. The shortest path from $q_0$ to $q_{err}$ with such a sequence of statements goes from $q_2$ to $q_{err}$ after it has gone from $q_2$ to $q_3$ once before. The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the $\text{assume}$ statement $n = 0$, the update statement $n --$, and the $\text{assume}$ 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 $A_2$ 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. $A_2$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts $n = 0$, $n_0 = n_1$, and $n_0 \geq 0$).

To summarize, we have twice taken a path from $q_0$ to $q_{err}$, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set $(p \neq 0)$ and $(p = 0)$.
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\exists$ means a transition reading any letter, an edge labeled with $\exists \land \{p := 0\}$ means a transition reading any letter except for $p := 0$, etc.).

It is possible to execute the assume statements $p \neq 0$ and $p = 0$ without an update of $p$ in between.

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 $q_0$ to $q_{\text{err}}$ with such a sequence of statements goes from $q_0$ to $q_3$ 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 $n = 0$, $n = 0$, and $n = 0$).

To summarize, we have twice taken a path from $q_0$ to $q_{\text{err}}$, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set $(p \neq 0)$, $(p = 0)$.
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\exists$ means a transition reading any letter, an edge labeled with $\exists \cap \{ p := 0 \}$ means a transition reading any letter except for $p := 0$, etc.).

Possible to execute the assume statements $p \neq 0$ and $p = 0$ without an update of $p$ in between.

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 $\bullet_0$ to $\bullet_{err}$ with such a sequence of statements goes from $\bullet_2$ to $\bullet_{err}$ after it has gone from $\bullet_2$ to $\bullet_3$ 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 $n = 0$, $n_0 = n_1$, and $n_0 = 0$).

To summarize, we have twice taken a path from $\bullet_0$ to $\bullet_{err}$, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set

\[
(p \neq 0) \\
(p = 0)
\]
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\triangle$ means a transition reading any letter, an edge labeled with $\triangle \cap \{p := 0\}$ means a transition reading any letter except for $p := 0$, etc.).

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 $n = 0$, $n = 0 = n$, and $n = 0 \neq 0$).

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 $(p = 0)$. 

$(p \neq 0)$

$(p = 0)$
correct program (error location is not reachable)

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

$$P_1$$

$P_1$

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

$$(p != 0)$$

$$(p == 0)$$
does a proof exist for every error 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 $l_5$. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not automaton

alphabet: \{statements\}
The program in Figure 1 is the adaptation of an example in [17] to our control flow graph and, at some point, enter the body of the while loop when the value of \( p = 0 \) (and the execution of the assert statement goes via (it does not have a possible execution) because it is not infeasible.

We can argue the correctness of \( P \) rather directly if we split the executions between (and with any statements before or after).

I.e., \( P \) is never changed and remains non-zero (and the assert statement fails).

\( P \) goes via \( P \) to a new reason: it is not possible to execute the while loop without an update of \( p \).

**Example 1: automata from infeasibility proofs**

\[
\begin{align*}
\ell_0 & \quad \text{p \neq 0} \\
\ell_1 & \quad \text{n < 0} \\
\ell_2 & \quad \text{n >= 0} \\
\ell_3 & \quad \text{n != 0} \\
\ell_4 & \quad \text{p := 0} \\
\ell_5 & \quad \text{p := 0}
\end{align*}
\]

\[
\begin{align*}
q_0 & \quad \Sigma \\
q_1 & \quad \Sigma \setminus \{ \text{p := 0} \} \\
q_2 & \quad \Sigma
\end{align*}
\]

inclusion between automata
Incremental Construction

\[ A_1, \ldots, A_n \] ` a la CEGAR

Program \( \mathcal{P} \)

\[ \mathcal{A}_\mathcal{P} \subseteq A_1 \cup \cdots \cup A_n \]

Construct \( \mathcal{A}_{n+1} \) such that

1. \( w \in \mathcal{A}_{n+1} \)
2. \( \mathcal{A}_{n+1} \subseteq \{ \text{infeasible traces} \} \)

\( w \) infeasible?

Yes

No

Take \( w \) such that

\( w \in \mathcal{A}_\mathcal{P} \setminus A_1 \cup \cdots \cup A_n \)

\( \mathcal{P} \) is correct

\( \mathcal{P} \) is incorrect
The program execution would start from \( P \) to \( P_1 \) if it contains \( p \neq 0 \), and the assertion statement \( p = 0 \) is 0, the statement \( n \neq 0 \) is never changed and remains non-zero (and the assert statement \( p 
eq 0 \)) after it has gone from \( A \) to \( B \).

The sequence of statements goes from \( A \) to \( B \) with \( n \neq 0 \), and the while loop will exit directly, without executing the \( a \) statements to define the correctness of the program executions. In the example of \( P \) is never changed and remains non-zero (and the assert statement \( p 
eq 0 \)) after it has gone from \( A \) to \( B \).

The sequence of statements goes from \( A \) to \( B \) with \( n \neq 0 \), and the while loop will exit directly, without executing the \( a \) statements to define the correctness of the program executions. In the example of \( P \) is never changed and remains non-zero (and the assert statement \( p 
eq 0 \)) after it has gone from \( A \) to \( B \).

For one thing, we can use automata as an expressive means to argue the correctness of a program specification. 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 case split and then constructing the corresponding correctness arguments, characterize different cases of execution paths.

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 argue the correctness of a program specification. 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 case split and then constructing the corresponding correctness arguments, characterize different cases of execution paths.

Example 1: automata from infeasibility proofs

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

To summarize, we have twice taken a path from \( A \) to \( B \) with such a case split and then constructed the corresponding correctness arguments, demonstrating the inclusion check fails and returns word in \( P \setminus P_1 \).
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 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 $\ell_0$ to $\ell_{err}$ goes via $\ell_1$ and $\ell_2$. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not:

\[
(p \neq 0) \\
(n \geq 0) \\
(n = 0) \\
(p := 0) \\
(n--) \\
(n \geq 0) \\
(p = 0)
\]
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 `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 `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 `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 $\ell_0$ to $\ell_{\text{err}}$ goes via $\ell_1$ and $\ell_2$. The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not

- $(p \neq 0)$
- $(n \geq 0)$
- $(n == 0)$
- $(p := 0)$
- $(n--)$
- $(n >= 0)$
- $(p == 0)$
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\nabla$ means a transition reading any letter, an edge labeled with $\nabla\{p := 0\}$ means a transition reading any letter except for $p := 0$, etc.).

We construct the automaton $A_1$ in Figure 2 which recognizes the set of all sequences of statements that contain $p != 0$ and $p == 0$ without an update of $p$ in between. 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 $p != 0$ and $p == 0$).

A sequence of statements is not accepted by $A_1$ if it contains $p != 0$ and $p == 0$ with an update of $p$ in between. The shortest path from $q_0$ to $q_{err}$ with such a sequence of statements goes from $q_2$ to $q_{err}$ after it has gone from $q_2$ to $q_3$ once before. The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the $\text{assume}$ statement $n == 0$, the update statement $n--$, and the $\text{assume}$ statement $n >= 0$ unless there is an (other) update of $n$ between $n == 0$ and $n--$ or between $n--$ and $n >= 0$.

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 $n >= 0$ 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 $n = 0$, $n_0 = n_1$, and $n_0 >= 0$).

To summarize, we have twice taken a path from $q_0$ to $q_{err}$, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set word not in $P_1$.

(word not in $P_1$

- $(p != 0)$
- $(n >= 0)$
- $(n == 0)$
- $(p := 0)$
- $(n--)$
- $(n >= 0)$
- $(p == 0)$)
(n == 0)
(n--)
(n >= 0)
Example 1: automata from infeasibility proofs

\[
\begin{align*}
&\text{ex1} := 0; \\
&\text{assume } p \neq 0; \\
&\text{while } (n \geq 0) \{
  \text{assert } p \neq 0; \\
  n--; \\
  \text{err} \\
\}
\end{align*}
\]

\[
\begin{align*}
&\text{ex2} := 1; \\
&\text{assume } p = 0; \\
&\text{while } (n \neq 0) \{
  \text{assert } p = 0; \\
  n--; \\
  \text{err} \\
\}
\end{align*}
\]

To summarize, we have twice taken a path from \(p_0\) (it does not have a possible execution) because it is not infeasible (it does not have a possible execution) because it is not infeasible (it does not have a possible execution) because it is not infeasible (it does not have a possible execution) because it is not infeasible (it does not have a possible execution) because it is not infeasible because it is not infeasible because it is not infeasible. The sequence of statements on the corresponding path in the automaton, analyzed the reason for the infeasibility, and constructed an automaton which each recognizes the set of all sequences of statements that contain the statements infeasible and the execution of the assert statement is never changed and remains non-zero (and the assert statement cannot fail). If the value of \(p\) is 0, the statement:

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

will exit directly, without executing the next branch of the conditional is executed, then the value of \(n\) is not changed and remains non-zero (and the assert statement cannot fail). If the value of \(p\) is 0, then:

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

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 cannot fail). If the value of \(p\) is 0, then:

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

then:

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

and the program executions. In the example of our setting we 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

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

gets executed at least once during the execution or it does not. If not, then we have twice taken a path from \(p_0\) and constructed an automaton which each recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts and the incorrect setting. In our setting we use 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 set of all sequences of statements that contain the statements is 0, the statement:

\[\text{assert } p \neq 0;\]
We construct the automaton $A$ depicted in Figure 2 which recognizes the set of all sequences of statements that contain

```
{ n == 0 } { n-- } { n >= 0 },
```

and any statements before and after. I.e.,

- $(n == 0)$
- $(n--)$
- $(n >= 0)$

The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the update statement $p := 0$ without an update of $n$. Therefore, an edge labeled with $\Sigma$ means a transition reading any let-

```
{ n >= 0 } { n-- } { n >= 0 },
```

and between (and with any statements before or after). I.e.,

```
\{ n != 0 \},
```

which are a proof of correctness for the infeasibility of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts $n >= 0$, $n--$, and $n != 0$).
correct program $P_2$ constructed from a proof

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

\[ A_1, \ldots, A_n \] ` a la CEGAR

Program \( \mathcal{P} \)

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

\( A_\mathcal{P} \subseteq A_1 \cup \cdots \cup A_n \)?

Yes

Yes

\( \mathcal{P} \) is correct

No

No

\( w \) infeasible?

Yes

Take \( w \) such that
\( w \in A_\mathcal{P} \setminus A_1 \cup \cdots \cup A_n \)

\( \mathcal{P} \) is incorrect
check inclusion between automata
does a proof exist for every trace?
We will describe an execution of $\text{ex1}$.

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 di erent cases of execution paths. For another, instead of first fixing the value of $p$, we can use automata to characterize the case of exactly the executions for which the corresponding correctness argument applies. We will next illustrate this in the example of automaton which each recognizes the set of all sequences of statements that contain the statements $p := 0$ 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 $p \neq 0$ fails). If the while loop will exit directly, without executing the branch of the conditional is executed, then the value of $p$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds).

To summarize, we have twice taken a path from $p \neq 0$ to $p = 0$, and the while loop will exit directly, without executing the branch of the conditional is executed, then the value of $p$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds). I.e., $\text{ex1}$ does not fail. If the while loop will exit directly, without executing the branch of the conditional is executed, then the value of $p$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds).

While we can construct an automaton for a given correctness argument so that the case split and then constructing the corresponding correctness arguments, we can argue the correctness of execution would start infeasible in between.

Example 1: automata from infeasibility proofs

- $P$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds).
- If the while loop will exit directly, without executing the branch of the conditional is executed, then the value of $p$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds).

We construct the automaton $A$ which recognizes the set of infeasible traces.

To have a proof of correctness for $\text{ex1}$, an edge labeled with $\text{err}$ means a transition reading any letter after it has gone from $p \neq 0$ to $p = 0$. I.e., $\text{ex1}$ does not fail. If the while loop will exit directly, without executing the branch of the conditional is executed, then the value of $p$ is never changed and remains non-zero (and the assert statement $p \neq 0$ succeeds). I.e., $\text{ex1}$ does not fail.
Incremental Construction

A₁,...,Aₙ

` a la CEGAR

Aₚ ⊆ A₁ ∪ ... ∪ Aₙ ?

w infeasible?

program P

construct Aₙ₊₁ such that
1. w ∈ Aₙ₊₁
2. Aₙ₊₁ ⊆ { infeasible traces }

take w such that
w ∈ Aₚ \ A₁ ∪ ... ∪ Aₙ

P is correct

P is incorrect
<table>
<thead>
<tr>
<th>Infeasible traces</th>
<th>Feasible traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>No corresponding executions</td>
<td>At least one corresponding execution</td>
</tr>
</tbody>
</table>
Infeasible traces

Feasible traces

Error traces
Infeasible traces

Feasible traces

Error traces

Proof Generalization
Infeasible traces

Feasible traces

Proof Generalization
Infeasible traces

Feasible traces

Error traces
Infeasible traces

Feasible traces

Error traces

Proof Generalization
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
\[\ell_0: x := 0;\]
\[\ell_1: y := 0;\]
\[\ell_2: \text{while(nondet)} \{ x++; \}\]
\hspace{1em} assert(x != -1);
\hspace{1em} assert(y != -1);
program $P$:

all behaviors of program $P$ covered by two programs below:
We use them in the same way as above in order to construct the automaton corresponding to one path from a static analysis [8] applied to the program fragment that corresponds to the preceding example can only have self-loops. The automaton can have arbitrary loops. In contrast, an automaton constructed as a Floyd-Hoare automaton has three states, one for each assertion: the initial state for $x=0$, the (only) final state for $x=-1$, and determine that the automaton $\Delta$ 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 $\Delta \subseteq \Delta'$. Hoare triples. The four Hoare triples below are sufficient to prove the correctness of the example of the program $P$ (which labels a path from $A$ to $\text{err}$ in Figure 3 shows that sometimes a more refined justification is required. The sequence of the two statements on paths from $\text{ex1}$ to $\text{ex2}$ in Example 1: $x:=0$, $y:=0$, $x++$, $x==-1$, $y==-1$, $x==0$, the (only) final state.
unsatisfiable core of unsatisfiability proof uses variable $x$ => program constructed from unsatisfiability proof has no self-loop with statement $x++$ in

\[
\begin{align*}
\ell_0 & \xrightarrow{x:=0} \ell_1 \\
\ell_1 & \xrightarrow{y:=0} \ell_2 \\
\ell_2 & \xrightarrow{x++} \ell_2 \\
\ell_2 & \xrightarrow{x=-1} \ell_{err} \\
\end{align*}
\]

\[
\begin{align*}
x & := 0 \\
y & := 0 \\
x & ++ \\
x & == -1
\end{align*}
\]
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 $P_{ex1} \subseteq A_1 \cup A_2$.

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 $x==-1$.

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

Hoare triples proving infeasibility:

\[
\text{infeasibility} \iff \text{pre/postcondition pair} \ (true, false)
\]
Hoare triples $\rightarrow$ correct program

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

construction of correct program from Floyd-Hoare proof of infeasibility of trace
(remember: infeasibility \iff 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)
The example of the program in Figure 3 shows that sometimes a more powerful form of justification is required. The sequence of the two statements involved justification is required. The sequence of the two statements in the example above, the variable `x` is modified, but the variable `y` is not. So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements? So how can we determine that a statement modifies the variable that appears in the two statements?

Hoare triples $\mapsto$ automaton

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

sequencing of Hoare triples $\mapsto$ run of automaton
program $P$:

all behaviors of program $P$ covered by two programs below:
covering check = automata inclusion check

\[ P_{\text{ex2}} \subseteq A_1 \cup A_2 \]
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 translating SMT solver [6] which generates the assertion corresponding to one path from a static analysis [8] applied to the program fragment that corresponds to in the preceding example can only have self-loops. In contrast, an automaton constructed as Floyd-Hoare automaton can have arbitrary loops. In our implementation [12], the set of Hoare triples comes from an interpolation.

The automaton in Figure 4 has four transitions, one for each Hoare triple. The second trace is not reachable.

x := 0
y := 0
x++

y === -1
automaton from **unsatisfiability core** of infeasibility proof for second trace
Hoare proof
for infeasibility of second trace

\[
\begin{align*}
\{ \text{true} \} & \quad \text{x:=0} \quad \{ \text{true} \} \\
\{ \text{true} \} & \quad \text{y:=0} \quad \{ y = 0 \} \\
\{ y = 0 \} & \quad \text{x++} \quad \{ y = 0 \} \\
\{ y = 0 \} & \quad \text{y===-1} \quad \{ \text{false} \}
\end{align*}
\]
automaton from Hoare proof
for infeasibility of second trace

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

\[ P_{ex2} \subseteq A_1 \cup A_2 \]
Incremental Construction

A₁, ..., Aₙ ` a la CEGAR

Program \( \mathcal{P} \)

Construct \( \mathcal{A}_{n+1} \) such that
1. \( w \in \mathcal{A}_{n+1} \)
2. \( \mathcal{A}_{n+1} \subseteq \Sigma^* \backslash \text{CORRECT} \)

\( \mathcal{A}_{\mathcal{P}} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \)?

\( w \in \Sigma^* \backslash \text{CORRECT} \) ?

Take \( w \) such that
\( w \in \mathcal{A}_{\mathcal{P}} \backslash \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \)

\( \mathcal{P} \) 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