Search Modulo Theory

Andreas Podelski
University of Freiburg
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...
\( \ell_0: \text{assume } p \neq 0; \)

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

\( \ell_5: \)
\( \ell_0: \text{assume } p \neq 0; \)

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

\[
\text{no execution violates assertion} \quad = \quad \text{no execution reaches error location}
\]
path in infinite state space of program: *feasible* path in finite control flow graph
infinite search space: symbolically by finite graph
(edges labeled by constraints and updates)

path in infinite search space: feasible path in finite graph

feasibility = Satisfiability Modulo Theory (SMT)
infeasible trace

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

unsatisfiable formula

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

infeasible/unsatisfiable ... Modulo Theory
infeasible trace

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

unsatisfiable \textit{Modulo Theory}

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

\[ x' = 1 \land x' = -1 \]
Automated Program Verification

Andreas Podelski
University of Freiburg

joint work with
Matthias Heizmann and Jochen Hoenicke
University of Freiburg
Azadeh Farzan and Zachary Kincaid
University of Toronto
automaton
|ˌôˈtämətən|

automation
|ˌôtəˈmāSHən|
Incremental Construction

A \cdots, A_n ` a la CEGAR

Program $P$ is correct

$P$ is incorrect

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

$w$ infeasible? 

take $w$ such that

$w \in A_P \setminus (A_1 \cup \cdots \cup A_n)$

$A_{n+1}$ such that

1. $w \in A_{n+1}$
2. $A_{n+1} \subseteq \{ \text{infeasible traces} \}$

$P$ is correct

$P$ is incorrect
Ultimate Automizer

**ULTIMATE WEB INTERFACE**

**Task:** Verify C

**Sample:** McCarthy91.c

**Tool:** Trace Abstraction

---

```c
12 /* requires \true;
13 @ ensures x > 101 || \result == 91;
14 */
15 int f91(int x);
16
17 int f91(int x) {
18   if (x > 100)
19     return x - 10;
20   else {
21     return f91(f91(x+11));
22   }
23
24
25
26
```

---

**SETTINGS**

---

**EXECUTE**

---

**Show editor fullscreen**

**Choose File**

No file selected

---

**Line** | **Ultimate Result**
-----|-------------------
21 | procedure precondition always holds
21 | procedure precondition always holds
13 | procedure postcondition always holds
program = automaton

constructed from proof

proof by SMT solver
search *Modulo Theory*

add lemmas to prune the search space

lemmas inferred from proofs of *SMT* solver

*SMT*: Satisfiability Modulo Theory
search *Modulo Theory*

add lemmas to prune the search space
lemmas inferred from proofs of *SMT* solver

lemmas are automata (sets of paths)
automata constructed from proofs of *SMT* solver

*SMT*: Satisfiability Modulo Theory
The program $P_{ex1}$ in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use $\text{assert}$ 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 $\text{assert}$ 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 $\text{assert}$ 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 a $\text{error trace}$: sequence of statements along path to error location.
error trace: sequence of statements along path to 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 $l_0$ 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 error trace: word accepted by program automaton
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 program correct.

$$\text{program correct } = \text{ no feasible error trace}$$
Incremental Construction

A₁, ..., Aₙ

`a la CEGAR

Program \( P \)

\( P \) is correct

\( P \) is incorrect

\[ \mathcal{A}_P \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \] ?

\( w \) infeasible?

\[ \{ \text{infeasible traces} \} \]

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

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

Take \( w \) such that

\[ w \in \mathcal{A}_P \setminus \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \]

\( \mathcal{P} \) is correct

\( \mathcal{P} \) is incorrect
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 $l_0$ to $l_{\text{err}}$ 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 feasible 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 $\text{assert}$ 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 $\text{assert}$ 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 $\text{assert}$ 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 $l_{\text{err}}$ 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 complex control? - just ignore it!
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

$$(p \neq 0)$$

$$(n \geq 0)$$

$$(p == 0)$$

error trace: sequence of statements along an error path
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 $(p \neq 0)$ $(n \geq 0)$ $(p == 0)$ $(p != 0)$ $(p==0)$

trace infeasible, formula unsatisfiable, take unsatisfiable core
(p \neq 0)

(p==0)

unsatisfiable core
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 $n = 0$, $n = 0$, and $n \geq 0$).

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 $(p = 0)$. Construct automaton from unsatisfiable core (step 1).
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 $q_{err}$ 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 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 $n_0 \leq n_2$).

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)
(p = 0)
```

construct automaton from unsatisfiable core (step 2)
automaton constructed from unsatisfiability proof

Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\triangleright$ means a transition reading any letter, an edge labeled with $\triangleright \cap \{ p := 0 \}$ 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. 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 $q_2$ with such a sequence of statements goes from $q_0$ to $q_2$ after it has gone from $q_0$ to $q_1$ 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 $q_2$, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set of sequences of statements that contain $p \neq 0$ and $p = 0$ without an update of $p$ in between.

“data automaton” (ignores control of program)
The program executions. In the example of

Example 1: automata from infeasibility proofs

The sequence of statements on this path is infeasible for a new reason: it

To summarize, we have twice taken a path from

We construct the automaton

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

subset?

subset?

program automaton subset of data automaton?
We will describe an execution of a while loop when the value of $n$ is 0, the statement $p := 0$; see Figure 1. The automaton $P_{ex1}$ is the adaptation of an example in [17] to our setting. In our setting we use infeasible programs (it does not have a possible execution) because it is not possible to execute the sequence of statements $n --$ except for $p := 0$.

We can construct an automaton for a given correctness argument so that the case split and then constructing the corresponding correctness arguments, the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of an automaton characterizes the case of exactly the executions for which the correctness argument applies. We can argue the correctness of the subset?

The program, an execution would start $p := 0$; and, at some point, enter the body of the loop will exit directly, without executing the

program automaton not subset of data automaton
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:

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

This sequence of statements cannot happen in the given program.

new error trace
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\triangledown$ means a transition reading any letter, an edge labeled with $\triangledown \cap \{ p := 0 \}$ 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. 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 $q_{err}$ 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 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 $n = 0$, $n = n = 0$, and $n > 0$).

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 of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of $p \neq 0$ and $p = 0$).

data automaton does not accept new 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 $\ell_0$ 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 $(p != 0)$

$(n >= 0)$

$(n == 0)$

$(p := 0)$

$(n --)$

$(n >= 0)$

$(p == 0)$

trace infeasible, take unsatisfiable core
(n == 0)

(n--)

(n >= 0)

unsatisfiable core
construct second data automaton from unsatisfiable core (step 1)
Example 1: automata from infeasibility proofs

We can argue the correctness of the program executions. In the example of Fig. 1 is the adaptation of an example in [17] to our setting. In our setting we use infeasible infeasible of its infeasibility, 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 between (and with any statements before or after). I.e., the shortest path from to goes via and, at some point, enter the body of the while loop when the value of is 0 (and the execution of the assert statement is never changed and remains non-zero (and the assert statement)

We construct the automaton which are a proof of correctness for construct second data automaton from unsatisfiable core (step 2)

\[
\begin{align*}
\text{p}_0 & \xrightarrow{\Sigma} \\
\text{n} &= 0 \\
\text{p}_1 & \xrightarrow{\Sigma \setminus \{\text{n--}\}} \\
\text{n} &= 0 \\
\text{p}_2 & \xrightarrow{\Sigma \setminus \{\text{n--}\}} \\
\text{n} &\geq 0 \\
\text{p}_3 & \xrightarrow{\Sigma}
\end{align*}
\]
We can argue the correctness of the program executions. In the example of Example 1: automata from infeasibility proofs.

```
4
3
2
1
0
```

We will describe an execution of `ex1` in Figure 1 is the adaptation of an example in [17] to our control flow graph.

```
while(n >= 0)
    assume p != 0;
    n--;  
if(n == 0)
    assert p != 0;
```

We construct the automaton `P` which recognizes the set of all sequences of statements that contain the statements `while(n >= 0)` and `if(n == 0)`. I.e., an edge labeled with `p := 0` means a transition reading any letter in between. The shortest path from `p := 0` to `p := 0` is `p != 0`.

To summarize, we have twice taken a path from `q1` to `q1`. We construct the automaton `A` which recognizes the set of all sequences of statements that are infeasible `q1` means a transition reading any letter in between (and with any statements before or after). I.e., `q1` is an edge labelled with `p := 0`, `q1` is a subset of union of data automata.

The sequence of statements goes from `p := 0` to `p := 0`, and the inconsistency of `p := 0` fails. If the value of `p` is never changed and remains non-zero (and the assert statement `p != 0`), then `p := 0` is infeasible. If the value of `p` is 0, the statement `p := 0` cannot fail. If the program executions fail, then `p := 0` is impossible to execute the `p != 0` statement.

The value of `p` in `ex1` is 0, the statement `p := 0` is infeasible.

The value of `p` in `ex1` is never changed and remains non-zero (and the assert statement `p != 0`) is never executed. The value of `p` in `ex1` is never changed and remains non-zero (and the assert statement `p != 0`) is never executed. The value of `p` in `ex1` is never changed and remains non-zero (and the assert statement `p != 0`).

To summarize, we have twice taken a path from `p := 0` to `p := 0`. We construct the automaton `A` which recognizes the set of all sequences of statements that contain the statements `p := 0` and `p == 0`. I.e., an edge labeled with `p := 0` means a transition reading any letter in between. The shortest path from `p := 0` to `p := 0` is `p != 0`.

To summarize, we have twice taken a path from `p := 0` to `p := 0`. We construct the automaton `A` which recognizes the set of all sequences of statements that are infeasible `p := 0` means a transition reading any letter in between (and with any statements before or after). I.e., `p := 0` is an edge labelled with `p := 0`, `p := 0` is a subset of union of data automata.

The sequence of statements goes from `p := 0` to `p := 0`, and the inconsistency of `p := 0` fails. If the value of `p` is never changed and remains non-zero (and the assert statement `p != 0`) is never executed. The value of `p` in `ex1` is never changed and remains non-zero (and the assert statement `p != 0`) is never executed. The value of `p` in `ex1` is never changed and remains non-zero (and the assert statement `p != 0`).
program \( \mathcal{P} \)

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

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

- \( w \) infeasible?

  - yes
  - no

  - take \( w \) such that
    \( w \in \mathcal{A}_{\mathcal{P}} \setminus \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \)

- \( \mathcal{P} \) is correct

- \( \mathcal{P} \) is incorrect
repeat

- take error trace in program
- check infeasibility of error trace
- construct data automaton from unsatisfiability proof

until program is subset of union of data automata
if automata are constructed from **unsatisfiable core**

is the verification algorithm complete?

(can we prove every correct program correct?)
\(\ell_0: \ x := 0;\)
\(\ell_1: \ y := 0;\)
\(\ell_2: \ \text{while(nondet)} \ \{x++;\}\)
\hspace{1em} assert(x != -1);
\hspace{1em} assert(y != -1);
\( \ell_0: \ x := 0; \)
\( \ell_1: \ y := 0; \)
\( \ell_2: \ \text{while(\text{non}d) \{x++;\}} \)
  \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 translating SMT solver [6] which generates the assertion one path from a static analysis [8] applied to the program fragment that corresponds to the statements in the sequence (here, the variable that appears in the two statements). So how can we account for the paths that loop in 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 a word accepted by 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, namely

\[
P_{\text{ex1}}(x=y) \subseteq P_{\text{ex2}}(x=y)
\]

where \(P_{\text{ex1}}\) and \(P_{\text{ex2}}\) are the automata recognizing the sets of all words for which a word is accepted by the automaton in Example 1 and Example 2, respectively. The example of the program in Figure 3 shows that sometimes a more involved justification is required. The sequence of the two statements involved is preserved if one adds statements that do not modify any of the variables of the program. It is “easy” to justify the construction of the automata infeasibility of the word (i.e., the corresponding sequence of statements), and we can, however, check a condition which is stronger, namely that all sequences of statements that are infeasible for the specific reason. The two cases (the condition is stronger because not every such path is infeasible) have constructed an automaton which recognizes the set of all words for which one or \(x==0\) \(y==0\) \(x++\) \(x===-1\)
trace infeasible =

trace satisfies pre/postcondition pair \((true, false)\)
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 $P_{ex1} \sqsubseteq A_1 \sqsubseteq A_2$.

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!=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 `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 `x0 holds after the update `x:=0, that it is invariant under the updates `y:=0 and `x++, and that is blocks the execution of the assume statement `x==-1.

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

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

all traces accepted by automaton are infeasible

infeasible = satisfy pre/postcondition pair \((true, false)\)
construction “Hoare proof $\dashv\rightarrow$ automaton”

Hoare triple $\dashv\rightarrow$ transition

\[
\begin{align*}
\{ \text{true} \} & : x := 0 \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & : y := 0 \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & : x++ \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & : x==1 \quad \{ \text{false} \}
\end{align*}
\]
construction “Hoare proof $\iff$ automaton”

assertion $\iff$ state

Hoare triple $\iff$ transition

precondition $\iff$ initial state

postcondition $\iff$ final state

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

A 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 a Floyd-Hoare automaton for $q_1$ has three states, one for each assertion: the initial state $q_0$, the state $q_1$, and the state $q_2$, which is the (only) final state.

The automaton in Figure 4 has four transitions, one for each Hoare triple.

The four Hoare triples below are sufficient to prove the correctness of the infeasibility of all paths that reach the error location from $l_0$ in the example. The construction of such a program automaton subset data automaton?

```
x := 0
y := 0
x++
```

```
x == -1
assert(x != -1);
```

```
y := 0
x := 0
stuff happens
```

Where does the set of Hoare triples come from? In this example, it may come from an interpolating SMT solver.

In our implementation [12], the set of Hoare triples comes from an interpolating SMT solver.
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 `corresponding to `one path from the Floyd-Hoare automaton.

The four Hoare triples below are sufficient to prove the correctness of the program:

```
assert(y != -1);
assert(x != -1);
while(nondet)
  y := 0;
  x := 0;
```

In our implementation [12], the set of Hoare triples comes from an interpolating SMT solver [6] which generates the assertion `true to the location \( P_{\text{ex2}} \).

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

Where does the set of Hoare triples come from? In this example, it may come from a static analysis [8] applied to the program fragment that corresponds to the program in Figure 3.

```
x
y
y===-1
```

The construction of such a static analysis may assign an abstract value \([A, B, C]\) to \( x \), the (only) final state in Figure 4 has four transitions, one for each Hoare triple.

The four Hoare triples below are sufficient to prove the infeasibility of all program locations:

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

The automaton can have arbitrary loops. In contrast, an automaton constructed as a Floyd-Hoare automaton generalizes to any set of Hoare triples. The resulting program automaton is not a subset of the data automaton:

```
q0
q1
q2
x:=0
x++
```
automaton from **Hoare triples**
which prove infeasibility

\[
\begin{align*}
\{true\} & \; x:=0 \; \{true\} \\
\{true\} & \; y:=0 \; \{y = 0\} \\
\{y = 0\} & \; x++ \; \{y = 0\} \\
\{y = 0\} & \; y==1 \; \{false\}
\end{align*}
\]
automaton from Hoare triples which prove infeasibility

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

we can construct the same automaton from unsatisfiable core (since variable \( x \) does not appear in unsatisfiable core)
construction of automaton from unsatisfiable core is a special case of
construction of automaton from Hoare proof

exists proof for infeasibility of trace
⇒
exists Hoare proof whose assertions are invariant under any statement that does not update a variable in unsatisfiable core
We use them in the same way as above in order to construct the automaton.

The automaton has three states, one for each assertion: the initial state $q_0$, the state $q_1$ for $x = 0$, and the final state $q_2$ for $x = -1$. The automaton can have arbitrary loops. In contrast, an automaton constructed as

$$P \subseteq \ell_0 \cup \ell_1 \cup \ell_2 \cup \ell_{end}$$

where $\ell_0$, $\ell_1$, $\ell_2$, and $\ell_{end}$ are the program automaton $\subseteq$ of union of data automata.

The four Hoare triples below are sufficient to prove the correctness of the

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

The construction of such a

$$\{ x = 0 \} y := 0 \{ y = 0 \}$$

The four Hoare triples below are sufficient to prove the infeasibility of all

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

Where does the set of Hoare triples come from? In this example, it may come

$$\{ x = 0 \} y := 0 \{ y = 0 \}$$

The four Hoare triples below are sufficient to prove the correctness of the

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

Where does the set of Hoare triples come from? In this example, it may come

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

The four Hoare triples below are sufficient to prove the infeasibility of all

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

Where does the set of Hoare triples come from? In this example, it may come

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

The four Hoare triples below are sufficient to prove the correctness of the

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

Where does the set of Hoare triples come from? In this example, it may come

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$

The four Hoare triples below are sufficient to prove the correctness of the

$$\{ y = 0 \} x := 0 \{ y = 0 \}$$
We use them in the same way as above in order to construct the automaton corresponding to $\text{in}$ in the preceding example can only have self-loops. An automaton can have arbitrary loops. In contrast, an automaton constructed as a Floyd-Hoare automaton.

Fig. 4: Automata

The automaton $\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_{\text{en}}$ has four transitions, one for each Hoare triple.

Where does the set of Hoare triples come from? In this example, it may come from the infeasibility of all paths that reach the error location via the edge labeled with $\text{err}$.

The four Hoare triples below are sufficient to prove the correctness of the program; i.e., they are a subset of union of data automata!

The construction of such a path is not reachable.

program automaton is subset of union of data automata!
Incremental Construction
A₁,...,Aₙ

A is correct
P is incorrect

Wᵢ ✓

Aₚ ⊆ A₁ ∪ ⋯ ∪ Aₙ ?

W infeasible?

Aₚ ⊆ A₁ ∪ ⋯ ∪ Aₙ ?

{ infeasible traces }

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

take w such that
w ∈ Aₚ \ A₁ ∪ ⋯ ∪ Aₙ

P is correct
P is incorrect

w infeasible?
repeat

- take error trace in program

- check infeasibility of error trace

- construct data automaton from Hoare triples

until program is subset of union of data automata
repeat

- take error trace in program

- check infeasibility of error trace

- construct data automaton from unsatisfiability proof

until program is subset of union of data automata
repeat
- take error trace in program
- check infeasibility of error trace
- construct data automaton from Hoare triples
until program is subset of union of data automata
completeness of verification algorithm

for every correct program
there exists data automata
such that
program automaton $\subseteq$ union of data automata
conclusion and future work
automaton constructed from proof

proof generated by SMT solver

(Satisfiability checker Modulo Theory)
a trace is a word

a trace is a program
we can use automata to express new sets of traces

program is just one particular automaton
program expresses one particular set of traces

“cover program by union of simple automata”
Automizer

C and Boogie, safety and termination
sequential programs ........... nondeterministic finite automata
termination ........................ Buchi automata
recursion .......................... nested word automata
concurrency ........................ alternating finite automata
unbounded parallelism .... predicate automata
proofs that count ............ Petri net ⊆ counting automaton
data base of automata

automata constructed from proofs
Incremental Construction

$A_1, \ldots, A_n$ `a la CEGAR

Program $\mathcal{P}$

Construct $\mathcal{A}_{n+1}$ such that
1. $w \in \mathcal{A}_{n+1}$
2. $\mathcal{A}_{n+1} \subseteq \{ \text{infeasible traces} \}$

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

$w$ infeasible?

Take $w$ such that $w \in \mathcal{A}_\mathcal{P} \setminus \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$

$\mathcal{P}$ is correct

$\mathcal{P}$ is incorrect
construct automaton from proof of **incorrectness**

error diagnosis
statement *irrelevant* if on loop in automaton

classify error paths
error paths equivalent if same automaton