Proof Spaces

Andreas Podelski

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

• new paradigm for automatic verification
• automata
• Marc Segelken: $\omega$-Cegar  [CAV 2007]
• verification for networked traffic control systems
Ultimate Automizer

**ULTIMATE WEB-INTERFACE**

```c
/*@ requires \true;
@ ensures x > 101 || \result == 91;
*/
int f91(int x);

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Ultimate Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>procedure precondition always holds</td>
</tr>
<tr>
<td>21</td>
<td>procedure precondition always holds</td>
</tr>
<tr>
<td>13</td>
<td>procedure postcondition always holds</td>
</tr>
</tbody>
</table>
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. \( A_{n+1} \subseteq \{ \text{infeasible traces} \} \)

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

\( w \) infeasible?

\( \mathcal{P} \) is correct

\( \mathcal{P} \) is incorrect

\( \mathcal{P} \) is correct

\( \mathcal{P} \) is incorrect

take \( w \) such that
\( w \in A\_\mathcal{P} \setminus A\_1 \cup \cdots \cup A\_n \)
Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Jochen Hoenicke, Azadeh Farzan, Zachary Kincaid, Markus Lindenmann, Betim Musa, Christian Schilling, Alexander Nutz, Stefan Wissert, Evren Ermis

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

proof spaces

- new paradigm for automatic verification
- automata
- Marc Segelken: $\omega$-Cegar [CAV 2007]
- verification for networked traffic control systems
The AVACS Vision

To **Cover the Model- and Requirement Space of Complex Safety Critical Systems**

with **Automatic Verification Methods**

Giving Mathematical Evidence of Compliance of Models

To **Dependability, Coordination, Control and Real-Time Requirements**
Automating Verification of Cooperation, Control, and Design in Traffic Applications *

Werner Damm\textsuperscript{1,2}, Alfred Mikschl\textsuperscript{1}, Jens Oehlerking\textsuperscript{1}, Ernst-Rüdiger Olderog\textsuperscript{1}, Jun Pang\textsuperscript{1}, André Platzer\textsuperscript{1}, Marc Segelken\textsuperscript{2}, and Boris Wirtz\textsuperscript{1}
In case that the EoA is a rail-road crossing the train has to lock the rail-road crossing before the train will reach this point. The train has to initialize a lock request to the rail-road crossing and acknowledge the lock request. After receiving a safe message the train can send a request for a new EoA to the RBC. The point to initialize a lock request to a level crossing is calculated by

\[ x_{\text{lock}} = x_{\text{c}} - 2 \cdot \big(\text{time} + \text{max send delay}\big) \cdot v \]  

The time to set up the rail-road crossing in a safe state is stored in the \(x_{\text{time}}\) variable. These three points \((x_{\text{b}}, x_{\text{c}}, x_{\text{lock}})\) are updated every time the brake point state is entered. A spatial view of this scenario is shown in Fig. 4, and Fig. 5.

After explaining the main ideas to guarantee a safe motion, we continue to discuss Fig. 3. The transition is enabled after receiving an end-of-authority message from the RBC. By taking this transition the two variables which count the messages to the RBC and to the rail-road crossing are initialized to 0. The variable \(x_{\text{cross}}\) picks up the information if a rail-road crossing is just in front of the current position \(p\) of the train. The position of the rail-road crossing itself is stored in the \(x_{\text{p}}\) variable. The information of the rail-road crossing is read out of the track data dictionary. If there is no rail-road crossing ahead and the current position of the train is before the point to initialize the service brake and before the point to initialize an EoA-request the transition will be taken and an ev drive event is generated to switch into the driving mode of the train. In case there is a rail-road crossing ahead the transition will be enabled. If the train has passed the \(x_{\text{b}}\) point a lock request is generated by the ev com cross event released by the transition 4. If the train has passed the \(x_{\text{c}}\) point a lock request is generated by the ev com cross event released by the transition 4. If the train has passed the \(x_{\text{b}}\) point the transition 7 is enabled which leads to the service brake mode by the ev brake event. In
The supervision of the velocity of the train is modelled in the four states labelled move (Fig. 6). The previous value of the desired speed is set to 0 on entering the default init state and stored in the od variable. The drive mode is switched on by receiving the ev drive event enabling the transition and the variable dc (for drive control) is set, the desired speed at the current position of the train is read out of the track data dictionary and stored in the variable dst and the slope of the current track segment is stored in the variable slope.

– If the current desired speed is different from the previous value the transition is enabled and this change is signalled through a reset on the cvd variable, state switch and the transition 3. The new value of the current desired speed is stored in the move state.

– If there is no change in the desired speed, the transition 4 is enabled and the move state is entered.

– If a brake event (ev brake) occurs in the change state, the transition 12 is enabled, the drive mode is switched off (dc = 0), the service brake mode is switched on (sb = 1) and finally the init brake state is entered.

Fig. 5. Snapshot of dynamic calculations
holistic verification methodology

dedicated methods for:
- cooperation layer
- control layer
- design layer

model checking for discrete hybrid systems
- Lin AIGs
- $\omega$-Cegar
For the drive train system, we have performed this kind of analysis for a fixed sampling rate (0.1 seconds) and a discrete controller obtained by a textbook discretization method for linear systems (zero-pole matching transformation [17]). The resulting sampled-data system consists of the continuous-time drive train dynamics given in Subsection 2.1 and the discretized controller, and it can still be proven stable by this method.

7 Proving Safety of Local Control and Design Models

In this section, we present our approach of model checking safety properties of local control and design models of the example. We first outline our general methods for verification of hybrid systems with non-trivial discrete behaviour (Subsections 7.1 and 7.3); then we build both continuous-time and discrete-time models of the system based on its Matlab-Simulink description and show model checking results of these models (Subsections 7.2 and 7.4).

7.1 Model Checking Hybrid Systems with Large Discrete State Spaces

We have proposed an approach for verification of hybrid systems, which contain large discrete state spaces and simple continuous dynamics given as constants [11] (methods dealing with richer dynamics, e.g., given as differential inclusions, are currently under development). Large discrete state space arise naturally in industrial hybrid systems, due to the need to represent discrete inputs, counters, sanity-check bits, possibly multiple concurrent state machines etc, which typically join with properties of sensor values determine the selection of relevant control laws. Thus this non-trivial discrete behavior cannot be treated by considering discrete states one by one as in tools based on the notion of hybrid automata. We have developed a model checker dealing with ACTL properties for this application class.

Fig. 17. The Lin-AIG structure
proof spaces

• new paradigm for automatic verification
• automata
• Marc Segelken: $\omega$-Cegar [CAV 2007]
• verification for networked traffic control systems
Abstraction and Counterexample-guided Construction of $\omega$-automata for Model Checking of Step-discrete linear Hybrid Models*

Marc Segelken


* This research was partially supported by the German Research Foundation (DFG) under contract SFB/TR 14 AVACS, see www.avacs.org
Construction of ω-automaton. Thus we follow a strategy of completely ruling out generalized conflicts by constructing an ω-automaton $A_C$ that accepts all runs not containing any known conflict as a subsequence. Considering partial regulation laws as atomic characters and $C$ as the set of all previously detected generalized conflicts, the behavior of $A_C$ can be described by an LTL formula:

$$A_C \models \neg F \lor (\rho_1 \land X(\rho_2 \land X(... \land X\rho_n))) \quad (21)$$
proof spaces

• new paradigm for automatic verification

• automata

• Marc Segelken: ω-Cegar  [CAV 2007]

• verification for networked traffic control systems
\( \ell_0: \text{assume } p \neq 0; \)

\( \ell_1: \text{while}(n \geq 0) \)
\quad \{
\quad \ell_2:
        \quad \text{if}(n = 0) \)
        \quad \{
        \quad \ell_3: \quad \text{p} := 0;
        \quad \}
\quad \}
\quad \ell_4: \quad n--;\)
\quad }
\ell_5: \quad
\(\ell_0: \text{assume } p \neq 0;\)
\(\ell_1: \text{while}(n \geq 0)\)
\{
\(\ell_2: \quad \text{assert } p \neq 0;\)
if\((n == 0)\)
\{
\(\ell_3: \quad p := 0;\)
\}
\(\ell_4: \quad n--;\)
\}
\(\ell_5: \)
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 \( \ell_5 \) goes via \( \ell_2 \) and \( \ell_1 \). The sequence of statements on this path is infeasible (it does not have a possible execution) because it is not executable (the execution violates assertion).

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

**automaton**

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

The program \( P_{\text{ex1}} \) in Figure 1 is the adaptation of an example in [17] to our setting. In our setting we use `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

\[
(p \neq 0) \\
(n \geq 0) \\
(p = 0)
\]
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 \textit{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 \textit{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 \textit{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 \textit{assert} statement cannot fail). If the \textit{then} branch of the conditional is executed, then the value of \( n \) is 0, the statement \textit{n--} decrements the value of \( n \) from 0 to 1, and the while loop will exit directly, without executing the \textit{assert} statement.

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

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 \((p \neq 0)\) \((n \geq 0)\) \((p = 0)\) \((p \neq 0)\) \((p==0)\) \((p==0)\).
(p \neq 0)

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

A sequence of statements is not accepted by $A_1$ if it contains $p \neq 0$ and $p = 0$ with an update of $p$ in between. The shortest path from $q_0$ to $q_{\text{err}}$ with such a sequence of statements goes from $q_2$ to $q_{\text{err}}$ 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 $n_0 \geq 0$).
Fig. 2: Automata $A_1$ and $A_2$ which are a proof of correctness for $P_{ex1}$ (an edge labelled with $\land$ means a transition reading any letter, an edge labeled with $\land \cap \{p := 0\}$ 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 $n = 0, n_0 = n_1, \text{and } n_0$).

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 $(p \neq 0)$ and $(p = 0)$.
automaton constructed from unsatisfiability proof

![Automaton Diagram]

accepts all traces with the same unsatisfiability proof
this path is on the corresponding path in the rectness argument applies. We will next illustrate this in the example of automaton characterizes the case of exactly the executions for which the cor-
we can construct an automaton for a given correctness argument so that the
use automata. For one thing, we can use automata as an expressive means to
n cannot fail). If the
the value of
gets executed at least once during the execution or it does not. If not, then
into two cases, namely according to whether the
while loop when the value of
is 0, the statement
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
ex1
P

Fig. 1: Example program

 assert p != 0;

 P

 Fig. 2: Automata

 assert p != 0;

 P

 does a proof exist for every trace ?
Incremental Construction

\[ \mathcal{A}_1, \ldots, \mathcal{A}_n \]

` a la CEGAR

Program \( \mathcal{P} \)

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

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

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

\( w \) infeasible?

\[ w \in \mathcal{A}_{\mathcal{P}} \setminus (\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n) \]

\( \mathcal{P} \) is correct

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

The program $P_{ex1}$ 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 $(p \neq 0)$ $(n \geq 0)$ $(n == 0)$ $(p := 0)$ $(n--)$ $(n \geq 0)$ $(p == 0)$

**new trace:**

$$ (p \neq 0) $$
$$ (n \geq 0) $$
$$ (n == 0) $$
$$ (p := 0) $$
$$ (n--) $$
$$ (n \geq 0) $$
$$ (p == 0) $$
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 \textit{assert} statements to define the correctness of the program executions. In the example of $P_{\text{ex1}}$, an incorrect execution would start with a non-zero value for the variable $p$ and, at some point, enter the body of the while loop when the value of $p$ is 0 (and the execution of the \textit{assert} statement fails).

We can argue the correctness of $P_{\text{ex1}}$ rather directly if we split the executions into two cases, namely according to whether the \textit{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 \textit{assert} statement cannot fail). If the \textit{then} branch of the conditional is executed, then the value of $n$ is 0, the statement $n--$ decrements the value of $n$ from 0 to 1, and the while loop will exit directly, without executing the \textit{assert} statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $P_{\text{ex1}}$.
(n == 0)
(n--)
(n >= 0)
Example 1: automata from infeasibility proofs

We will describe an execution of the program

```plaintext
4
2
1
0
```

which are a proof of correctness for the program executions. In the example of the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of an automaton which each recognizes the set of all sequences of statements that contain the statements to define the correctness of the program.

We can infer a case split like the one above automatically. The key is to construct the corresponding correctness arguments, characterized di-

The infeasible path is shortest path from

in between.

The sequence of statements on this path is infeasible for a new reason: it cannot fail. If the case split and then constructing the corresponding correctness arguments, characterize di-

Infeasible

Fig. 1: Example program

The infeasibility, and constructed an automaton which each recognizes the set of all sequences of statements that contain the statements

\(p \Leftarrow 0\)

\(p\)

\(n \Leftarrow 0\)

\(n > 0\)

\(n \leftarrow 0\)

\(n \Leftarrow 0\)

\(n \Rightarrow 0\)

\(n \Leftarrow 0\)

\(n \Rightarrow 0\)

```plaintext
if (n == 0)
assert p != 0;
\}
while (n >= 0)
assume p != 0;
n--
```
Example 1: automata from infeasibility proofs

```
{ n == 0 }
while(n >= 0)
{ if(n == 0)
  assert p != 0;
  break;

  p := 0
  n--;
}
```

We will describe an execution of the program infeasible in between.

We construct the automaton depicted in Figure 2 which recognizes the set of all sequences of statements that contain the statements to define the correctness of the program executions. In the example of the control flow graph in Figure 1 is the adaptation of an example in [17] to our

To summarize, we have twice taken a path from $p_0$ to $p_3$, analyzed the reason why $p_0$ is never changed and remains non-zero (and the assert statement fails while loop when the value of $p$ is 0, the statement

We can construct an automaton for a given correctness argument so that the characterizes 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 construct an automaton for a given correctness argument so that the characterizes different cases of execution paths. For another, instead of first fixing

We can argue the correctness of while loop when the value of

branch of the conditional is executed, then the value of $n$ will exit directly, without executing the

expression and, at some point, enter the body of the

sequence of statements goes from

and the

set of all sequences of statements that contain

the inconsistency of

of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts

except for

except for

is not possible to execute the

in between. The shortest path from

is 0, the statement

p := 0
n--

is 0 (and the execution of the

p

statements to define the correctness of the

without an update of

p

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

serializes the situation where

for the same reason as above (i.e., the inconsistency of

of sequences of statements that contain

p

and

is 0, the statement

p

statement.

and

is not possible to execute the

serializes the situation where

for the same reason as above (i.e., the inconsistency of

of sequences of statements that contain

p

and

is 0, the statement

p

statement.

and

is not possible to execute the

serializes the situation where

for the same reason as above (i.e., the inconsistency of

of sequences of statements that contain

p

and

is 0, the statement

p

statement.

and

is not possible to execute the

serializes the situation where

for the same reason as above (i.e., the inconsistency of

of sequences of statements that contain

p

and

is 0, the statement

p

statement.

and

is not possible to execute the

serializes the situation where

for the same reason as above (i.e., the inconsistency of

of sequences of statements that contain

p

and

is 0, the statement

p

statement.
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. For another, instead of first fixing ex1, we can use automata as an expressive means to argue the correctness of the loop will exit directly, without executing the statement. If the value of n is never changed and remains non-zero (and the assert statement fails), then 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 ex1, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing ex1, we can use automata as an expressive means to characterize different cases of execution paths.
Incremental Construction

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

Program \( \mathcal{P} \)

\[ \mathcal{A}_\mathcal{P} \subseteq \bigcup_{i=1}^{n} A_i \] ?

\( w \) infeasible?

Yes

\[ \mathcal{A}_\mathcal{P} \subseteq \bigcup_{i=1}^{n} A_i \] ?

No

\( \mathcal{P} \) is correct

\( \mathcal{P} \) is incorrect

\[ w \in \mathcal{A}_\mathcal{P} \setminus \bigcup_{i=1}^{n} A_i \]

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

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

No

\( \mathcal{P} \) is incorrect

Yes

\( \mathcal{P} \) is correct
automata constructed from unsatisfiable core

are not sufficient in general

(verification algorithm not complete)
• new paradigm for automatic verification
• automata
• Marc Segelken: $\omega$-Cegar  [CAV 2007]
• verification for networked traffic control systems
\( \ell_0: x := 0; \)
\( \ell_1: y := 0; \)
\( \ell_2: \text{while}(\text{nondet}) \{x++;\} \)
   assert(x != -1);
   assert(y != -1);
We use them in the same way as above in order to construct the automaton paths that reach the error location via the edge labeled with proof.

It has three states, one for each assertion: the initial state, the state corresponding to the assertion that holds after the update to the location involved justification is required. The sequence of the two statements is preserved if one adds statements that do not modify any of the variables of the program.

Example 2: automata from sets of Hoare triples

Can one automatically check that every possible execution of the program is feasible? – The corresponding decision problem is undecidable. We have constructed an automaton which recognizes the set of all words for which the infeasibility of a sequence of statements (such as the sequence in Example 1: $P_1$, $P_2$, $x=-1$) can, however, check a condition which is stronger, namely that all sequences of statements on paths from one of the two cases? – The corresponding decision problem is undecidable. We have twice taken the set of Hoare triples comes from an interpolation SMT solver \cite{6} which generates the assertion corresponding to one path from in the preceding example can only have self-loops.

For $\{x \neq -1\}$, the state $q$ accepts a word exactly if the word labels a path from the initial state to a final state (i.e., the automaton...
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 $P_{ex\,1}$ 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 $P_{ex\,1}$ 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 $P_{ex\,1}$ (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 $P_{ex\,1} \vDash A_1 \sqsubseteq A_2$.

To rephrase our summary in the terminology of automata, we have twice taken a word accepted by the automaton $P_{ex\,1}$, 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 $P_{ex\,2}$ 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 $x==-1$.

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

Hoare triples proving infeasibility:
infeasibility $\iff$ pre/postcondition pair $(true, false)$
Hoare triples \(\longrightarrow\) automaton

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

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

sequencing of Hoare triples  \[\longrightarrow\] run of automaton
inference rule for sequencing

\[
\{p\} \ s \ \{q'\} \\
\{q'\} \ s' \ \{q\} \\
\hline \\
\{p\} \ s ; s' \ \{q\}
\]
proof space

infinite space of Hoare triples “\{pre\} trace \{post\}”

closed under inference rule of sequencing

generated from finite basis of Hoare triples “\{pre\} stmt \{post\}”
proof of sample trace:

\[
\begin{align*}
\{ \text{true} \} & \quad \text{x:=0} \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad \text{y:=0} \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad \text{x++} \quad \{ x \geq 0 \} \\
\{ x \geq 0 \} & \quad \text{x==−1} \quad \{ \text{false} \}
\end{align*}
\]
finite basis of Hoare triples \("\{pre\} stmt \{post\}\)"

can be obtained from proofs of sample traces

proof space

infinite space of Hoare triples \("\{pre\} trace \{post\}\)"

closed under inference rule of sequencing
finite basis of Hoare triples “{pre} stmt {post}” $\longrightarrow$ automaton

{ 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 \}

sequencing of Hoare triples in basis $\longrightarrow$ run of automaton
proof space contains \( \{\text{true}\} \) trace \( \{\text{false}\} \)

if

exists sequencing of Hoare triples in basis

if

exists accepting run of automaton

proof space

infinite space of Hoare triples \( \{\text{pre}\} \) trace \( \{\text{post}\} \)

closed under inference rule of sequencing

generated from finite basis of Hoare triples \( \{\text{pre}\ \text{stmt} \} \text{post} \)
paradigm:

- construct proof space
- check proof space
simplify task for program verification:

Don’t give a proof.
Show that a proof exists.
inclusion check:
show that, for every word in the given set,
an accepting run exists
simplify task for program verification:

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