Regular Abstraction
An Alternative to Predicate Abstraction

Matthias Heizmann

University of Freiburg

October 2008
Task: Safety Check/Reachability analysis

Well known methods:
Task: Safety Check/Reachability analysis

Well known methods:

- Model Checking
  e.g. transition system, search error state
  Problem: state explosion, infinite

Idea of predicate abstraction: Combine both approaches.
Task: Safety Check/Reachability analysis

Well known methods:

- Model Checking
  e.g. transition system, search error state
  Problem: state explosion, infinite

- Theorem Proving
  e.g. hoare calculus,
  undecideable if we use expressive logic
  so not automatic

Idea of predicate abstraction: Combine both approaches.
Task: Safety Check/Reachability analysis

Well known methods:

- Model Checking
  e.g. transition system, search error state
  Problem: state explosion, infinite

- Theorem Proving
  e.g. hoare calculus,
  undecideable if we use expressive logic
  so not automatic

Idea of predicate abstraction: Combine both approaches.
0: if (x<5) {
1:     x++
2:     assert x<6
3: }
4: some other code
\[ \neg x \geq 5 \]
\[ \neg x \geq 6 \]
\[ x \geq 5 \]
\[ x \geq 6 \]
\[ \neg x \geq 5 \]
\[ \neg x \geq 6 \]
\[ x \geq 5 \]
\[ \neg x \geq 6 \]

\[ x \geq 5 \]
\[ x \geq 6 \]

\[ \pre(x \geq 5 \land x < 6, x++) \land x < 5 \land x < 6 \]
\[ = x \geq 4 \land x < 5 \land x < 5 \land x < 6 \]
satisfiable!
Input: A program $\mathcal{P}$

**build abstraction:**
build automaton $A_i$ that represents $(\mathcal{P}, p_1, \ldots, p_n)$

**verify abstraction:**
is $A_i$ safe?

**spuriousness check:**
is $\pi$ feasible?
$pre(true, \pi)$ satisfiable?

**choose refinement:**
choose new predicates $p_{i1}, \ldots, p_{im}$

$\pi$ spurious

$\mathcal{P}$ is not safe
+ counterexample $\pi$

$\mathcal{P}$ is safe

$\pi$ is feasible
not safe + counterexample $\pi$

$\pi$ is feasible

Matthias Heizmann | Regular Abstraction
Predicate Abstraction is Inefficient

Check if counterexample is spurious.

\[
pre(true, x<5; \ x++; \ x\geq 6) = x < 5 \land x \geq 5
\]

unsatisfiable
Predicate Abstraction is Inefficient

Check if counterexample is spurious.

\[ \text{pre}(true, x < 5; x++; x \geq 6) = x < 5 \land x \geq 5 \]

unsatisfiable

Build the abstraction.

\[ x < 6, x < 5 \quad \xrightarrow{x++; \quad x++} \quad x \geq 6, x \geq 5 \]
Predicate Abstraction is Inefficient

Check if counterexample is spurious.

$$\text{pre}(true, x<5; x++; x\geq 6) = x < 5 \land x \geq 5$$

unsatisfiable

Build the abstraction.

$$\begin{align*}
x < 6, x < 5 \\
x \geq 6, x \geq 5
\end{align*}$$

$$\text{pre}(x \geq 6 \land x \geq 5, x++) \land x < 5 \land x < 6$$

$$= x \geq 5 \land x \geq 4 \land x < 5 \land x < 6$$

unsatisfiable
Flaws of Predicate Abstraction

- Slow, a lot of theorem proving necessary
Flaws of Predicate Abstraction

- Slow, a lot of theorem proving necessary
- when combining with cegar:
  - theorem proving to choose new predicates
  - theorem proving to build new abstraction
  - every CEGAR iteration - build a new abstraction from scratch
“A clever person solves a problem. A wise person avoids it.”
Albert Einstein (1879-1955)
Consider CFG as finite automaton.

\[ \mathcal{L}(P) = \{x<5; x++; x\geq 6\} \]
Consider CFG as finite automaton.

\[ \mathcal{L}(\mathcal{P}) = \{ x < 5; x++; x \geq 6 \} \]

Definition

We define a Regular Abstraction of

- a Program \( \mathcal{P} = (\text{LOC}, \delta, \text{LOC}_{\text{init}}, \text{LOC}_{\text{err}}) \)
- and a set of regular expressions \( \alpha_1, \ldots, \alpha_n \)

very declarative, by its Error Traces

\[ \Pi^{\text{reg}}_{\text{err}}(\mathcal{P}, \alpha_1, \ldots, \alpha_n) := \mathcal{L}(\mathcal{P}) \setminus \bigcup_{i=1}^{n} \mathcal{L}(\alpha_i) \]
\[ \Pi_{\text{err}}^{\text{reg}}(\mathcal{P}, \alpha_1, \ldots, \alpha_n) = \mathcal{L}(\mathcal{P}) \setminus \mathcal{L}(\alpha_1 \mid \ldots \mid \alpha_n) \]
\[ \Pi_{\text{err}}^{\text{reg}}(\mathcal{P}, \alpha_1, ..., \alpha_n) = \mathcal{L}(\mathcal{P}) \setminus \mathcal{L}(\alpha_1 | ... | \alpha_n) \]

\[ \sum^* \]

NFA accepting \( \mathcal{L}(\alpha_1 | ... | \alpha_n) \)
\[ \Pi_{\text{err}}^{\text{reg}}(\mathcal{P}, \alpha_1, \ldots, \alpha_n) = \mathcal{L}(\mathcal{P}) \backslash \mathcal{L}(\alpha_1|\ldots|\alpha_n) \]

\[ \sum^* \]

NFA accepting \( \mathcal{L}(\alpha_1|\ldots|\alpha_n) \)

\[ \downarrow \]

\[ \downarrow \text{negation} \]

DFA accepting \( \sum^* \backslash \mathcal{L}(\alpha_1|\ldots|\alpha_n) \)
\[ \Pi_{\text{err}}^{\text{reg}}(\mathcal{P}, \alpha_1, \ldots, \alpha_n) = \mathcal{L}(\mathcal{P}) \setminus \mathcal{L}(\alpha_1 | \ldots | \alpha_n) \]

NFA accepting \( \mathcal{L}(\alpha_1 | \ldots | \alpha_n) \)
\[ \downarrow \]
\[ \downarrow \text{negation} \]
DFA accepting \( \Sigma^* \setminus \mathcal{L}(\alpha_1 | \ldots | \alpha_n) \)
\[ \downarrow \]
\[ \downarrow \text{product with } \mathcal{P} \]
DFA accepting \( \Pi_{\text{err}}^{\text{reg}}(\mathcal{P}, \alpha_1, \ldots, \alpha_n) \)
Task: Exclude the spurious counterexample \( x<5; \ x++; \ x>=6 \)
Task: Exclude the spurious counterexample  \( x<5; \ x++; \ x>=6 \)
Task: Exclude the spurious counterexample \( x<5 \); \( x++ \); \( x>=6 \)
Example - Build Abstraction

\[ x < 5 \]
\[ x++ \]
\[ x \geq 6 \]
\[ x \geq 5 \]
\[ x < 6 \]
\[ \sum \{ x < 6 \} \]
\[ \sum \{ x++ \} \]
\[ \sum \{ x \geq 6 \} \]
\[ \sum \{ x = \} \]

Accepts $\Pi_{error}(x < 5; x++; x \geq 6) = \emptyset$

Matthias Heizmann

Regular Abstraction
Accepts $\Pi_{error}(x<5; x++; x>=6) = \emptyset$
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  Can build abstraction without theorem proving
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  - Can build abstraction without theorem proving
- Reuse old abstraction in refinement
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  Can build abstraction without theorem proving
- Reuse old abstraction in refinement
- Enables new heuristics to speed up verification
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  - Can build abstraction without theorem proving
- Reuse old abstraction in refinement
- Enables new heuristics to speed up verification
- Regular Abstraction $\supseteq$ Predicate Abstraction
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  Can build abstraction without theorem proving
- Reuse old abstraction in refinement
- Enables new heuristics to speed up verification
- $\text{Regular Abstraction} \supseteq \text{Predicate Abstraction}$
- $\text{Regular Abstraction} \not= \text{Predicate Abstraction}$
Advantages of Regular Abstraction over Predicate Abstraction

- Economical use of theorem proving
  - Can build abstraction without theorem proving
- Reuse old abstraction in refinement
- Enables new heuristics to speed up verification
- Regular Abstraction $\supseteq$ Predicate Abstraction
- Regular Abstraction $\neq$ Predicate Abstraction
- We can combine both approaches
Predicate Abstraction for Program 
\( P = (LOC, \delta_P, LOC_{init}, LOC_{error}) \) and predicates \( p_1, \ldots, p_n \) is 

\[
A^{pred}_{p_1,\ldots,p_n} = (LOC \times \{0,1\}^n, \delta_{pred}, LOC_{init} \times \{0,1\}^n, LOC_{error} \times \{0,1\}^n)
\]

\[
\ell' \in \delta_P(\ell, st), \ pre\left( \bigwedge_{i=1}^{n} (b'_i \leftrightarrow p_i), st \right) \land \bigwedge_{i=1}^{n} (b_i \leftrightarrow p_i) \text{ is satisfiable}
\]

\[
(\ell', b'_1, \ldots, b'_n) \in \delta_{Pred}\left((\ell, b_1, \ldots, b_n), st\right)
\]
Predicate Abstraction for Program

\( \mathcal{P} = (\mathit{LOC}, \delta_{\mathcal{P}}, \mathit{LOC}_{\text{init}}, \mathit{LOC}_{\text{error}}) \) and predicates \( p_1, \ldots, p_n \) is

\[
\mathcal{A}^{\text{pred}}_{p_1,\ldots,p_n} = (\mathit{LOC} \times \{0,1\}^n, \delta_{\text{pred}}, \mathit{LOC}_{\text{init}} \times \{0,1\}^n, \mathit{LOC}_{\text{error}} \times \{0,1\}^n)
\]

\[
\ell' \in \delta_{\mathcal{P}}(\ell, st), \quad \text{pre}\left( \bigwedge_{i=1}^{n} (b'_i \leftrightarrow p_i), \quad st \right) \land \bigwedge_{i=1}^{n} (b_i \leftrightarrow p_i) \text{ is satisfiable}
\]

\[
(l', b'_1, \ldots, b'_n) \in \delta_{\text{Pred}}((\ell, b_1, \ldots, b_n), st)
\]

\[
\mathcal{A}^{\text{sim}}_{p_1,\ldots,p_n} = (\{0,1\}^n, \delta_{\text{sim}}, \{0,1\}^n, \{0,1\}^n)
\]

\[
\text{pre}\left( \bigwedge_{i=1}^{n} (b'_i \leftrightarrow p_i), \quad st \right) \land \bigwedge_{i=1}^{n} (b_i \leftrightarrow p_i) \text{ is satisfiable}
\]

\[
(b'_1, \ldots, b'_n) \in \delta((b_1, \ldots, b_n), st)
\]
Predicate Abstraction for Program

\[ P = (\text{LOC}, \delta_P, \text{LOC}_{\text{init}}, \text{LOC}_{\text{error}}) \]

and predicates \( p_1, \ldots, p_n \) is

\[ \mathcal{A}_{p_1, \ldots, p_n}^{\text{pred}} = (\text{LOC} \times \{0, 1\}^n, \delta_{\text{pred}}, \text{LOC}_{\text{init}} \times \{0, 1\}^n, \text{LOC}_{\text{error}} \times \{0, 1\}^n) \]

\[ \ell' \in \delta_P(\ell, \text{st}) \land \text{pre}(\bigwedge_{i=1}^{n} (b_i' \leftrightarrow p_i), \text{st}) \land \bigwedge_{i=1}^{n} (b_i' \leftrightarrow p_i) \text{ is satisfiable} \]

\[ (\ell', b_1', \ldots, b_n') \in \delta_{\text{Pred}}((\ell, b_1, \ldots, b_n), \text{st}) \]

\[ \mathcal{A}_{p_1, \ldots, p_n}^{\text{sim}} = (\{0, 1\}^n, \delta_{\text{sim}}, \{0, 1\}^n, \{0, 1\}^n) \]

\[ \text{pre}(\bigwedge_{i=1}^{n} (b_i' \leftrightarrow p_i), \text{st}) \land \bigwedge_{i=1}^{n} (b_i' \leftrightarrow p_i) \text{ is satisfiable} \]

\[ (b_1', \ldots, b_n') \in \delta((b_1, \ldots, b_n), \text{st}) \]

\[ \mathcal{A}_{p_1, \ldots, p_n}^{\text{pred}} = P \times \mathcal{A}_{p_1, \ldots, p_n}^{\text{sim}} \]
Regular Abstraction \supseteq \text{Predicate Abstraction}

\[ (0, 0) \quad x < 5, x < 6 \]
\[ (0, 1) \quad x < 5, x \geq 6 \]
\[ (1, 0) \quad x \geq 5, x < 6 \]
\[ (1, 1) \quad x \geq 5, x \geq 6 \]

\( x++ \), \( x < 5 \), \( x < 6 \)
\( x++ \)
\( x => 5 \), \( x < 6 \)

\( A_{\text{Sim}}^{x \geq 5, x \geq 6} \)

for our example
Regular Abstraction $\neq$ Predicate Abstraction
Regular Abstraction ≠ Predicate Abstraction

\[ \Pi^{reg}_{err}(x := 0; x = 0; x != 0) = \mathcal{L}(x := 0; x := x; x != 0) \]
Regular Abstraction ≠ Predicate Abstraction

\[ \prod_{err}^{reg}(x := 0; x \neq 0, x := 0; x = 0) = \mathcal{L}(x := 0; x := x; + x \neq 0) \]

\[ s_0 \xrightarrow{x := 0} s_1 \xrightarrow{x \neq 0} s_2 \xrightarrow{\Sigma} \]

Diagram: Two states with transitions labeled by assignment and test conditions.
Regular Abstraction ≠ Predicate Abstraction

\[
\prod_{\text{err}}^{\text{reg}}(s_0 \xrightarrow{x:=0} s_1 \xrightarrow{x\neq0} s_2) = \mathcal{L}(x:=0; x=x;^+ x\neq0)
\]

Σ \{x:=0\} \xrightarrow{x:=0} s_0 \xrightarrow{s_1} \xrightarrow{s_2} Σ \{x\neq0\}

X := x

x := 0

x := 0; x != 0

Π_{\text{err}}^{\text{reg}}(s_0 \xrightarrow{x:=0} s_1 \xrightarrow{x\neq0} s_2) = \mathcal{L}(x:=0; x=x;^+ x\neq0)
Regular Abstraction \neq\text{Predicate Abstraction}

\[
\Pi_{err}^{reg}(\text{x:=0; x\neq 0}) = L(\text{x:=0}; \text{x:=x};^+ \text{x\neq 0})
\]
build abstraction: 
build automaton $A_i$ that accepts $\Pi_{\text{reg}}(P, \alpha_1, ..., \alpha_i)$

choose refinement: 
choose regular expression $\alpha_{i+1}$ that describes only unfeasible paths and especially $\pi$

spuriousness check: 
is $\pi$ feasible? 
$\text{pre}(true, \pi)$ satisfiable?

verify abstraction: 
$L(A_i) = \emptyset$ ?

$\pi$ spurious

$\pi$ is feasible 
not safe + counterexample $\pi$

$P$ is not safe + counterexample $\pi$

$P$ is safe 
abstraction is safe

Input: A program $P$

$i := 0$

$i++$
Given: A spurious counterexample $\pi$

Task: Choose regular expression $\alpha$ such that

1. $\pi \in \mathcal{L}(\alpha)$
2. $\mathcal{L}(\alpha)$ is set of unfeasible sequences of statements
Given: A spurious counterexample $\pi$
Task: Choose regular expression $\alpha$ such that

- $\pi \in \mathcal{L}(\alpha)$
- $\mathcal{L}(\alpha)$ is set of unfeasible sequences of statements

1. Iteration: Exclude $x:=0; x<0$
2. Iteration: Exclude $x++; x:=0; x<0$
3. Iteration: Exclude $x++; x++; x:=0; x<0$
Choose Refinement

Given: A spurious counterexample $\pi$
Task: Choose regular expression $\alpha$ such that

- $\pi \in L(\alpha)$
- $L(\alpha)$ is set of unfeasible sequences of statements

1. Iteration: Exclude $x := 0; \ x < 0$
2. Iteration: Exclude $x++; x := 0; \ x < 0$
3. Iteration: Exclude $x++; x++; x := 0; \ x < 0$
Given: A spurious counterexample $\pi$
Task: Choose regular expression $\alpha$ such that
- $\pi \in \mathcal{L}(\alpha)$
- $\mathcal{L}(\alpha)$ is set of unfeasible sequences of statements

1. Iteration: Exclude $x:=0; x<0$
2. Iteration: Exclude $x++; x:=0; x<0$
Choose Refinement

Given: A spurious counterexample $\pi$

Task: Choose regular expression $\alpha$ such that

- $\pi \in \mathcal{L}(\alpha)$
- $\mathcal{L}(\alpha)$ is set of unfeasible sequences of statements

1. Iteration: Exclude $x:=0; x<0$
2. Iteration: Exclude $x++; x:=0; x<0$
3. Iteration: Exclude $x++; x++; x:=0; x<0$
Given: A spurious counterexample $\pi$

Task: Choose regular expression $\alpha$ such that

- $\pi \in \mathcal{L}(\alpha)$
- $\mathcal{L}(\alpha)$ is set of unfeasible sequences of statements
- it would be nice if $\mathcal{L}(\alpha)$ contains traces that would become counterexamples if not excluded now.

1. Iteration: Exclude $x:=0; x<0$
2. Iteration: Exclude $x++; x:=0; x<0$
3. Iteration: Exclude $x++; x++; x:=0; x<0$
Mark Segelkens Approach
find minimal unfeasible coherent subsequence

counterexample $\pi = st_0 \ st_1 \ st_2 \ st_3 \ st_4 \ st_5 \ st_6 \ st_7 \ st_8 \ st_9$
counterexample $\pi = st_0 \ s t_1 \ s t_2 \ s t_3 \ s t_4 \ s t_5 \ s t_6 \ s t_7 \ s t_8 \ s t_9$

$pre(true, \ s t_2 \ s t_3 \ s t_4 \ s t_5 \ s t_6 \ s t_7)$ is unsatisfiable

$pre(true, \ s t_3 \ s t_4 \ s t_5 \ s t_6 \ s t_7)$ is satisfiable

$pre(true, \ s t_2 \ s t_3 \ s t_4 \ s t_5 \ s t_6)$ is satisfiable
counterexample $\pi = st_0 \; st_1 \; st_2 \; st_3 \; st_4 \; st_5 \; st_6 \; st_7 \; st_8 \; st_9$

$$\text{pre}(true, \; st_2 \; st_3 \; st_4 \; st_5 \; st_6 \; st_7)$$ is unsatisfiable

$$\text{pre}(true, \; st_3 \; st_4 \; st_5 \; st_6 \; st_7)$$ is satisfiable

$$\text{pre}(true, \; st_2 \; st_3 \; st_4 \; st_5 \; st_6)$$ is satisfiable

So we can exclude $\Sigma^* \; st_2 \; st_3 \; st_4 \; st_5 \; st_6 \; st_7 \; \Sigma^*$
minimal unfeasible coherent subsequence of a counterexample

\[ st_2 \ st_3 \ st_4 \ st_5 \ st_6 \ st_7 \]
minimal unfeasible coherent subsequence of a counterexample

\[ st_2 \ st_3 \ st_4 \ st_5 \ st_6 \ st_7 \]

\[ \text{pre}(true, \ st_2 \ st_4 \ st_7) \] is unsatisfiable
\[ \text{pre}(true, \ st_2 \ st_7) \] is satisfiable

and \( st_3, \ st_5, \ st_6 \) do not manipulate variables “that are responsible” for the conflict.
minimal unfeasible coherent subsequence of a counterexample

\[ \text{pre}(true, \, st_2 \, st_4 \, st_7) \] is unsatisfiable
\[ \text{pre}(true, \, st_2 \, st_7) \] is satisfiable

and \( st_3, \, st_5, \, st_6 \) do not manipulate variables “that are responsible” for the conflict.

Then we can exclude \( \Sigma^* st_2 \, \Gamma^*_{st_4 st_7} \, st_4 \, \Gamma^*_{st_7 st_7} \, \Sigma^* \)

Where \( \Gamma_{st} \) is the set of all statements except assignments that change variables occurring in \( st \).
How to deal with loops

If we find $\phi_{inv}$ such that $\phi_{inv} \Rightarrow \text{pre}(\text{true}, \text{ste})$ then we can exclude $\Sigma^* \text{ste}^* \Sigma^*$.
How to deal with loops

If we find \( \phi^{\text{inv}} \) such that

\[
\phi^{\text{inv}} \Rightarrow \text{pre}(\text{true}, \text{st})
\]

\[
\phi^{\text{inv}} \Rightarrow \text{pre}(\phi^{\text{inv}}, \text{st})
\]

\[
\text{pre}(\phi^{\text{inv}}, \text{st}) \text{ unsatisfiable}
\]

then we can exclude \( \Sigma^{\ast} \text{st}^{\ast} \Sigma^{\ast} \).

\[
st_{b} \, \, st_{l} \, \, st_{e}
\]
How to deal with loops

If we find $\phi_{inv}$ such that $\phi_{inv} \Rightarrow \text{pre}(\text{true}, \text{st})$, then we can exclude $\Sigma^\ast$. 

$\text{st}_b \text{ st}_l \text{ st}_e$

$\text{st}_b \text{ st}_l \text{ st}_l \text{ st}_e$
How to deal with loops

If we find $\varphi_{\text{inv}}$ such that $\varphi_{\text{inv}} \Rightarrow \text{pre}(\text{true}, \text{st})$

$\varphi_{\text{inv}} \Rightarrow \text{pre}(\varphi_{\text{inv}}, \text{st})$

$\text{pre}(\varphi_{\text{inv}}, \text{st}_{b})$ unsatisfiable

then we can exclude $\Sigma_{\text{st}} \Sigma_{\text{st}}$.

Matthias Heizmann

Regular Abstraction
How to deal with loops

If we find $\varphi_{inv}$ such that
- $\varphi_{inv} \Rightarrow \text{pre}(true, st_e)$
- $\varphi_{inv} \Rightarrow \text{pre}(\varphi_{inv}, st_l)$
- $\text{pre}(\varphi_{inv}, st_b)$ unsatisfiable
then we can exclude

$\Sigma^* st_b st_l^* st_e \Sigma^*$
Model Checking as a Game

Consider this as a finite two-person game on a directed graph. Player 0 plays in \( P \) and Player 1 plays in \( NFA_{\alpha_1|...|\alpha_n} \). Players must move alternately along transitions. Player 0 begins, Player 1 must take an equally labelled transition as Player 0. Player 0 wins if he reaches an accepting state. Player 1 cannot move. Player 1 wins if he reaches an accepting state. Player 0 cannot move.

If Player 0 has a winning strategy there is a counterexample.

CFG of Program \( P \)

Traces we want to exclude \( \alpha_1, ..., \alpha_n \)

We searching \( \pi \) such that

\( \pi \) accepted by \( P \)

\( \pi \) not accepted by \( NFA_{\alpha_1|...|\alpha_n} \)
CFG of Program $\mathcal{P}$

Traces we want to exclude $\alpha_1, \ldots, \alpha_n$

We searching $\pi$ such that

$\pi$ accepted by $\mathcal{P}$

$\pi$ not accepted by $NFA_{\alpha_1|\ldots|\alpha_n}$

Consider this as a finite two-person game on a directed graph.
CFG of Program $\mathcal{P}$

Traces we want to exclude $\alpha_1, \ldots, \alpha_n$

We searching $\pi$ such that

$\pi$ accepted by $\mathcal{P}$
$\pi$ not accepted by $\text{NFA}_{\alpha_1|\ldots|\alpha_n}$

Consider this as a finite two-person game on a directed graph.

Player0 plays in $\mathcal{P}$
Player1 plays in $\text{NFA}_{\alpha_1|\ldots|\alpha_n}$

Players must move alternately along transitions. Player0 begins, Player1 must take an equally labelled transition as Player0.
Model Checking as a Game

CFG of Program $\mathcal{P}$

Traces we want to exclude $\alpha_1, \ldots, \alpha_n$

We searching $\pi$ such that

$\pi$ accepted by $\mathcal{P}$

$\pi$ not accepted by $\text{NFA}_{\alpha_1|\ldots|\alpha_n}$

Consider this as a finite two-person game on a directed graph.

Player0 plays in $\mathcal{P}$

Player1 plays in $\text{NFA}_{\alpha_1|\ldots|\alpha_n}$

Players must move alternately along transitions. Player0 begins, Player1 must take an equally labelled transition as Player0.

Player0 wins if
- he reaches an accepting state.
- Player1 cannot move

Player1 wins if
- he reaches an accepting state.
- Player0 cannot move
- infinite games

Matthias Heizmann | Regular Abstraction
Model Checking as a Game

CFG of Program $\mathcal{P}$

Traces we want to exclude $\alpha_1, \ldots, \alpha_n$

We searching $\pi$ such that

- $\pi$ accepted by $\mathcal{P}$
- $\pi$ not accepted by $NFA_{\alpha_1|\ldots|\alpha_n}$

Consider this as a finite two-person game on a directed graph.

Player0 plays in $\mathcal{P}$

Player1 plays in $NFA_{\alpha_1|\ldots|\alpha_n}$

Players must move alternately along transitions. Player0 begins, Player1 must take an equally labelled transition as Player0.

Player0 wins if

- he reaches an accepting state.
- Player1 cannot move

Player1 wins if

- he reaches an accepting state.
- Player0 cannot move
- infinite games

If Player0 has a winning strategy there is a counterexample.
Regular Abstraction

- new abstraction technique, based on paths
- economical use of theorem proving
- strictly more expressive than predicate abstraction