Proofs That Count

Zachary Kincaid
Azadeh Farzan
Andreas Podelski

1 University of Toronto
2 University of Freiburg

January 22, 2014
Given a program $P$ and a specification $\varphi_{\text{pre}}/\varphi_{\text{post}}$, prove

$\{\varphi_{\text{pre}}\} P \{\varphi_{\text{post}}\}$
Software verification

Concurrent

Goal

Given a program $P$ and a specification $\varphi_{\text{pre}} / \varphi_{\text{post}}$, prove

$$\{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \}$$
Given a program $P$ and a specification $\varphi_{\text{pre}}/\varphi_{\text{post}}$, prove

$$\{\varphi_{\text{pre}}\} P \{\varphi_{\text{post}}\}$$
Given a program $P$ and a specification $\varphi_{\text{pre}}/\varphi_{\text{post}}$, prove

$$\{\varphi_{\text{pre}}\} P \{\varphi_{\text{post}}\}$$

- Proofs for concurrent programs sometimes make use of *counting* arguments.
Given a program $P$ and a specification $\varphi_{\text{pre}}/\varphi_{\text{post}}$, prove

$$\{\varphi_{\text{pre}}\} P \{\varphi_{\text{post}}\}$$

- Proofs for concurrent programs sometimes make use of counting arguments.
  - Readers/Writers protocol: “the number of active readers”
Given a program \( P \) and a specification \( \varphi_{\text{pre}} / \varphi_{\text{post}} \), prove

\[
\{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \}
\]

- Proofs for concurrent programs sometimes make use of *counting* arguments.
  - Readers/Writers protocol: “the number of active readers”
  - Ticket protocol: “the number of processes with a smaller ticket”
A counting argument is a proof that a program satisfies its specification which uses auxiliary *counters*:

- Can be used in assertions.
- *Auxiliary* (or *ghost*) variables: do not appear in the program. Think: Owicki-Gries.
Example

Precondition: \( \{ s = t = 0 \} \)
1: t++
2: \texttt{assert}(t > s)
3: s++
Example

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)

2: \texttt{assert}(t > s) \quad 2: \texttt{assert}(t > s)

3: \( s++ \)

There is no Owicki-Gries proof that does not use auxiliary variables.

Inductive invariant:

\[ \#_2 + \#_3 = t \]
Example

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)
2: \( \text{assert}(t > s) \)
3: \( s++ \)

There is no Owicki-Gries proof that does not use auxiliary variables.
Example

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)         1: \( t++ \)         1: \( t++ \)
2: \texttt{assert}(t > s) 2: \texttt{assert}(t > s) \ldots 2: \texttt{assert}(t > s)
3: \( s++ \)         3: \( s++ \)         3: \( s++ \)

There is no Owicki-Gries proof that does not use auxiliary variables. Inductive invariant:

\#2 + \#3 = t - s
Example

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)
2: \( \text{assert}(t > s) \)
3: \( s++ \)

Inductive invariant:

\[ #2 + #3 = t - s \]
Example

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)   
2: \( \text{assert}(t > s) \)   
3: \( s++ \)   

1: \( t++ \)   
2: \( \text{assert}(t > s) \)   
3: \( s++ \)   

There is no Owicki-Gries proof that does not use auxiliary variables.

Inductive invariant:

\[ \#2 + \#3 = t - s \]

# of threads at line 2

# of threads at line 3
How do we formalize counting arguments?
Challenges

How do we formalize counting arguments?

How do we synthesize counting arguments automatically?
Language-theoretic approach

Precondition: \( \{ s = t = 0 \} \)

1: \( t++ \)
2: \( \text{assert}(t > s) \)
3: \( s++ \)

1: \( t++ \)
2: \( \text{assert}(t > s) \)
3: \( s++ \)

...
Language-theoretic approach

Precondition: \( \{ s = t = 0 \} \)

Program:

\[ \begin{align*}
&\text{t++; [t \leq s]} \\
&\text{t++; t++; s++; [t \leq s]} \\
&\text{t++; s++; t++; [t \leq s]} \\
&\vdots
\end{align*} \]
Language-theoretic approach

Precondition: \( \{ s = t = 0 \} \)

Program:

\[
\begin{align*}
&\quad \text{t++; [t \leq s]} \\
&\quad \text{t++; t++; s++; [t \leq s]} \\
&\quad \text{t++; s++; t++; [t \leq s]} \\
&\quad \text{...}
\end{align*}
\]

Proof:

\[
\forall \tau \in \mathcal{L} (\text{Proof}). \{ \varphi_{\text{pre}} \} \tau \{ \varphi_{\text{post}} \}
\]
Language-theoretic approach

Precondition: \( \{ s = t = 0 \} \)

Program

\[
t++; [t \leq s]
\]
\[
t++; t++; s++; [t \leq s]
\]
\[
t++; s++; t++; [t \leq s]
\]

\[
\forall \tau \in \mathcal{L}(\text{Proof}). \{ \varphi_{\text{pre}} \} \tau \{ \varphi_{\text{post}} \}
\]

Proof
Language-theoretic approach

Precondition: \( \{ s = t = 0 \} \)

Program

\[
\begin{align*}
& \text{t++; } [t \leq s] \\
& \text{t++; t++; s++; } [t \leq s] \\
& \text{t++; s++; t++; } [t \leq s] \\
& \vdots \\
\end{align*}
\]

\( \forall \tau \in \mathcal{L}(\text{Proof}). \{ \varphi_{\text{pre}} \} \tau \{ \varphi_{\text{post}} \} \)

Proof rule

If there exists a Proof such that \( \mathcal{L}(\text{Program}) \subseteq \mathcal{L}(\text{Proof}) \), then

\[
\{ \varphi_{\text{pre}} \} \text{Program} \{ \varphi_{\text{post}} \}
\]
Counting proofs

Counting proof = counting automaton + inductive annotation
Counting proofs

Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional \( \mathbb{N} \)-valued counter variables. *Assume one counter variable for this talk.*
  Transitions are labeled by a counter action \( \in \{ \text{inc}, \text{dec}, \text{tst}, \text{nop} \} \)

\[
\begin{align*}
q_0 & \quad \text{start} \\
& \quad s+/dec \quad t+/inc \\
& \quad [t \leq s]/\text{tst} \\
& \quad q_0 \\
& \quad q_1 \\
& \quad s+/nop \quad t+/nop \\
& \quad [t \leq s]/\text{nop}
\end{align*}
\]
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables.
  
  Assume one counter variable for this talk.
  
  Transitions are labeled by a counter action $\in \{\text{inc}, \text{dec}, \text{tst}, \text{nop}\}$

\[ q_0 \]
\[ k = 0 \]
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. Assume one counter variable for this talk. Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$

![Diagram of a counting automaton](image)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. *Assume one counter variable for this talk.* Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$.
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional \( \mathbb{N} \)-valued counter variables. 
  Assume one counter variable for this talk.

Transitions are labeled by a counter action \( \in \{ \text{inc, dec, tst, nop} \} \)

![Diagram of counting automaton with transitions labeled by counter actions.]

\[
\begin{align*}
q_0 & \quad k = 0 \quad \xrightarrow{\text{inc}} \quad q_0 & \quad k = 1 \quad \xrightarrow{\text{dec}} \quad q_0 \quad k = 0 \quad \xrightarrow{\text{inc}} \quad q_0 \quad k = 1 \\
\end{align*}
\]

\[
\begin{align*}
q_0 & \quad k = 0 \quad \xrightarrow{\text{t++}} \quad q_0 & \quad k = 1 \quad \xrightarrow{\text{s++}} \quad q_0 \quad k = 0 \quad \xrightarrow{\text{t++}} \quad q_0 \quad k = 1 \\
\end{align*}
\]
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. Assume one counter variable for this talk. Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional \( \mathbb{N} \)-valued counter variables. 
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action \( \in \{\text{inc, dec, tst, nop}\} \)

![Diagram of counting automaton with transitions labeled by counter actions.](image_url)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional \( \mathbb{N} \)-valued counter variables. Assume one counter variable for this talk.

  Transitions are labeled by a counter action \( \in \{ \text{inc, dec, tst, nop} \} \)

```
q₀ \( k = 0 \) \xrightarrow{\text{inc}} q₀ \( k = 1 \) \xrightarrow{s++} q₀ \( k = 0 \) \xrightarrow{\text{dec}} X
```

```
q₁ \( k = 0 \) \xrightarrow{\text{tst}} q₁ \( k = 1 \) \xrightarrow{\text{inc}} q₁ \( k = 0 \) \xrightarrow{\text{dec}} X
```

Z. Kincaid (U. Toronto)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. 
  *Assume one counter variable for this talk.*
  Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$

- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)
Counting proofs

Counting proof = counting automaton + inductive annotation

- Counting automaton = DFA with additional \( \mathbb{N} \)-valued counter variables. **Assume one counter variable for this talk.**
  
  Transitions are labeled by a counter action \( \in \{ \text{inc, dec, tst, nop} \} \)

- Inductive annotation = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$
- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)

\[
\begin{align*}
q_0 & \xrightarrow{\text{inc}} q_0 & q_0 & \xrightarrow{\text{dec}} q_0 & q_0 & \xrightarrow{\text{inc}} q_0 & q_0 & \xrightarrow{\text{tst}} q_1 \\
  k = 0 & \{k = t - s\} & k = 1 & \{k = t - s\} & k = 0 & \{k = t - s\} & k = 1 & \{\text{false}\}
\end{align*}
\]
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. 
  *Assume one counter variable for this talk.*
  Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$

- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. *Assume one counter variable for this talk.* Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$
- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional $\mathbb{N}$-valued counter variables. 
  *Assume one counter variable for this talk.*
  Transitions are labeled by a counter action $\in \{\text{inc, dec, tst, nop}\}$

- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)
Counting proofs

Counting proof = counting automaton + inductive annotation

- **Counting automaton** = DFA with additional \( \mathbb{N} \)-valued counter variables. **Assume one counter variable for this talk.** Transitions are labeled by a counter action \( \in \{ \text{inc}, \text{dec}, \text{tst}, \text{nop} \} \)

- **Inductive annotation** = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)

\[
\begin{align*}
q_0 & \xrightarrow{k=0} q_0, \{0 = t - s\} \xrightarrow{\text{inc}} q_0, \{1 = t - s\} \xrightarrow{\text{dec}} q_0, \{0 = t - s\} \xrightarrow{\text{inc}} q_1, \{1 = t - s\} \xrightarrow{\text{tst}} q_1, \{\text{false}\} \\
q_0 & \xrightarrow{k=0} q_0, \{0 = t - s\} \xrightarrow{\text{inc}} q_0, \{1 = t - s\} \xrightarrow{\text{dec}} q_0, \{0 = t - s\} \xrightarrow{\text{tst}} q_1, \{\text{false}\}
\end{align*}
\]
Challenges

How do we formalize counting arguments?

How do we synthesize counting arguments automatically?
“Learning” a counting argument

Program $P$
Spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{\text{pre}}/\varphi_{\text{post}}$?

Yes

Add $\tau$ to $Tr$.

No

Let $\tau$ be a cex

No

Does the proof accept all traces?

Yes

Construct a counting proof for $Tr$.  

No

Z. Kincaid (U. Toronto)  Proofs That Count  January 22, 2014
“Learning” a counting argument

Program $P$
Spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{\text{pre}}/\varphi_{\text{post}}$?

No
Let $\tau$ be a cex

Yes

Does the proof accept all traces?

No

Add $\tau$ to $Tr.$

Yes

Construct a counting proof for $Tr.$

Yes

$✓$

No

$✗$
“Learning” a counting argument

Program $P$
Spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{\text{pre}}/\varphi_{\text{post}}$?

No

Let $\tau$ be a cex

Yes

Add $\tau$ to $Tr.$

Construct a counting proof for $Tr.$

Does the proof accept all traces?

✓

No
“Learning” a counting argument

Program $P$
Spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{\text{pre}}/\varphi_{\text{post}}$?

No
Let $\tau$ be a cex

Yes
Add $\tau$ to $Tr.$

No

Yes

Does the proof accept all traces?

Construct a counting proof for $Tr.$
“Learning” a counting argument

Program $P$
Spec $\varphi_{\text{pre}} / \varphi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{\text{pre}} / \varphi_{\text{post}}$?

No
- Let $\tau$ be a cex

Yes

Add $\tau$ to $Tr.$

No

Yes

Does the proof accept all traces?

Construct a counting proof for $Tr.$
"Learning" a counting argument

Program $P$

Spec $\varphi_{pre}/\varphi_{post}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{pre}/\varphi_{post}$?

No

Let $\tau$ be a cex

Yes

Add $\tau$ to $Tr.$

Does the proof accept all traces?

Yes

Construct a counting proof for $Tr.$

No

Does $\tau$ satisfy $\varphi_{pre}/\varphi_{post}$?
Constructing a counting proof

Goal

Given a finite set of traces $Tr$ and a spec $\phi_{pre}/\phi_{post}$, construct a counting proof $\langle A, \phi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation simultaneously.

- Insight #1: Bounded synthesis is decidable
  - Bound the size of the counting proof (think: # of states)
  - Encode bounded proof synthesis as a formula in a decidable theory (QF_UFNRA)
  - Use uninterpreted function symbols to encode the transition relation.
  - Use Farkas’ lemma to generate constraints searching for an inductive annotation (á la Colón et al.\textsuperscript{a})

\textsuperscript{a}Linear Invariant Generation using Non-linear Constraint Solving, CAV’03
Constructing a counting proof

**Goal**

Given a finite set of traces $Tr$ and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- Insight #2: Occam’s Razor – search for a “small” proof. More likely to generalize & use counters!

$$\tau = t++; s++; t++; [t \leq s]$$
Constructing a counting proof

Goal

Given a finite set of traces $Tr$ and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation \textit{simultaneously}.

- Insight #2: Occam’s Razor – search for a “small” proof. More likely to generalize & use counters!

\[
\tau = t++; s++; t++; [t \leq s]
\]

\[
\begin{align*}
q_0 & \xrightarrow{t++/\text{nop}} q_1 \\
q_1 & \xrightarrow{[t \leq s]/\text{nop}} q_2
\end{align*}
\]

\[
\begin{align*}
\{0 = t - s\} & \xrightarrow{s++/\text{nop}} q_0 \\
\{1 = t - s\} & \xrightarrow{\{\text{false}\}} q_2
\end{align*}
\]
Constructing a counting proof

**Goal**

Given a finite set of traces $Tr$ and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- **Insight #2**: Occam’s Razor – search for a “small” proof. More likely to generalize & use counters!

$$\tau = t++; s++; t++; [t \leq s]$$
“Learning” a counting argument

Program $P$
Spec $\varphi_{pre}/\varphi_{post}$

Choose a trace $\tau$

Does $\tau$ satisfy $\varphi_{pre}/\varphi_{post}$?

- No
  - Let $\tau$ be a cex
  - No
    - Yes
      - Does the proof accept all traces?
        - Yes
          - Construct a counting proof for $Tr.$
            - Yes
        - No
          - Add $\tau$ to $Tr.$
            - Yes
              - Construct a counting proof for $Tr.$

- Yes

$Z. Kincaid$ (U. Toronto)

Proofs That Count

January 22, 2014 11 / 15
Control flow nets

Control flow net = Petri net + program commands
Control flow nets

Control flow net = Petri net + program commands

1: t++ 1: t++ 1: t++
2: \texttt{assert}(t > s) 2: \texttt{assert}(t > s) \cdots 2: \texttt{assert}(t > s)
3: s++ 3: s++ 3: s++

Represents the set of error traces for the program.
Proof checking

Theorem

Let $P$ be a control flow net, and let $A$ be a counting automaton. The problem of determining whether $L(P) \subseteq L(A)$ is decidable.
Proof checking

Theorem

Let $P$ be a control flow net, and let $A$ be a counting automaton. The problem of determining whether $\mathcal{L}(P) \subseteq \mathcal{L}(A)$ is decidable.

• Reduction to Petri net language inclusion.
We can automate synthesis of a class of auxiliary variables!

**Program P**

**Spec** $\phi_{\text{pre}} / \phi_{\text{post}}$

Choose a trace $\tau$

Does $\tau$ satisfy $\phi_{\text{pre}} / \phi_{\text{post}}$?

- **No**
  - Let $\tau$ be a cex
  - **No**
- **Yes**
  - **Yes**
    - Does the proof accept all traces?
      - **Yes**
      - **✓**
      - **Bounded synthesis**
      - **Reduce to Petri net language inclusion**
      - **Search for small proof**
      - **Add $\tau$ to $Tr.$**
  - **No**

**Z. Kincaid (U. Toronto)**

Proofs That Count

January 22, 2014
What’s next?

- Implementation & Evaluation
  - Practical algorithm for inclusion?
    - Ultimately, inclusion relies on a reduction to Petri net reachability.
  - Practical nonlinear constraint solving?
- Synthesize other classes of auxiliary variables?