Commutativity in Concurrent Program Verification

Dominik Klumpp

klumpp@informatik.uni-freiburg.de

University of Freiburg

joint work with:	Azadeh Farzan	(University of Toronto)
	Andreas Podelski	(University of Freiburg)
	Marcel Ebbinghaus	(University of Freiburg)

AVM 2022

Example Program

$$\{ \; x = y = i = j = 0 \; \}$$

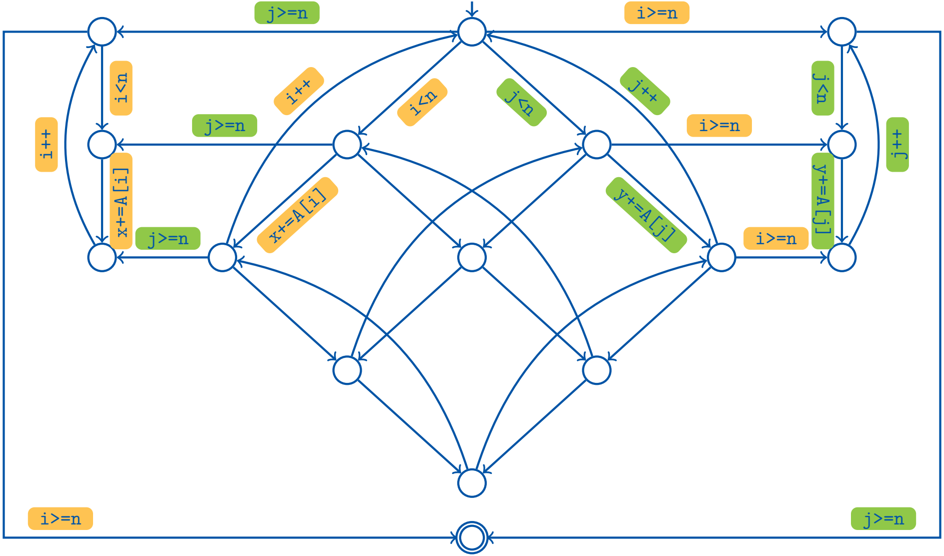
```
while (i < n) {
    x += A[i];
    i++;
}
```

$$\|$$

```
while (j < n) {
    y += A[j];
    j++;
}
```

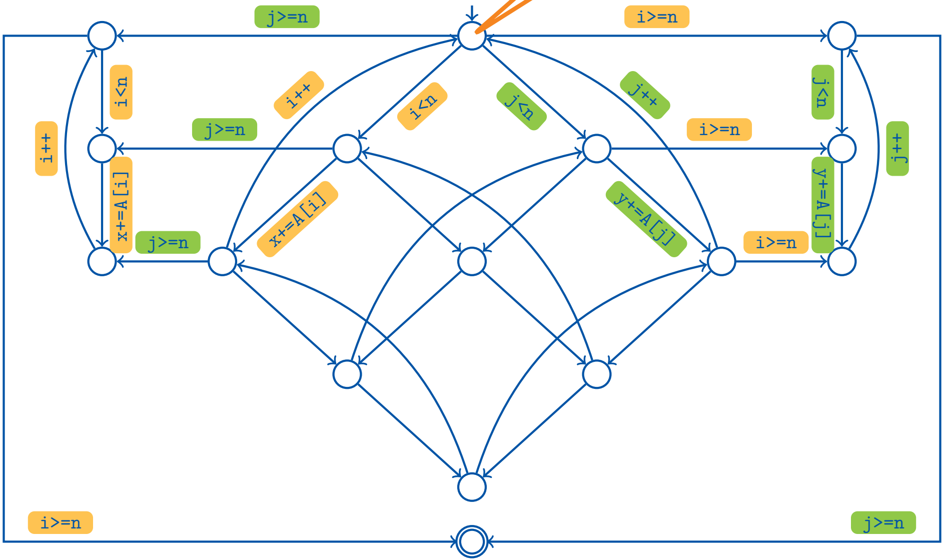
$$\{ \; x = y \; \}$$

Naïve Sequentialization



Naïve Sequentialization

$$x = \sum_{k=0}^{i} A[k] \wedge y = \sum_{k=0}^{j} A[k] \wedge i \leq n \wedge j \leq n$$



Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

$\Rightarrow$ equivalence between program interleavings

Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

$\Rightarrow$ equivalence between program interleavings

Extension: **proof-sensitive** commutativity

Example: `B[k]:=c` commutes with `B[l]:=d` **if proof guarantees** $k \neq l \vee c = d$

Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

$\Rightarrow$ equivalence between program interleavings

Extension: **proof-sensitive** commutativity

Example: `B[k]:=c` commutes with `B[l]:=d` **if proof guarantees** $k \neq l \vee c = d$
Typical Cases: aliasing, conditional updates (CAS), blocking statements (locks)

Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

$\Rightarrow$ equivalence between program interleavings

Extension: **proof-sensitive** commutativity

Example: `B[k]:=c` commutes with `B[l]:=d` **if proof guarantees** $k \neq l \vee c = d$
Typical Cases: aliasing, conditional updates (CAS), blocking statements (locks)

Key Property: Correct traces only equivalent to correct traces.

Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

Example: `x+=A[i]` `y+=A[j]` $\sim$ `y+=A[j]` `x+=A[i]`

$\Rightarrow$ equivalence between program interleavings

Extension: **proof-sensitive** commutativity

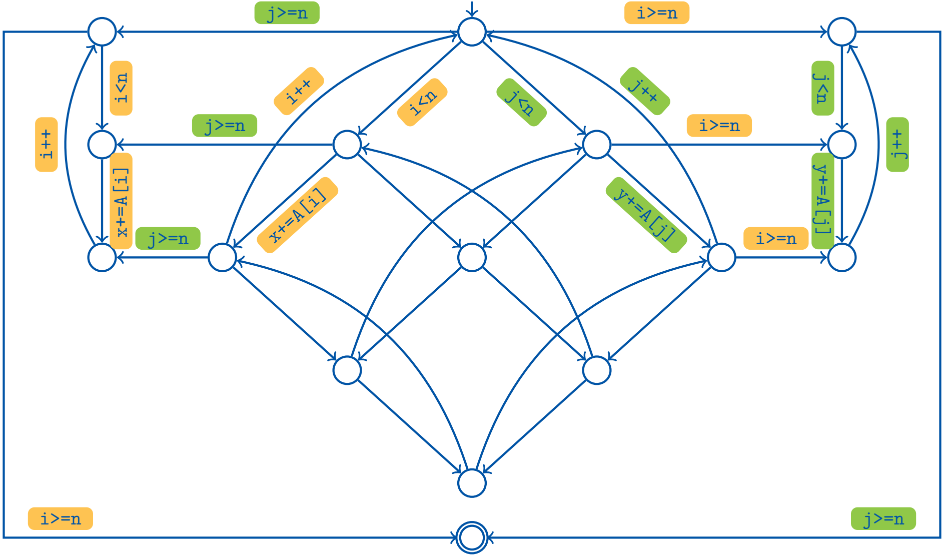
Example: `B[k]:=c` commutes with `B[l]:=d` **if proof guarantees** $k \neq l \vee c = d$
Typical Cases: aliasing, conditional updates (CAS), blocking statements (locks)

Key Property: Correct traces only equivalent to correct traces.

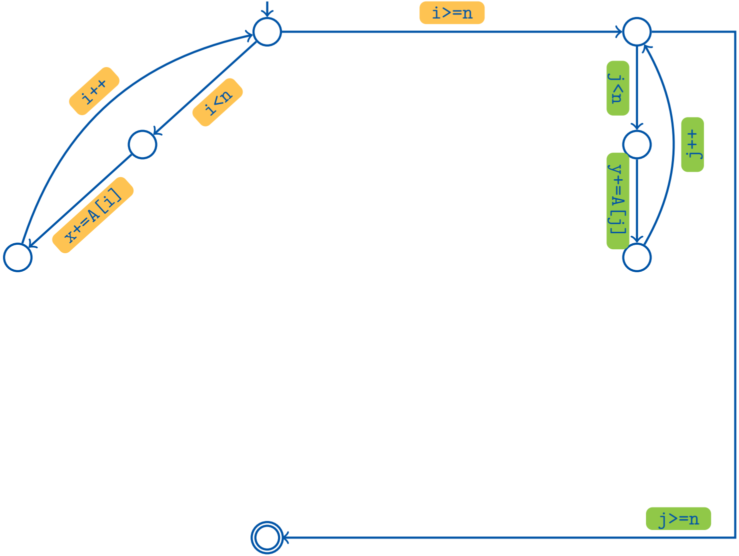
Reduction

One representative trace for each equivalence class

Naïve Sequentialization



Reduction I



Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

complex proofs to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

complex proofs to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / . . . assertions
- **quantitatively:** need many distinct proof assertions

$\rightsquigarrow$ reduction may have **simpler proof**

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

complex proofs to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

$\rightsquigarrow$ reduction may have **simpler proof**

exponential proof checking to show that proof covers all interleavings

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

complex proofs to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / . . . assertions
- **quantitatively:** need many distinct proof assertions

$\rightsquigarrow$ reduction may have **simpler proof**

exponential proof checking to show that proof covers all interleavings

$\rightsquigarrow$ **compactly represent** reductions

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

complex proofs to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

$\rightsquigarrow$ reduction may have **simpler proof**

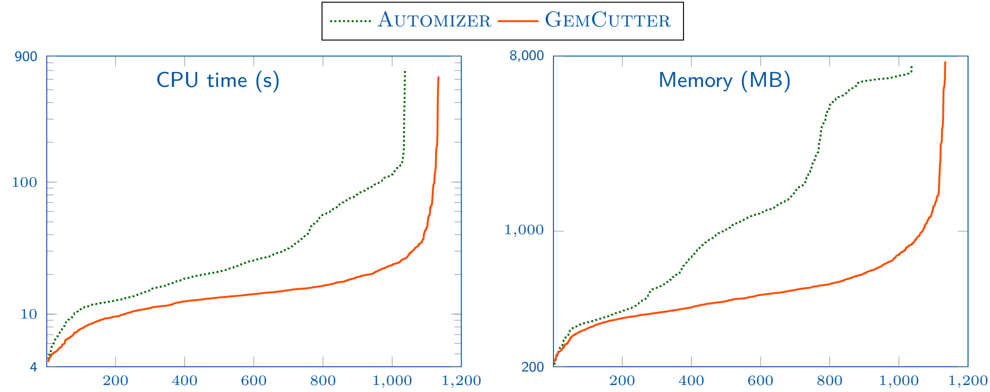
exponential proof checking to show that proof covers all interleavings
$\rightsquigarrow$ **compactly represent** reductions

[1] Ebbinghaus. *Tight Integration of Partial Order Reduction into Trace Abstraction Refinement*. BSc Thesis
[2] Klumpp, Dietsch, Heizmann, Schüssele, Ebbinghaus, Farzan and Podelski. *Ultimate GemCutter and the Axes of Generalization - (Competition Contribution)*. TACAS 2022

Evaluation

Implemented in ULTIMATE GEMCUTTER software model checker

Evaluated on SV-COMP'21 benchmarks and benchmarks of WEAVER tool



analyzed **50** more programs using significantly less time ($\approx 50\,\%$), memory ($\approx 27\,\%$), and refinement rounds ($\approx 64\,\%$)

Reductions

Reduction: One representative trace for each equivalence class

Reduction: One representative trace for each equivalence class

$$red^{I}_{\preceq}(P)$$

Reductions

Reduction: One representative trace for each equivalence class

program to be verified

$$red^{I}_{\preceq}(P)$$

Reductions

Reduction: One representative trace for each equivalence class

commutativity relation I
defines equivalence classes

program to be verified

$$red^{I}_{\preceq}(P)$$

Reductions

Reduction: One representative trace for each equivalence class

commutativity relation I

defines equivalence classes

program to be verified

$$red^{I}_{\preceq}(P)$$

preference order $\preceq$

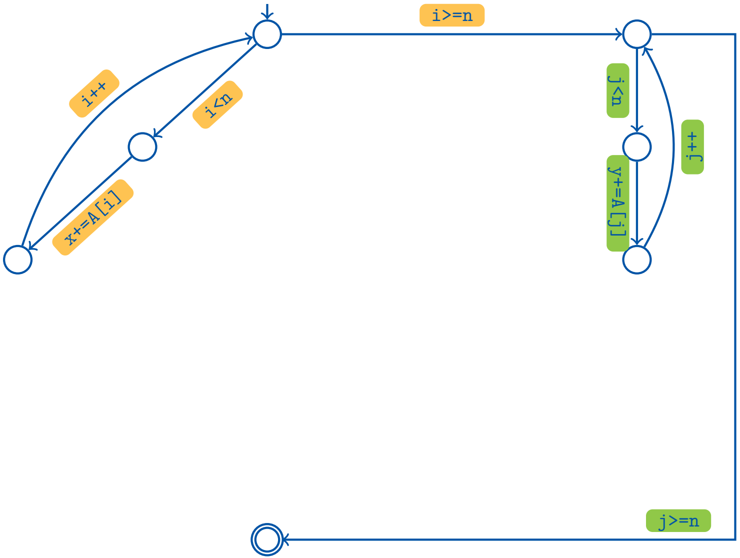
selects representatives for each equivalence class

Preference Orders

Selecting the right representatives

[3] Farzan, Klumpp and Podelski. *Sound sequentialization for concurrent program verification.* PLDI 2022

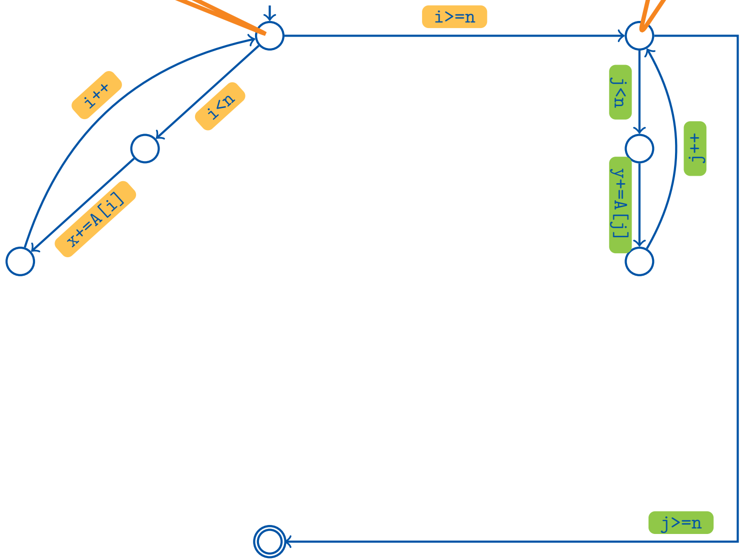
Reduction I



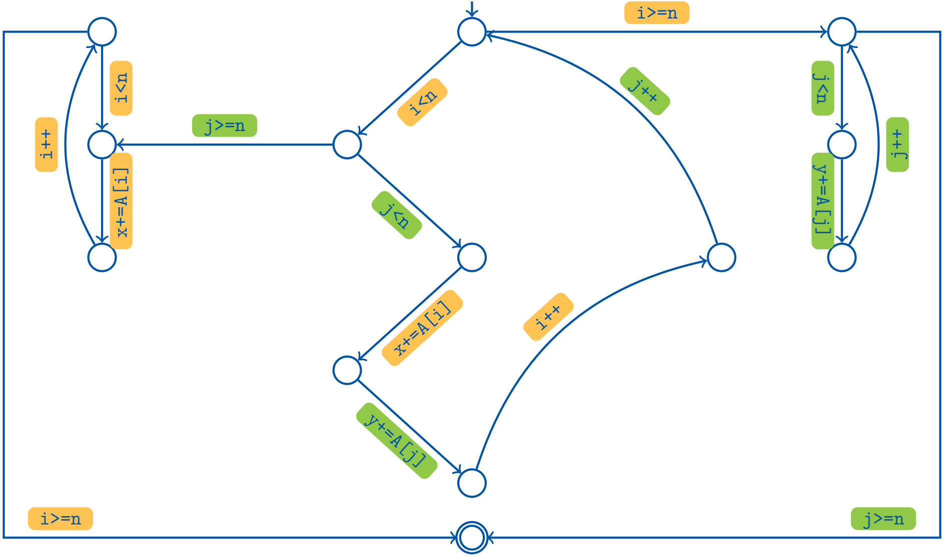
Reduction I

$x = \sum_{k=0}^{i} A[k] \wedge i \leq n \wedge y = 0 \wedge j = 0$

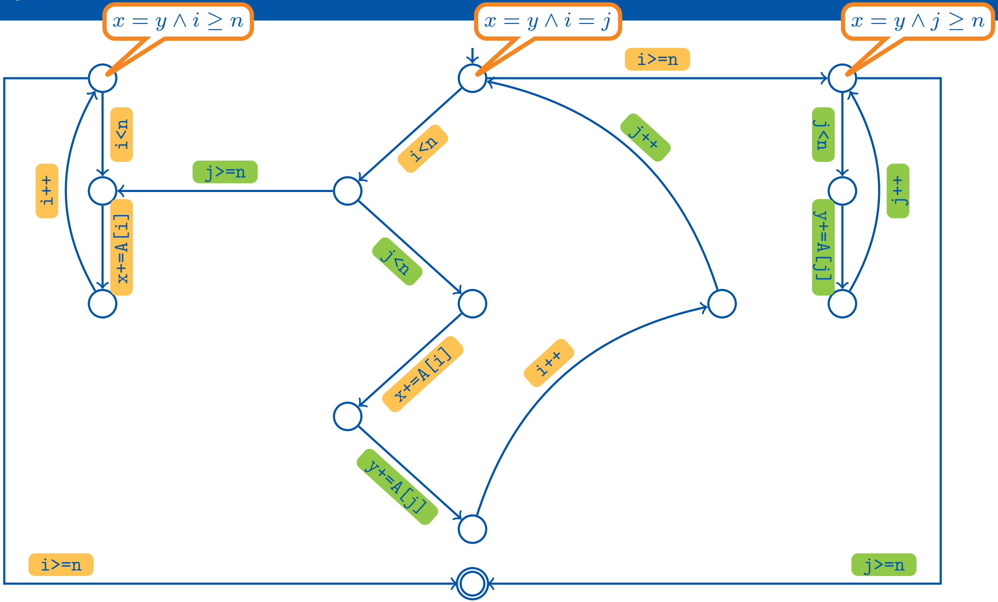
$x = \sum_{k=0}^{n} A[k] \wedge y = \sum_{k=0}^{j} A[k] \wedge j \leq n$



Reduction II



Reduction II



Characterizing Reductions

Preference orders characterize choice of reduction

Characterizing Reductions

Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)

Characterizing Reductions

Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only **most preferred representative** per equivalence class

$$red^I_{\preceq}(P) := \{\, \min_{\preceq}[\tau]_{\sim_I} \mid \tau \in P \,\}$$

Characterizing Reductions

Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only **most preferred representative** per equivalence class

$$red^{I}_{\preceq}(P) := \{ \min_{\preceq} [\tau]_{\sim_I} \mid \tau \in P \}$$

- independent of commutativity

Characterizing Reductions

Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only **most preferred representative** per equivalence class

$$red^{I}_{\preceq}(P) := \{\, \min_{\preceq}[\tau]_{\sim_I} \mid \tau \in P \,\}$$

- independent of commutativity
- same scheme of preference order applies to different programs

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings:** proofs not unnecessarily complex

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings:** proofs not unnecessarily complex
- **compact** representation

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings:** proofs not unnecessarily complex
- **compact** representation
  - through **weakly persistent membranes**

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings:** proofs not unnecessarily complex
- **compact** representation
  - through **weakly persistent membranes**
- **linear-size** representation in the best case

Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings:** proofs not unnecessarily complex
- **compact** representation
  - through **weakly persistent membranes**
- **linear-size** representation in the best case

$\rightarrow$ More on preference orders in **Marcel's talk**

Commutativity Relations

at different abstraction levels

[work in progress; presented at Commute workshop @ PLDI'22]

Commutativity

Statements st_1 and st_2 **commute**

iff

Commutativity

Statements st_1 and st_2 **commute**

iff

the order of execution does not matter

Commutativity

Statements st_1 and st_2 **commute**

iff

the order of execution does not matter

($st_1 st_2$ behaves exactly like $st_2 st_1$)

Formally: $[\![st_1 st_2]\!] = [\![st_2 st_1]\!]$

Commutativity

Statements st_1 and st_2 **commute**

iff

the order of execution does not matter
($\mathit{st}_1\mathit{st}_2$ behaves exactly like $\mathit{st}_2\mathit{st}_1$)
for all programs and wrt. all properties

Formally: $[\![\mathit{st}_1\mathit{st}_2]\!] = [\![\mathit{st}_2\mathit{st}_1]\!]$

Commutativity

Statements st_1 and st_2 **commute**

iff

the order of execution does not matter

($st_1 st_2$ behaves similar enough to $st_2 st_1$)

for a given program and property

Commutativity

Statements st_1 and st_2 **commute**

iff

abstract irrelevant details

the order of execution does not matter

($st_1 st_2$ behaves similar enough to $st_2 st_1$)

for a given program and property

Commutativity

Statements st_1 and st_2 **commute**

iff

abstract irrelevant details

preserve relevant details

the order of execution does not matter

($st_1 st_2$ *behaves similar enough to* $st_2 st_1$)

for a given program and property

Commutativity

Statements st_1 and st_2 **commute**

iff

abstract irrelevant details

preserve relevant details

the order of execution does not matter

($st_1 st_2$ *behaves similar enough to* $st_2 st_1$)

for a given (partial) proof

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

$$\frac{red^{I}_{\preceq}(P) \subseteq \mathcal{L}(\Pi)}{P \text{ is correct}}$$

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

traces proven correct by Π

$$\frac{red^{I}_{\preceq}(P) \subseteq \mathcal{L}(\Pi)}{P \text{ is correct}}$$

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

traces proven correct by Π

$$\frac{red^{I}_{\preceq}(P) \subseteq \mathcal{L}(\Pi) \quad I \text{ safe wrt. } \Pi}{P \text{ is correct}}$$

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

traces in $\mathcal{L}(\Pi)$ only equivalent to correct traces

traces proven correct by Π

$$\frac{red^I_{\preceq}(P) \subseteq \mathcal{L}(\Pi) \qquad I \text{ safe wrt. } \Pi}{P \text{ is correct}}$$

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

traces in $\mathcal{L}(\Pi)$ only equivalent to correct traces

traces proven correct by Π

$$\frac{red^{I}_{\preceq}(P) \subseteq \mathcal{L}(\Pi) \qquad I \text{ safe wrt. } \Pi}{P \text{ is correct}}$$

- commutativity I_C based on (concrete) semantics: safe wrt. all proofs Π

Safe Commutativity

Let Π be a **proof** (a set of Hoare triples).

traces in $\mathcal{L}(\Pi)$ only equivalent to correct traces

traces proven correct by Π

$$\frac{red^{I}_{\preceq}(P) \subseteq \mathcal{L}(\Pi) \qquad I \text{ safe wrt. } \Pi}{P \text{ is correct}}$$

- commutativity I_C based on (concrete) semantics: safe wrt. all proofs Π
- How to get safe commutativity for a particular proof Π?

Safe Abstraction

Let $\alpha : \mathit{Stmt} \to \mathit{Stmt}$.

Safe Abstraction

Let $\alpha : Stmt \rightarrow Stmt$.

$$I_\alpha := \{\, (st_1, st_2) \mid [\![\alpha(st_1)\alpha(st_2)]\!] = [\![\alpha(st_2)\alpha(st_1)]\!] \,\}$$

Safe Abstraction

Let $\alpha : \mathit{Stmt} \to \mathit{Stmt}$.

$$I_\alpha := \{\, (st_1, st_2) \mid [\![\alpha(st_1)\alpha(st_2)]\!] = [\![\alpha(st_2)\alpha(st_1)]\!] \,\}$$

Theorem (Safety)

If α satisfies

- **abstraction:** $[\![st]\!] \subseteq [\![\alpha(st)]\!]$ for all st
- **preservation:** $\{\varphi\}\alpha(st)\{\psi\}$ is valid, for all $\{\varphi\}st\{\psi\} \in \Pi$

then I_α is safe wrt. Π.

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Abstraction:

- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Abstraction:

- reads of irrelevant variables $\leadsto$ nondeterministic values
- assignment to irrelevant variables $\leadsto$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{\ \{\top\}$ `y:=x+x` $\{y \neq 1\}\ \}$. Then

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Abstraction:

- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{\ \{\top\}\ \texttt{y:=x+x}\ \{y \neq 1\}\ \}$. Then

$$\alpha_\Pi(\texttt{y:=x+x}) \quad : \quad \text{“assign y to some even value (nondet.)”}$$

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Abstraction:

- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{\ \{\top\}$ `y:=x+x` $\{y \neq 1\}\ \}$. Then

$\alpha_\Pi($`y:=x+x`$)$: "assign y to some even value (nondet.)"

$\alpha_\Pi($`x:=0`$)$: "do not change y"

Instance: Projection to the Proof

Idea: Variable x does not occur in the proof $\Rightarrow$ Ignore x when determining commutativity

Abstraction:

- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{\ \{\top\}$ `y:=x+x` $\{y \neq 1\}\ \}$. Then

$\alpha_\Pi($ `y:=x+x` $)$: "assign y to some even value (nondet.)"

$\alpha_\Pi($ `x:=0` $)$: "do not change y"

Now: $\alpha_\Pi($ `y:=x+x` $)$ commutes with $\alpha_\Pi($ `x:=0` $)$.

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

Limitations:

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity

Limitations:

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity
- abstraction easy to compute

Limitations:

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity
- abstraction easy to compute

Limitations:

- **theoretically:** may lose commutativity

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity
- abstraction easy to compute

Limitations:

- **theoretically:** may lose commutativity
- **practically:** introduces quantifiers

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity
- abstraction easy to compute

Limitations:

- **theoretically:** may lose commutativity
- **practically:** introduces quantifiers

Generally: abstract commutativity $\not\supseteq$ concrete commutativity

Instance: Projection to the Proof

Proposition: Projection to the proof is safe (it satisfies **abstraction** and **preservation**).

Advantages:

- often allows additional commutativity
- abstraction easy to compute

Limitations:

- **theoretically:** may lose commutativity
- **practically:** introduces quantifiers

Generally: abstract commutativity $\not\supseteq$ concrete commutativity
Solution: combine abstract with concrete commutativity

Combining Commutativity Relations

Observation: **Union** of safe commutativity relations may be unsafe!

Combining Commutativity Relations

Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof Π:** $\{\top\}$ `x:=1` $\{\top\}$ `x==1;z:=1` $\{\top\}$ `x==2;z:=2` $\{z = 2\}$

Combining Commutativity Relations

Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof Π:** $\{\top\}$ `x:=1` $\{\top\}$ `x==1;z:=1` $\{\top\}$ `x==2;z:=2` $\{z = 2\}$

`x:=1` `x==1;z:=1` `x==2;z:=2`

Combining Commutativity Relations

Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof Π:** $\{\top\}$ `x:=1` $\{\top\}$ `x==1;z:=1` $\{\top\}$ `x==2;z:=2` $\{z = 2\}$

`x:=1` `x==1;z:=1` `x==2;z:=2` $\sim_{I_C}$ `x:=1` `x==2;z:=2` `x==1;z:=1`

Combining Commutativity Relations

Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof** Π**:** $\{\top\}$ `x:=1` $\{\top\}$ `x==1;z:=1` $\{\top\}$ `x==2;z:=2` $\{z = 2\}$

`x:=1` `x==1;z:=1` `x==2;z:=2` $\sim_{I_C}$ `x:=1` `x==2;z:=2` `x==1;z:=1`

$\sim_{I_\alpha}$ `x==2;z:=2` `x:=1` `x==1;z:=1`

Combining Commutativity Relations

Idea: Sequentially combine commutativity relations

Combining Commutativity Relations

Idea: **Sequentially combine** commutativity relations

$$\underset{\substack{\uparrow \\ \textbf{proven}}}{\tau_1} \quad \overset{\textbf{(1) abstract}}{\sim_{I_\alpha}} \quad \underset{\Rightarrow}{\tau_2} \quad \overset{\textbf{(2) concrete}}{\sim_{I_C}} \quad \underset{\substack{\uparrow \\ \textbf{correct}}}{\tau_3}$$

Combining Commutativity Relations

Idea: **Sequentially combine** commutativity relations

$$\begin{array}{ccccc} & \textbf{(1) abstract} & & \textbf{(2) concrete} & \\ \tau_1 & \sim_{I_\alpha} & \tau_2 & \sim_{I_C} & \tau_3 \\ \uparrow & & & & \uparrow \\ \textbf{proven} & & \Rightarrow & & \textbf{correct} \end{array}$$

Combination through **new proof rule**:

$$\frac{red_{\preceq}^{I_\alpha, I_C}(P) \subseteq \mathcal{L}(\Pi) \qquad I_\alpha \text{ safe wrt. } \Pi}{P \text{ is correct}}$$

Combining Commutativity Relations

Idea: **Sequentially combine** commutativity relations

$$\begin{array}{ccccc} & \text{(1) abstract} & & \text{(2) concrete} & \\ \tau_1 & \sim_{I_\alpha} & \tau_2 & \sim_{I_C} & \tau_3 \\ \uparrow & & & & \uparrow \\ \text{proven} & & \Rightarrow & & \text{correct} \end{array}$$

Combination through **new proof rule**:

“more abstract than”

$$\frac{red_{\preceq}^{I_1,\ldots,I_n}(P) \subseteq \mathcal{L}(\Pi) \qquad I_1,\ldots,I_n \text{ safe wrt. } \Pi \qquad I_1 \supseteq \ldots \supseteq I_n}{P \text{ is correct}}$$

Combining Commutativity Relations

Idea: **Sequentially combine** commutativity relations

$$\begin{array}{ccccc} & \text{(1) abstract} & & \text{(2) concrete} & \\ \tau_1 & \sim_{I_\alpha} & \tau_2 & \sim_{I_C} & \tau_3 \\ \uparrow & & & & \uparrow \\ \text{proven} & & \Rightarrow & & \text{correct} \end{array}$$

Combination through **new proof rule**:

"more abstract than"

$$\frac{red_{\preceq}^{I_1,\ldots,I_n}(P) \subseteq \mathcal{L}(\Pi) \qquad I_1,\ldots,I_n \text{ safe wrt. } \Pi \qquad I_1 \supseteq \ldots \supseteq I_n}{P \text{ is correct}}$$

New **partial order reduction algorithm** for n commutativity relations

Conclusion

Summary

In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.

Summary

In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.

Preference Orders: Selection of representatives in reduction

- influences both proof simplicity and proof check efficiency
- trade-off between both aspects

Summary

In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.

Preference Orders: Selection of representatives in reduction

- influences both proof simplicity and proof check efficiency
- trade-off between both aspects

Commutativity Relations: Determines notion of equivalence

- automatically **computed** and **safe wrt. a proof**
- e.g. derived from **safe abstractions**
- new **proof rule** and **algorithm** combine commutativity relations

Summary

In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.

Preference Orders: Selection of representatives in reduction

- influences both proof simplicity and proof check efficiency
- trade-off between both aspects

Commutativity Relations: Determines notion of equivalence

- automatically **computed** and **safe wrt. a proof**
- e.g. derived from **safe abstractions**
- new **proof rule** and **algorithm** combine commutativity relations

Questions?