Detection and integration of conditional commutativity for concurrent program verification

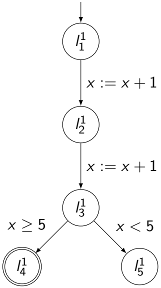
Marcel Ebbinghaus

University of Freiburg

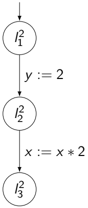
6^{th} Sep. 2024

AVM 2024

Concurrent programs



(a) Program automaton A_P^1 for thread1.



(b) Program automaton A_P^2 for thread2.

Figure: Two program automata modeling the threads of a concurrent program.

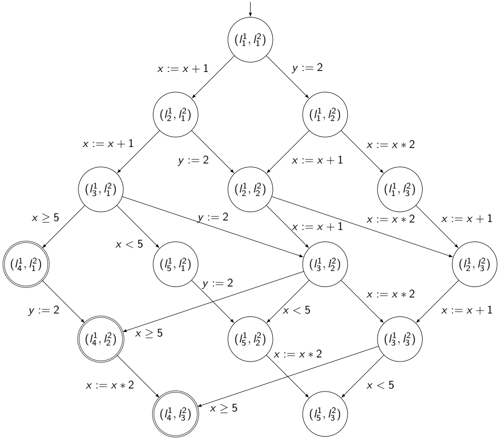


Figure: The concurrent program automaton A_P for the concurrent program consisting of A_P^1, A_P^2.

How do we prove concurrent programs?

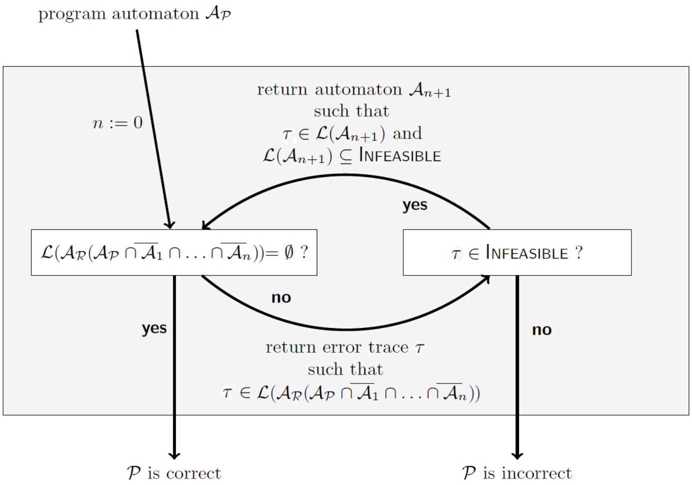


Figure: The CEGAR-Loop of Trace Abstraction Refinement (TAR) with integrated Partial Order Reduction (POR).

What is a reduction?

Problem

- The generalizations of TAR may not be able to cover all error traces

What is a reduction?

Problem

- The generalizations of TAR may not be able to cover all error traces

Commutativity

- Two statements commute if their order doesn't affect the semantics
- Example: $[\![(x := x + 1)(y := 2)]\!] \equiv [\![(y := 2)(x := x + 1)]\!]$

What is a reduction?

Problem

- The generalizations of TAR may not be able to cover all error traces

Commutativity

- Two statements commute if their order doesn't affect the semantics
- Example: $[\![(x := x + 1)(y := 2)]\!] \equiv [\![(y := 2)(x := x + 1)]\!]$

Reduction

- Traces that only differ in the order of commuting statements can be seen as equivalent
- A reduction is a subset of traces which contains at least one trace per equivalence class
- Proving the reduction is sufficient to prove the program

What is conditional commutativity?

- Conditional commutativity allows us to further refine the reduction

What is conditional commutativity?

- Conditional commutativity allows us to further refine the reduction

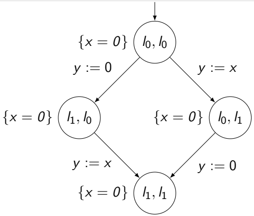


Figure: A program automaton with conditional commutativity.

What is conditional commutativity?

- Conditional commutativity allows us to further refine the reduction

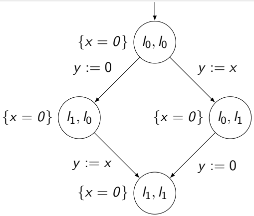


Figure: A program automaton with conditional commutativity.

- $[\![(y := 0)(y := x)]\!] \neq [\![(y := x)(y := 0)]\!]$,
  i.e. $(y := 0)$ and $(y := x)$ do not commute in general

What is conditional commutativity?

- Conditional commutativity allows us to further refine the reduction

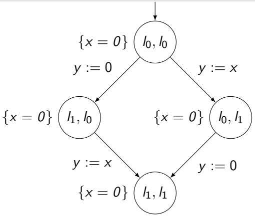


Figure: A program automaton with conditional commutativity.

- $[\![(y := 0)(y := x)]\!] \not\equiv [\![(y := x)(y := 0)]\!]$,
  i.e. $(y := 0)$ and $(y := x)$ do not commute in general
- $[\![(y := 0)(y := x)]\!]_{\{x=0\}} \equiv [\![(y := x)(y := 0)]\!]_{\{x=0\}}$,
  i.e. $(y := 0)$ and $(y := x)$ commute under condition $x = 0$

Why do we want more conditional commutativity?

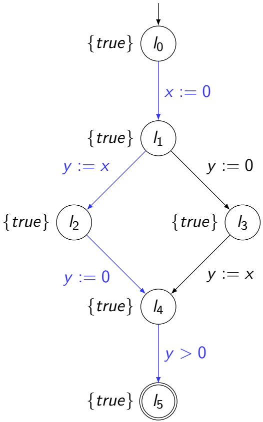


Figure: A simplified reduction automaton $A_R(A_P)$.

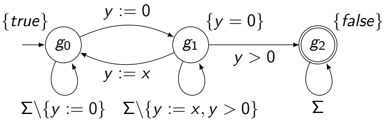


Figure: Generalization G_1 of trace $(x := 0)(y := x)(y := 0)(y > 0)$.

Why do we want more conditional commutativity?

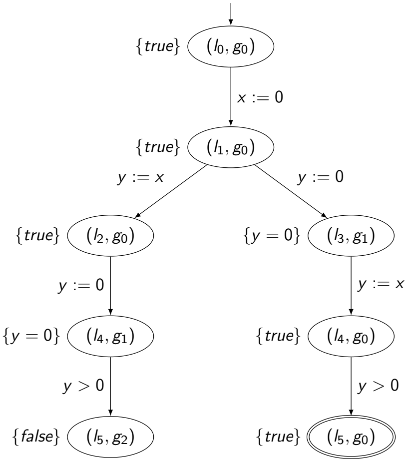


Figure: Reduction automaton $A_R(A_P \cap \overline{G_1})$.

- The generalization did not provide a sufficient commutativity condition
- Thus, we need another iteration of the refinement loop

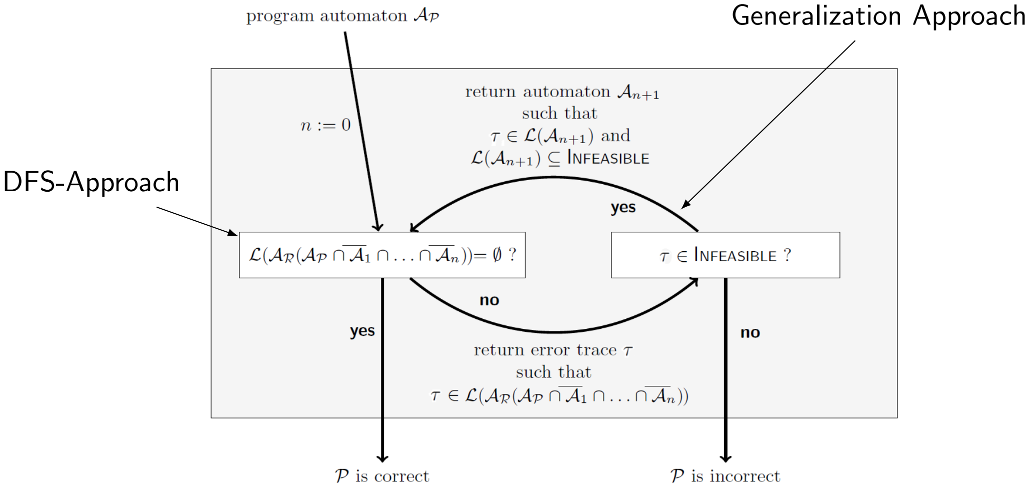


Figure: A modified CEGAR-Loop showing our two approaches.

Generalization Approach

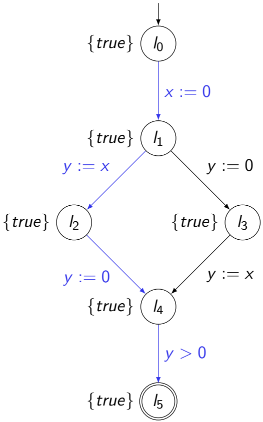


Figure: A simplified reduction automaton $A_R(A_P)$.

- 1. Traverse along the infeasible trace until two non-commuting statements occur or until its end
- Thus, until $(l_1, \emptyset)$ with non-commuting $y := x$ and $y := 0$

Generalization Approach

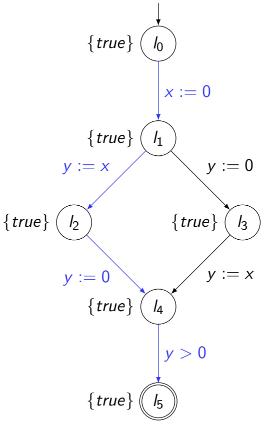


Figure: A simplified reduction automaton $A_R(A_P)$.

- 2. Decide if we want to check for conditional commutativity
- We use different criteria for this

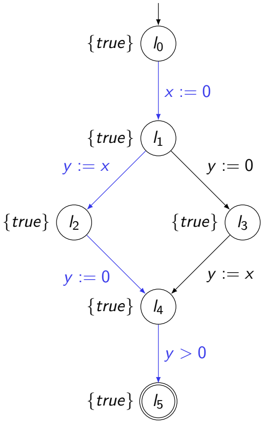


Figure: A simplified reduction automaton $A_R(A_P)$.

- 3. Try to calculate a commutativity condition
- For instance $x = 0$, since $(y := 0)$ and $(y := x)$ commute under condition $x = 0$

Generalization Approach

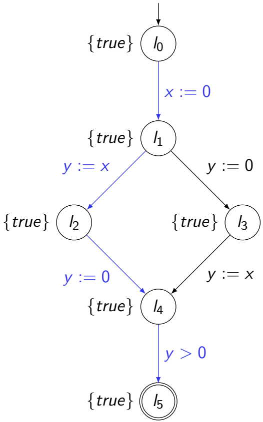


Figure: A simplified reduction automaton $A_R(A_P)$.

- 4. Try to prove that this condition holds after the current trace and store the proof
- For instance $\{true\}\{x = 0\}$ proves that condition $x = 0$ holds after trace $x := 0$, since $\{true\}x := 0\{x = 0\}$ is a valid Hoare-triple

Generalization Approach

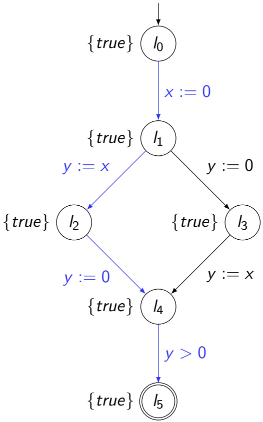


Figure: A simplified reduction automaton $A_R(A_P)$.

- 5. Continue with 1
- 1. Traverse along the infeasible trace until two non-commuting statements occur or until its end
- 6. Construct a generalization G' with integrated proofs

Generalization Approach

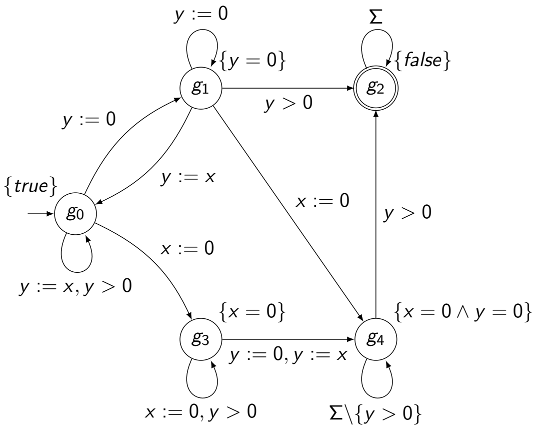


Figure: Generalization G_1' of trace $(x := 0)(y := x)(y := 0)(y > 0)$ with integrated proof $\{true\}\{x = 0\}$ for condition $x = 0$.

Generalization Approach

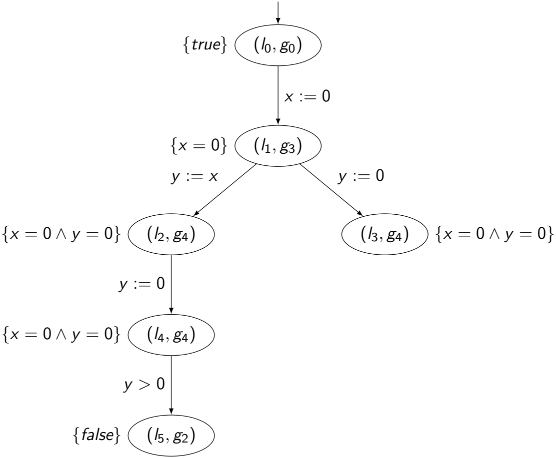


Figure: Reduction automaton $A_R(A_P \cap \overline{G_1'})$.

- The integration of conditional commutativity allows us to prune the remaining error traces
- Thus, we don't need another iteration of the refinement loop

DFS-Approach

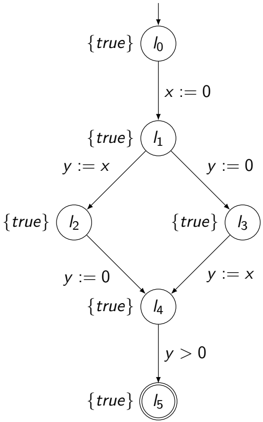


Figure: A simplified reduction automaton $A_R(A_P)$.

- 1. DFS until two non-commuting statements occur
- Thus, until $(l_1, \emptyset)$ with non-commuting $y := x$ and $y := 0$

DFS-Approach

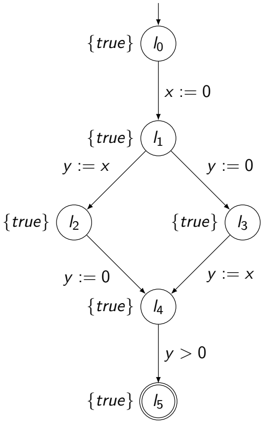


Figure: A simplified reduction automaton $A_R(A_P)$.

- 2. Decide if we want to check for conditional commutativity
- We use different criteria for this

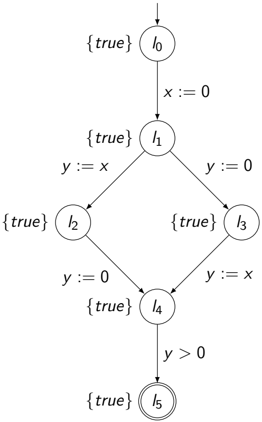


Figure: A simplified reduction automaton $A_R(A_P)$.

- 3. Try to calculate a commutativity condition
- For instance $x = 0$, since $(y := 0)$ and $(y := x)$ commute under condition $x = 0$

DFS-Approach

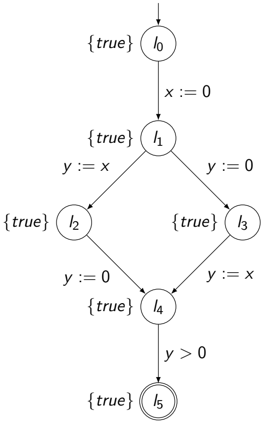


Figure: A simplified reduction automaton $A_R(A_P)$.

- 4. Try to prove that this condition holds after the current trace and construct a Floyd-Hoare automaton
- For instance proof $\{true\}\{x = 0\}$

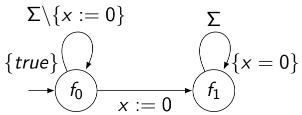


Figure: Floyd-Hoare automaton $A_{x=0}$ of $\{true, x = 0\}$.

- 5. Add this automaton to the trace abstraction and restart the DFS

DFS-Approach

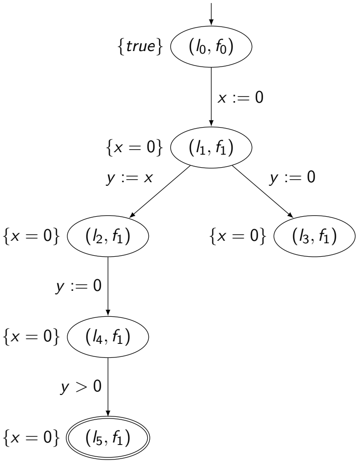


Figure: Reduction automaton $A_R(A_P \cap \overline{A_{x=0}})$.

- The integration of conditional commutativity allows us to prune one of the error traces
- Thus, we only need to consider the remaining error trace

Correctness and Termination

- We proved correctness of both approaches
- We showed that the DFS-approach is non-terminating in general
- We were able to guarantee and prove termination by using so called perfect proofs

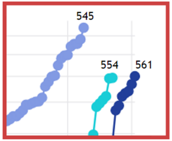
Evaluation

- We implemented both approaches into Ultimate GemCutter
- We used a total of 875 programs as benchmarks

Summary of observations

- The generalization approach proved more programs in total than GemCutter, while the DFS-approach proved less
- Both approaches were able to prove programs that the regular GemCutter didn't prove
- Both come with an overhead in time and memory consumption
- We think that the overhead is a reasonable one for the generalization approach

Regular GemCutter: ●
Generalization-Approach: ●
DFS-Approach: ●



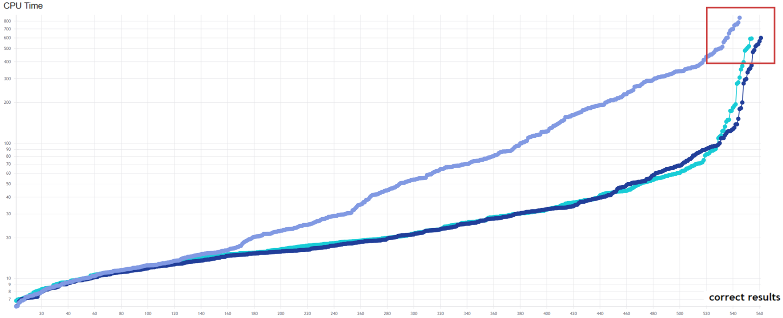


Figure: Logarithmic CPU-Time quantile-diagram.