The Concept of Preference Orders for Concurrent Program Verification

Marcel Ebbinghaus

Albert-Ludwigs-Universität Freiburg

14.09.2022

AVM 2022

Motivation

- The chosen order may have a huge impact on simplicity of proofs and overall efficiency for concurrent program verification

Motivation

- The chosen order may have a huge impact on simplicity of proofs and overall efficiency for concurrent program verification
- We want to have a practical definition of orders

Motivation

- The chosen order may have a huge impact on simplicity of proofs and overall efficiency for concurrent program verification
- We want to have a practical definition of orders
- Preference Orders as an automata based representation

Preference Orders

Preference Order:

A (total) Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a total order over the program statements

Preference Orders

Preference Order:

A (total) Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a total order over the program statements

- The monitor describes the switching of (thread) priorities

Preference Orders

Preference Order:

A (total) Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a total order over the program statements

- The monitor describes the switching of (thread) priorities
- The function assigns each product state an individual total order

Preference Orders

Preference Order:

A (total) Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a total order over the program statements

- The monitor describes the switching of (thread) priorities
- The function assigns each product state an individual total order
- We denote the set of all Preference Orders as *PrefOrd*

Preference Orders

Partial Preference Order:

A Partial Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a partial order over the program statements

Preference Orders

Partial Preference Order:

A Partial Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a partial order over the program statements

- We denote the set of all Partial Preference Orders as *PartPrefOrd*

Preference Orders

Partial Preference Order:

A Partial Preference Order for a given program automaton A^P is a tuple (A^M, f), where

- A^M is a total DFA (monitor)
- f maps each state of the product $A^P \times A^M$ to a partial order over the program statements

- We denote the set of all Partial Preference Orders as *PartPrefOrd*
- Since $TotOrd \subseteq PartOrd$, we have $PrefOrd \subseteq PartPrefOrd$

Preference Orders

- Where is the connection to the representative of an equivalence class?

Preference Orders

- Where is the connection to the representative of an equivalence class?
- Partial Preference Orders can be seen as orders over the program traces (words of the program automaton's language)

Preference Orders

- Where is the connection to the representative of an equivalence class?
- Partial Preference Orders can be seen as orders over the program traces (words of the program automaton's language)

Word Order:

Let $w_1 = a_1 a_2 ... a_n, w_2 = b_1 b_2 ... b_n$, then $(w_1 < w_2) \in wordOrder((A^M, f))$ iff for the first index i where the symbols a_i, b_i differ, we have $(a_i < b_i)$ at the corresponding product state (q_i^P, q_i^M)

Preference Orders

- Where is the connection to the representative of an equivalence class?
- Partial Preference Orders can be seen as orders over the program traces (words of the program automaton's language)

Word Order:

Let $w_1 = a_1 a_2 ... a_n, w_2 = b_1 b_2 ... b_n$, then $(w_1 < w_2) \in wordOrder((A^M, f))$ iff for the first index i where the symbols a_i, b_i differ, we have $(a_i < b_i)$ at the corresponding product state (q_i^P, q_i^M)

- The reduction contains the minimal word w.r.t the word order

Combination of Partial Preference Orders

- We can define lots of Preference Orders, but have to hardcode them

Combination of Partial Preference Orders

- We can define lots of Preference Orders, but have to hardcode them
- We want to be more flexible in using Preference Orders

Combination of Partial Preference Orders

- We can define lots of Preference Orders, but have to hardcode them
- We want to be more flexible in using Preference Orders
- Partial Preference Orders allow uncertainty between threads

Combination of Partial Preference Orders

- We can define lots of Preference Orders, but have to hardcode them
- We want to be more flexible in using Preference Orders
- Partial Preference Orders allow uncertainty between threads
- Idea: Can we use this to combine multiple of them?

$$\begin{array}{c} O_1 \in \mathit{PartPrefOrd} \\ + \\ O_2 \in \mathit{PartPrefOrd} \end{array} \longrightarrow ?$$

Combination of Partial Preference Orders

- We can define lots of Preference Orders, but have to hardcode them
- We want to be more flexible in using Preference Orders
- Partial Preference Orders allow uncertainty between threads
- Idea: Can we use this to combine multiple of them?
- Important: We want to receive a Partial Preference Order again!

$$\begin{array}{c} O_1 \in \mathit{PartPrefOrd} \\ + \\ O_2 \in \mathit{PartPrefOrd} \end{array} \longrightarrow (O_1 \rhd O_2) \in \mathit{PartPrefOrd}$$

Combination of Partial Preference Orders

- Our first approach was inspired by lexicographical orders

Combination of Partial Preference Orders

- Our first approach was inspired by lexicographical orders
- Intuitive idea: If the first order is uncertain, use the second

Combination of Partial Preference Orders

- Our first approach was inspired by lexicographical orders
- Intuitive idea: If the first order is uncertain, use the second
- Problem: Transitivity is not preserved!

$$\begin{array}{c} O_1 := a < c \\ + \\ O_2 := c < b < a \end{array} \longrightarrow (O_1 \rhd O_2) \notin \mathit{PartPrefOrd}$$

Combination of Partial Preference Orders

- Our first approach was inspired by lexicographical orders
- Intuitive idea: If the first order is uncertain, use the second
- Problem: Transitivity is not preserved!

$$\begin{array}{c} O_1 := a < c \\ + \\ O_2 := c < b < a \end{array} \longrightarrow (O_1 \rhd O_2) \notin \mathit{PartPrefOrd}$$

- Workaround to enforce transitivity probably possible, but expensive

Parameterized Preference Orders

- A different approach to gain more flexibility

Parameterized Preference Orders

- A different approach to gain more flexibility
- Idea: We define a function that constructs useful Preference Orders

$$f(\ldots) \longrightarrow O \in ParamPrefOrd \subseteq PartPrefOrd$$

Parameterized Preference Orders

- A different approach to gain more flexibility
- Idea: We define a function that constructs useful Preference Orders
- The function determines the structure according to two parameters

$$f(X, Y) \longrightarrow O \in ParamPrefOrd \subseteq PartPrefOrd$$

Parameterized Preference Orders

- The first parameter defines the thread priority switching

Parameterized Preference Orders

- The first parameter defines the thread priority switching
- We want to admit the following behaviors:
  - A thread may be the most priorized for multiple successive steps
  - A thread's next most priorized thread may change

Parameterized Preference Orders

- The first parameter defines the thread priority switching
- We want to admit the following behaviors:
  - A thread may be the most priorized for multiple successive steps
  - A thread's next most priorized thread may change
- We denote this parameter as a sequence $(thread, maxSteps)$...
- Example: $(t_1, 1)(t_2, 2)(t_1, 1)(t_3, 1)$

Parameterized Preference Orders

- The first parameter defines the thread priority switching
- We want to admit the following behaviors:
  - A thread may be the most priorized for multiple successive steps
  - A thread's next most priorized thread may change
- We denote this parameter as a sequence $(thread, maxSteps)...$
- Example: $(t_1, 1)(t_2, 2)(t_1, 1)(t_3, 1)$

- The second parameter defines which statements are considered a step
- Example: all write operations

Parameterized Preference Orders

- Suppose we are given the parameters as in the examples:
  - $(t_1, 1)(t_2, 2)(t_1, 1)(t_3, 1)$
  - all write operations
- Then the reduction contains the following word as representative of it's equivalence class since it is minimal:

`x:=x+y` `y:=y-x` `y<x` `y:=0` `x:=x+y` `y:=2y`

Conclusion and Outlook

- Preference Orders as a practical automata based representation for concurrent program verification

Conclusion and Outlook

- Preference Orders as a practical automata based representation for concurrent program verification
- Parameterized Preference Orders allow us to be more flexible

Conclusion and Outlook

- Preference Orders as a practical automata based representation for concurrent program verification
- Parameterized Preference Orders allow us to be more flexible
- Problem: We have to guess “good” parameters by hand!

Conclusion and Outlook

- Preference Orders as a practical automata based representation for concurrent program verification
- Parameterized Preference Orders allow us to be more flexible
- Problem: We have to guess “good” parameters by hand!
- Our current goal: Obtain an offline heuristic that uses the concurrent program’s structure to guess parameters

The End

Thank you for listening!

Any questions?