Stateless Model Checking Concurrent/Distributed Programs

Michalis Kokologiannakis   Viktor Vafeiadis

POPL 2025

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

$[\mathit{data} = \mathit{flag} = 0]$	
$\mathit{data} := 42$ $\mathit{flag} := 1$	$\mathtt{if}\ (\mathit{flag} = 1)$ $\quad \mathtt{assert}(\mathit{data} = 42)$

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

$[data = flag = 0]$	
$data := 42$ $flag := 1$	$\texttt{if}\ (flag = 1)$ $\quad\texttt{assert}(data = 42)$

SC ✓, x86 ✓, ARM ✗, POWER ✗, C++ ✗, Java ✗, …

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

$[data = flag = 0]$	
$data := 42$ $flag := 1$	$\texttt{if}\ (flag = 1)$ $\quad \texttt{assert}(data = 42)$

SC ✓, x86 ✓, ARM ✗, POWER ✗, C++ ✗, Java ✗, ...

Why concurrency?

All software is concurrent (cache, OS, networking, users, cloud)

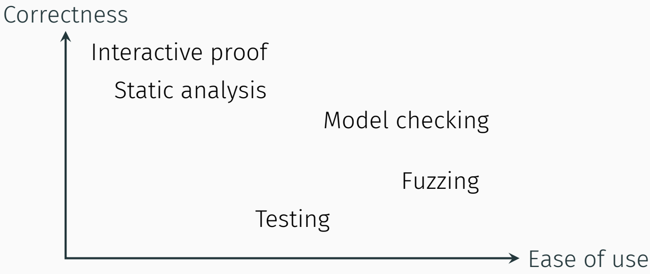
Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

$[data = flag = 0]$

$data := 42$	**if** $(flag = 1)$
$flag := 1$	**assert**$(data = 42)$

SC ✓, x86 ✓, ARM ✗, POWER ✗, C++ ✗, Java ✗, …



Why concurrency?

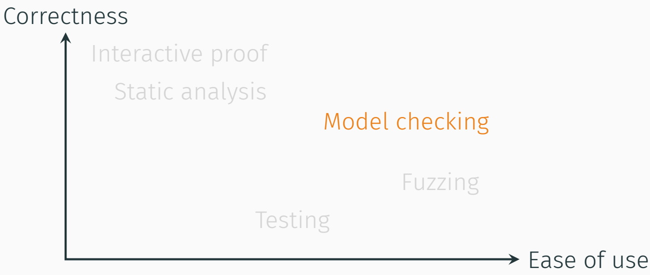
All software is concurrent (cache, OS, networking, users, cloud)

Concurrency **complicates** reasoning

$\rightsquigarrow$ **weak memory consistency** makes matters worse

$[data = flag = 0]$	
$data := 42$ $flag := 1$	**if** $(flag = 1)$ **assert**$(data = 42)$

SC ✓, x86 ✓, ARM ✗, POWER ✗, C++ ✗, Java ✗, …



Model-checking approaches

Stateful:

- Visit program states while recording visited states
- Assumes program has bounded state-space
- High memory usage

Stateless:

- Visit program states without recording visited states
- Assumes program always terminates
- Low memory usage

SMT-based:

- Encode program and specification as an SMT query
- Assumes program always terminates
- High memory usage

Our weapon of choice

[PLDI'19] Model checking weakly-consistent libraries

[POPL'22] Truly stateless, optimal dynamic partial order reduction

GenMC: state-of-the-art stateless model checker

- Correct, optimal, highly-parallelizable
- Works with almost any memory model
- Small memory footprint

plv.mpi-sws.org/genmc

Two papers in POPL'25 (Thu @ 15:00):

- Automatically checking linearizability under weak memory consistency
- Model checking C/C++ with mixed-size accesses

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Stateless model checking

SMC verifies a program by enumerating its interleavings

Stateless model checking

SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$

$$x := 1 \;\big\|\; y := 1 \;\big\|\; a := x$$

Stateless model checking

SMC verifies a program by enumerating its interleavings

$[x = y = 0]$		
$x := 1$	$y := 1$	$a := x$

$$x := 1 \downarrow y := 1 \downarrow a := x$$

$$y := 1 \downarrow x := 1 \downarrow a := x$$

$$x := 1 \downarrow a := x \downarrow y := 1$$

$$a := x \downarrow x := 1 \downarrow y := 1$$

$$a := x \downarrow y := 1 \downarrow x := 1$$

$$y := 1 \downarrow a := x \downarrow x := 1$$

Stateless model checking

SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; a := x$$

a = 1

$x := 1$	$y := 1$	$x := 1$
↓	↓	↓
$y := 1$	$x := 1$	$a := x$
↓	↓	↓
$a := x$	$a := x$	$y := 1$

a = 0

$a := x$	$a := x$	$y := 1$
↓	↓	↓
$x := 1$	$y := 1$	$a := x$
↓	↓	↓
$y := 1$	$x := 1$	$x := 1$

Stateless model checking

SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$
$$x := 1 \;||\; y := 1 \;||\; a := x$$

$a = 1$

$x := 1$	$y := 1$	$x := 1$
↓	↓	↓
$y := 1$	$x := 1$	$a := x$
↓	↓	↓
$a := x$	$a := x$	$y := 1$

$a = 0$

$a := x$	$a := x$	$y := 1$
↓	↓	↓
$x := 1$	$y := 1$	$a := x$
↓	↓	↓
$y := 1$	$x := 1$	$x := 1$

How to enumerate one interleaving per equivalence class?

(Partial order reduction)

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GENMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GENMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Interlude: Semantics under weak memory consistency

Interlude: Declarative semantics under WMC

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ \mathsf{G} \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathsf{ExecGraphs}(P) \;\middle|\; \mathsf{cons}_M(G) \right\}$$

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \text{ExecGraphs}(P) \;\middle|\; \text{cons}_M(G) \right\}$$

$$[\mathit{data} = \mathit{flag} = 0]$$

$\mathit{data} := 42$	$\parallel$	`if` $(\mathit{flag} = 1)$
$\mathit{flag} := 1$		`assert`$(\mathit{data} = 42)$

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

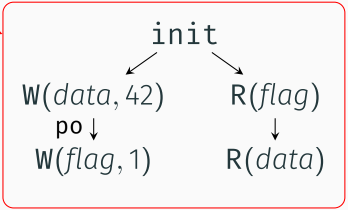
$$[data = flag = 0]$$

$$
\begin{array}{l|l}
data := 42 & \mathtt{if}\ (flag = 1) \\
flag := 1 & \quad \mathtt{assert}(data = 42)
\end{array}
$$

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathsf{ExecGraphs}(P) \;\middle|\; \mathsf{cons}_M(G) \right\}$$

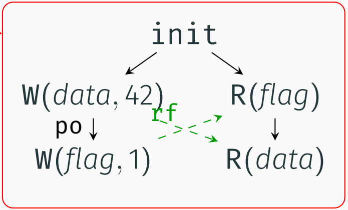
$[data = flag = 0]$	
$data := 42$	$\texttt{if}\ (flag = 1)$
$flag := 1$	$\quad\texttt{assert}(data = 42)$



Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

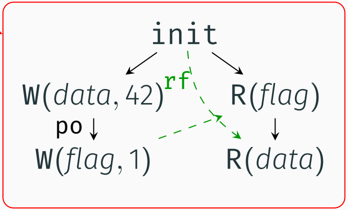
$[data = flag = 0]$	
$data := 42$	$\mathtt{if}\ (flag = 1)$
$flag := 1$	$\quad \mathtt{assert}(data = 42)$



Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \{G \in \text{ExecGraphs}(P) \mid \text{cons}_M(G)\}$$

$[data = flag = 0]$	
$data := 42$ $flag := 1$	`if` $(flag = 1)$ `assert`$(data = 42)$

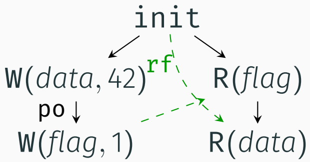


Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

$[data = flag = 0]$

$data := 42$	**if** $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$

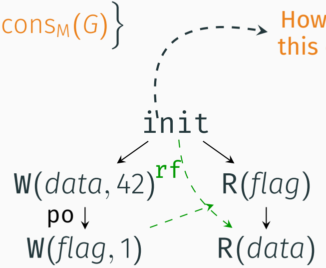


Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathsf{ExecGraphs}(P) \;\middle|\; \mathsf{cons}_M(G) \right\}$$

$[data = flag = 0]$	
$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$

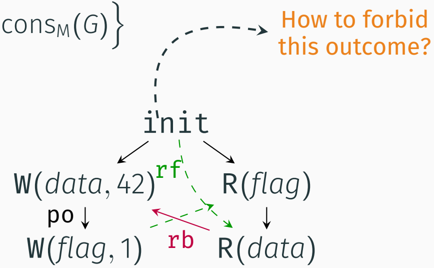
How to forbid this outcome?



Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

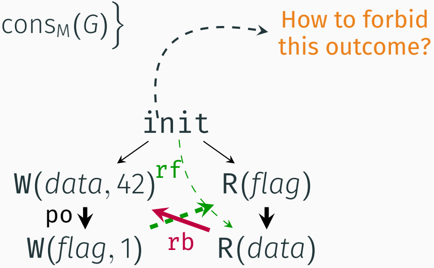
$[data = flag = 0]$	
$data := 42$	**if** $(flag = 1)$
$flag := 1$	$\texttt{assert}(data = 42)$



Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

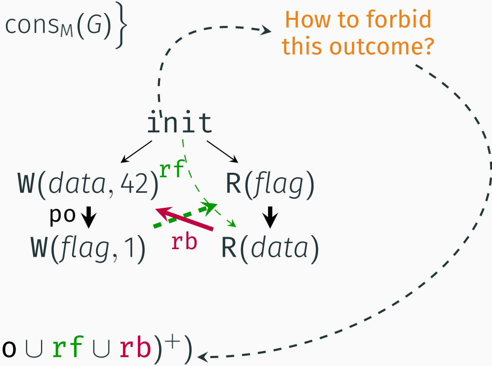
$[data = flag = 0]$	
$data := 42$	$\texttt{if}\ (flag = 1)$
$flag := 1$	$\quad \texttt{assert}(data = 42)$



Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \;\middle|\; \mathrm{cons}_M(G) \right\}$$

$[data = flag = 0]$	
$data := 42$ $flag := 1$	`if` $(flag = 1)$ `assert`$(data = 42)$

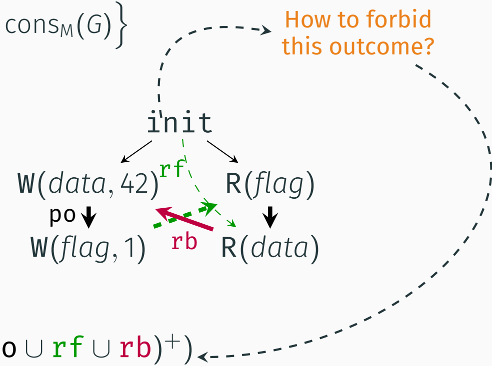


Fictional model M: irreflexive$((\mathtt{po} \cup \mathtt{rf} \cup \mathtt{rb})^+)$

Interlude: Declarative semantics under WMC

$$[\![P]\!]_M \triangleq \left\{ G \in \mathrm{ExecGraphs}(P) \mid \mathrm{cons}_M(G) \right\}$$

$[data = flag = 0]$	
$data := 42$	**if** $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



Fictional model M: $\mathrm{irreflexive}((\mathtt{po} \cup \mathtt{rf} \cup \mathtt{rb})^+)$

Real model SC: $\mathrm{irreflexive}((\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb})^+)$

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

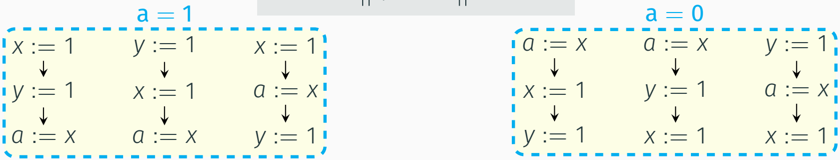
Each part will be followed by a demo

Stateless model checking

SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$

$$x := 1 \;\|\; y := 1 \;\|\; a := x$$



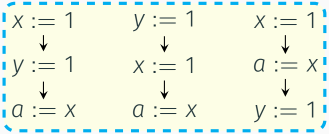
Key Idea: Represent equivalence classes with execution graphs

Stateless model checking

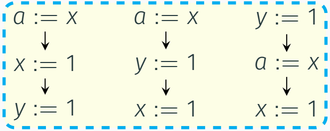
SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$
$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; a := x$$

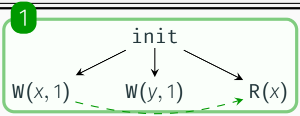
$a = 1$

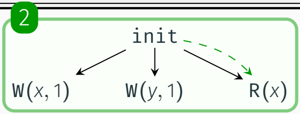


$a = 0$



Key Idea: Represent equivalence classes with execution graphs





Stateless model checking

SMC verifies a program by enumerating its interleavings

$$[x = y = 0]$$
$$x := 1 \parallel y := 1 \parallel a := x$$

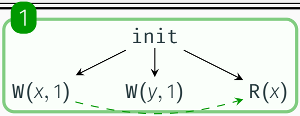
a = 1

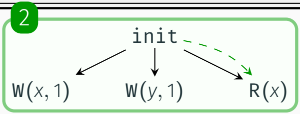
$x := 1$	$y := 1$	$x := 1$
↓	↓	↓
$y := 1$	$x := 1$	$a := x$
↓	↓	↓
$a := x$	$a := x$	$y := 1$

a = 0

$a := x$	$a := x$	$y := 1$
↓	↓	↓
$x := 1$	$y := 1$	$a := x$
↓	↓	↓
$y := 1$	$x := 1$	$x := 1$

Key Idea: Represent equivalence classes with execution graphs





We can verify programs by enumerating consistent execution graphs!

Stateless model checking

SMC verifies a program by enumerating its interleavings

$[x = y = 0]$

$x := 1 \parallel y := 1 \parallel a := x$

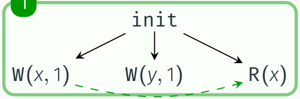
$a = 1$

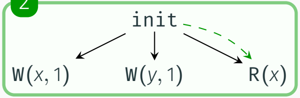
- $x := 1 \to y := 1 \to a := x$
- $y := 1 \to x := 1 \to a := x$
- $x := 1 \to a := x \to y := 1$

$a = 0$

- $a := x \to x := 1 \to y := 1$
- $a := x \to y := 1 \to x := 1$
- $y := 1 \to a := x \to x := 1$

Key Idea: Represent equivalence classes with execution graphs

1


2


does not assume SC

We can verify programs by enumerating consistent execution graphs!

GenMC's algorithm: Example #1

$[x = y = 0]$		
$x := 1$	$y := 1$	$a := x$

GenMC's algorithm: Example #1

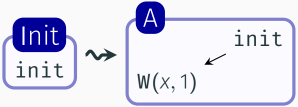
$$[x = y = 0]$$

$$x := 1 \;\|\; y := 1 \;\|\; a := x$$

GenMC's algorithm: Example #1

$$[x = y = 0]$$

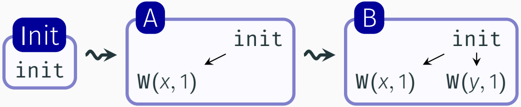
$$x := 1 \;\|\; y := 1 \;\|\; a := x$$



GenMC's algorithm: Example #1

$$[x = y = 0]$$

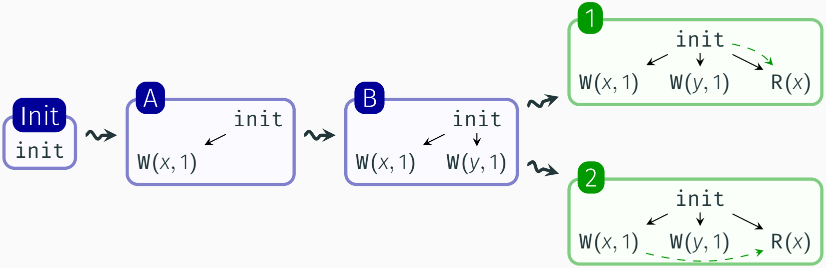
$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; a := x$$



GenMC's algorithm: Example #1

$$[x = y = 0]$$

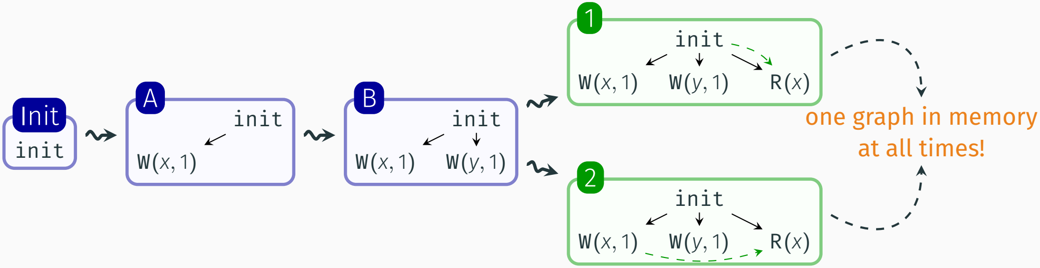
$$x := 1 \;\|\; y := 1 \;\|\; a := x$$



GenMC's algorithm: Example #1

$$[x = y = 0]$$

$$x := 1 \parallel y := 1 \parallel a := x$$



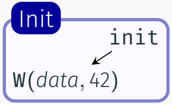
GenMC's algorithm: Example #2

$[data = flag = 0]$	
$data := 42$	**if** $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$

GenMC's algorithm: Example #2

$[data = flag = 0]$

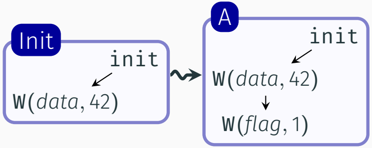
$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



GenMC's algorithm: Example #2

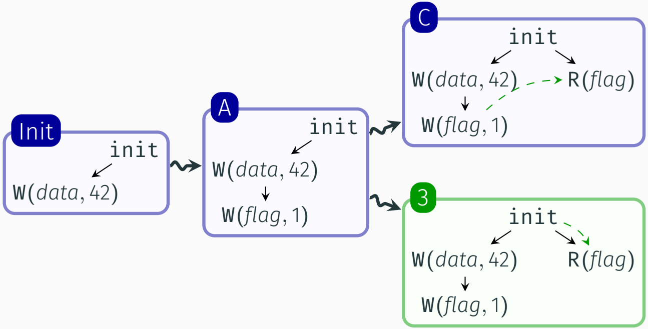
$[data = flag = 0]$

$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



GenMC's algorithm: Example #2

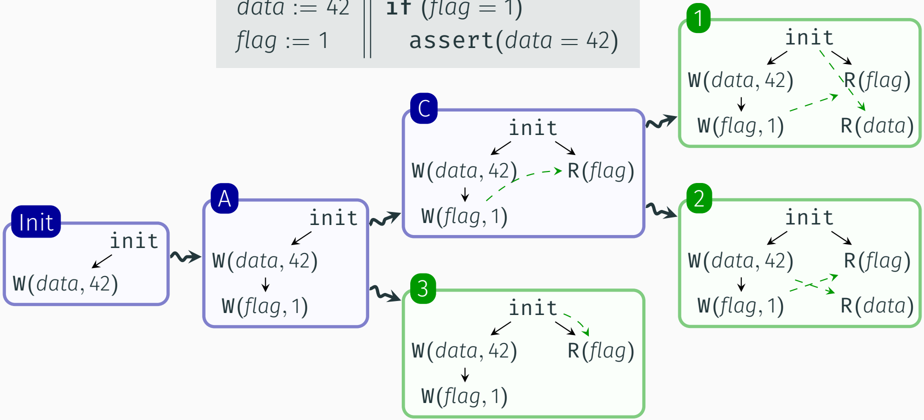
$[data = flag = 0]$	
$data := 42$	**if** $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



GenMC's algorithm: Example #2

$[data = flag = 0]$

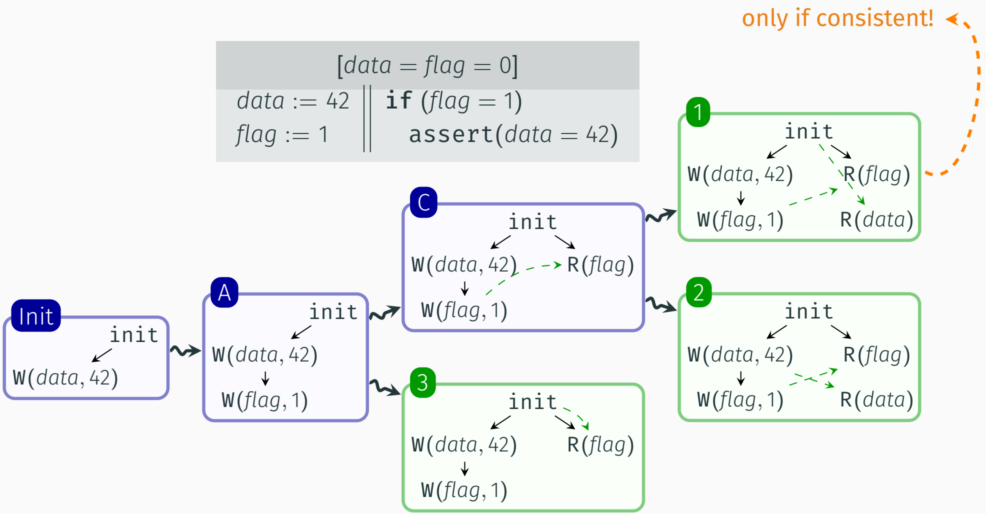
$data := 42$	**if** $(flag = 1)$
$flag := 1$	assert$(data = 42)$



GenMC's algorithm: Example #2

$[data = flag = 0]$

$data := 42$	$\texttt{if}\ (flag = 1)$
$flag := 1$	$\quad \texttt{assert}(data = 42)$



GenMC's algorithm: Example #3

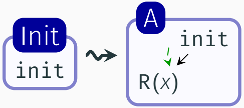
$[x = y = 0]$	
$a := x$ $y := a + 1$	$x := 1$

GenMC's algorithm: Example #3

$[x = y = 0]$	
$a := x$ $y := a + 1$	$x := 1$

GenMC's algorithm: Example #3

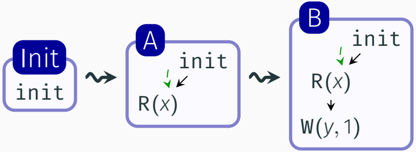
$[x = y = 0]$	
$a := x$ $y := a + 1$	$x := 1$



GenMC's algorithm: Example #3

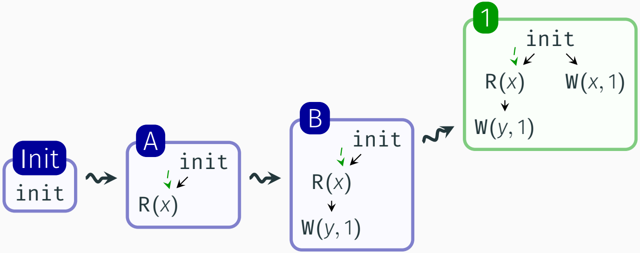
$$[x = y = 0]$$

$$\begin{array}{l} a := x \\ y := a + 1 \end{array} \quad \Big\| \quad x := 1$$



GenMC's algorithm: Example #3

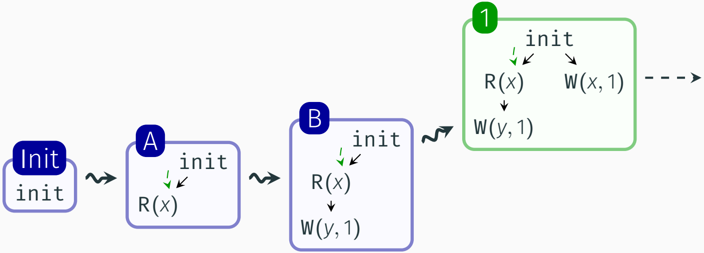
$[x = y = 0]$		
$a := x$ $y := a + 1$	$\|\|$	$x := 1$



GenMC's algorithm: Example #3

$$[x = y = 0]$$

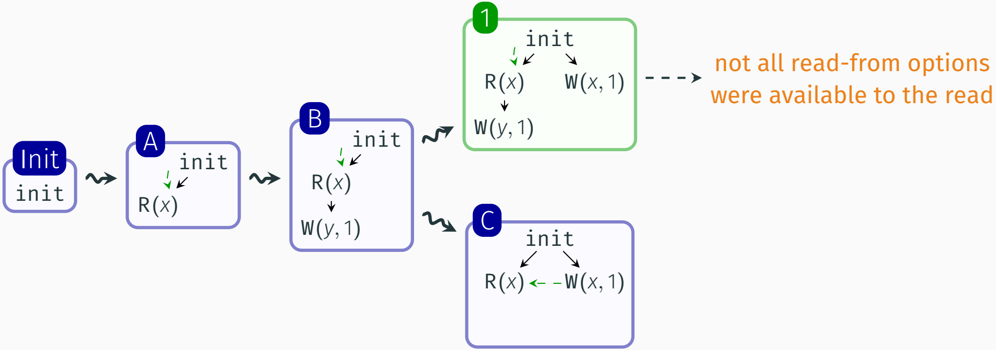
$$\begin{array}{l} a := x \\ y := a + 1 \end{array} \;\Bigg\|\; x := 1$$



not all read-from options were available to the read

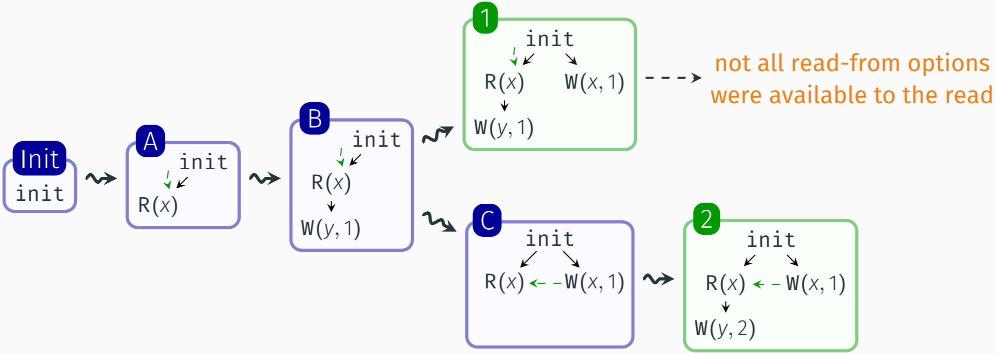
GENMC's algorithm: Example #3

$[x = y = 0]$	
$a := x$ $y := a + 1$	$x := 1$



GENMC's algorithm: Example #3

$[x = y = 0]$	
$a := x$ $y := a + 1$	$x := 1$



GenMC's algorithm: Example #4

$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$

GenMC's algorithm: Example #4

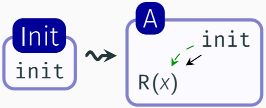
$$[x = y = 0]$$

$$\begin{array}{l} a := x \\ y := a + 1 \end{array} \;\Big\|\; b := y \;\Big\|\; x := 1$$

GenMC's algorithm: Example #4

$$[x = y = 0]$$

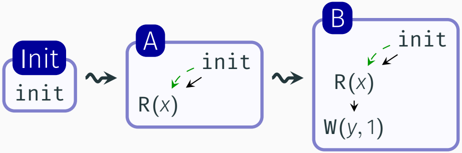
$$\begin{array}{l} a := x \\ y := a + 1 \end{array} \;\Big\|\; b := y \;\Big\|\; x := 1$$



GenMC's algorithm: Example #4

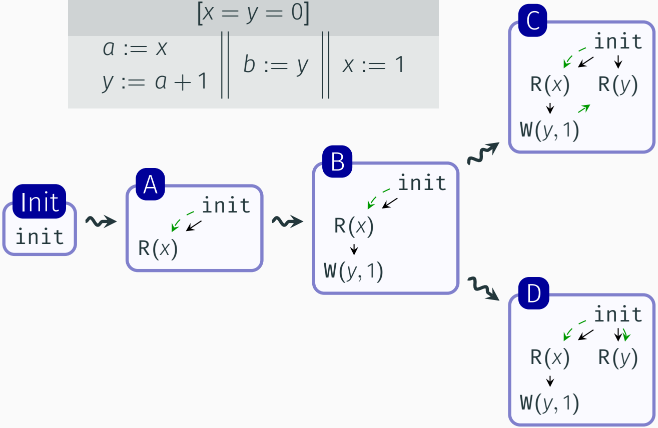
$$[x = y = 0]$$

$a := x$ $y := a + 1$	$b := y$	$x := 1$



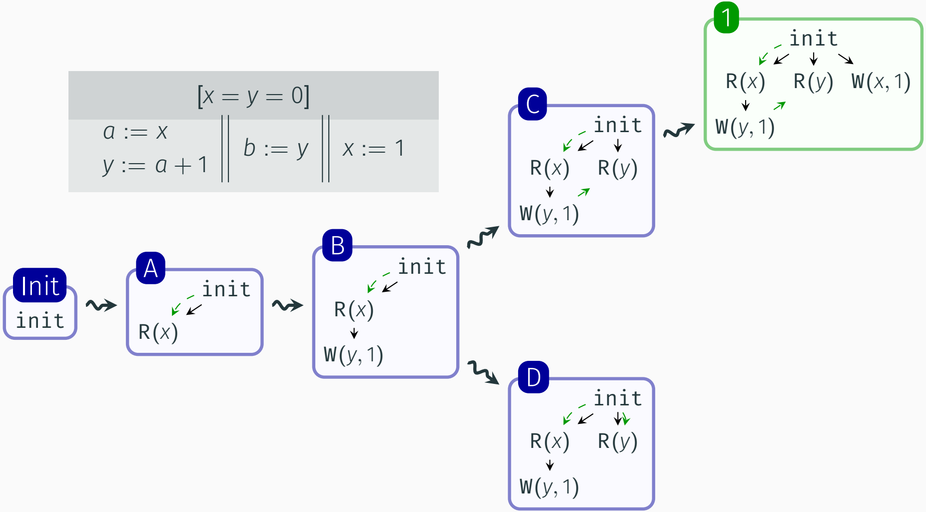
GenMC's algorithm: Example #4

$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$



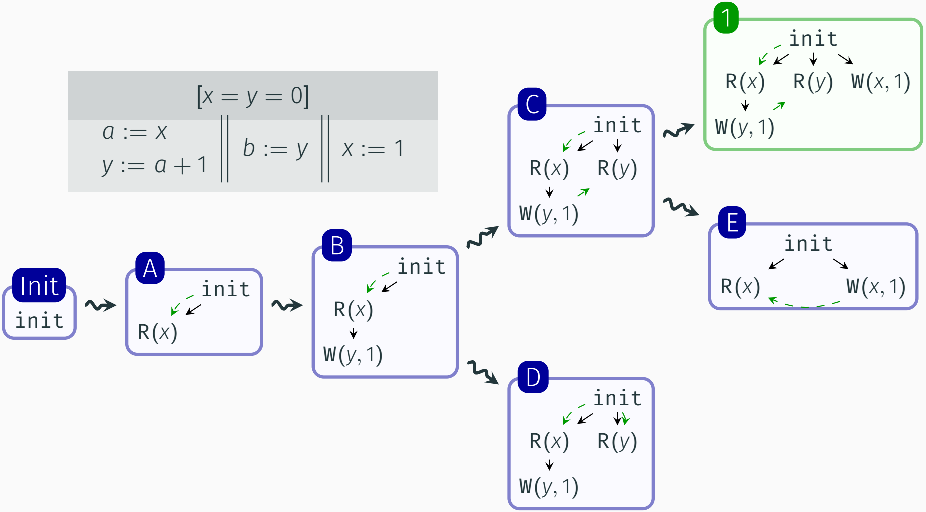
GenMC's algorithm: Example #4

$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$



GenMC's algorithm: Example #4

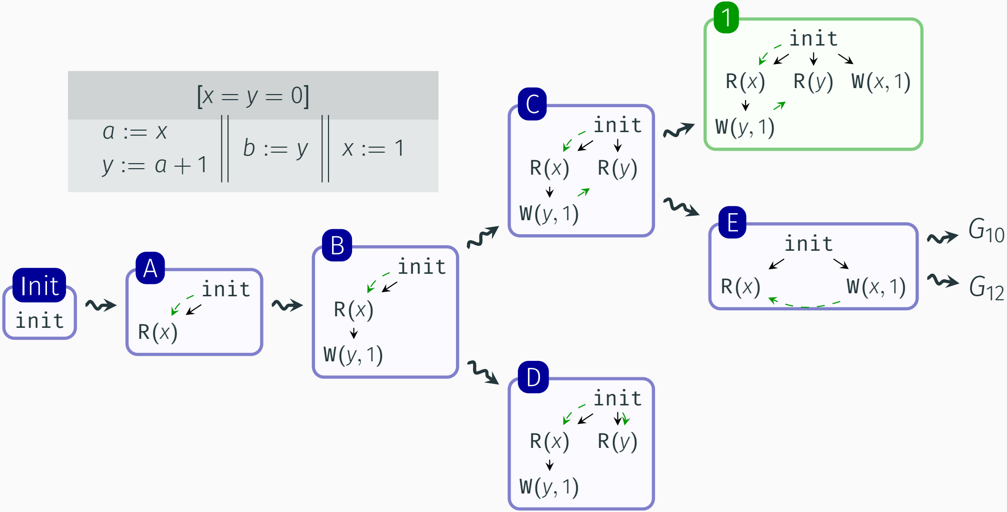
$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$



GENMC's algorithm: Example #4

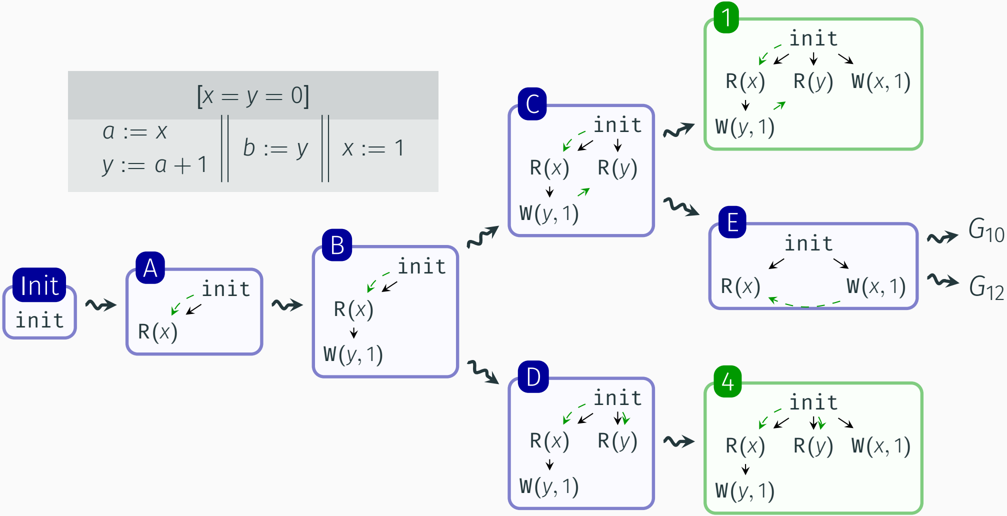
$[x = y = 0]$

$a := x$ $y := a + 1$	$b := y$	$x := 1$



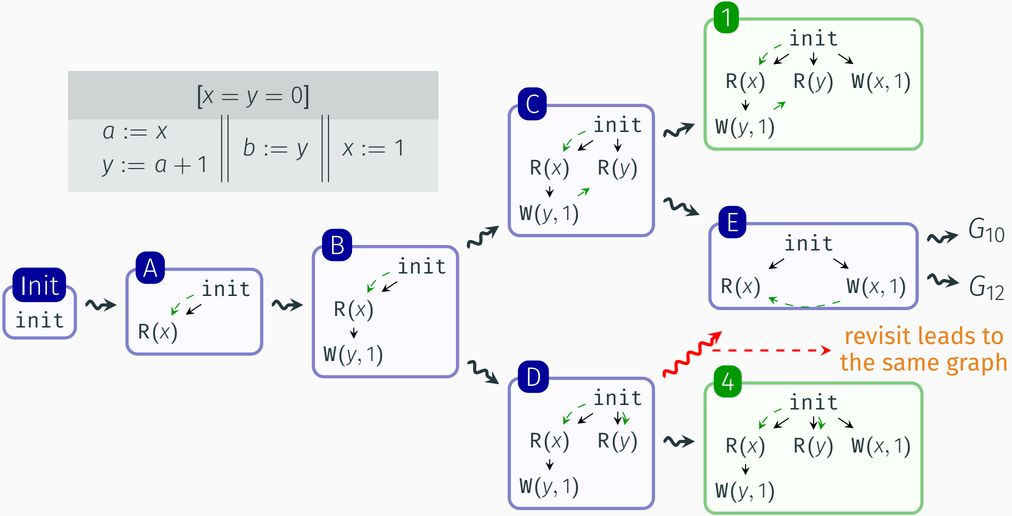
GenMC's algorithm: Example #4

$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$



GenMC's algorithm: Example #4

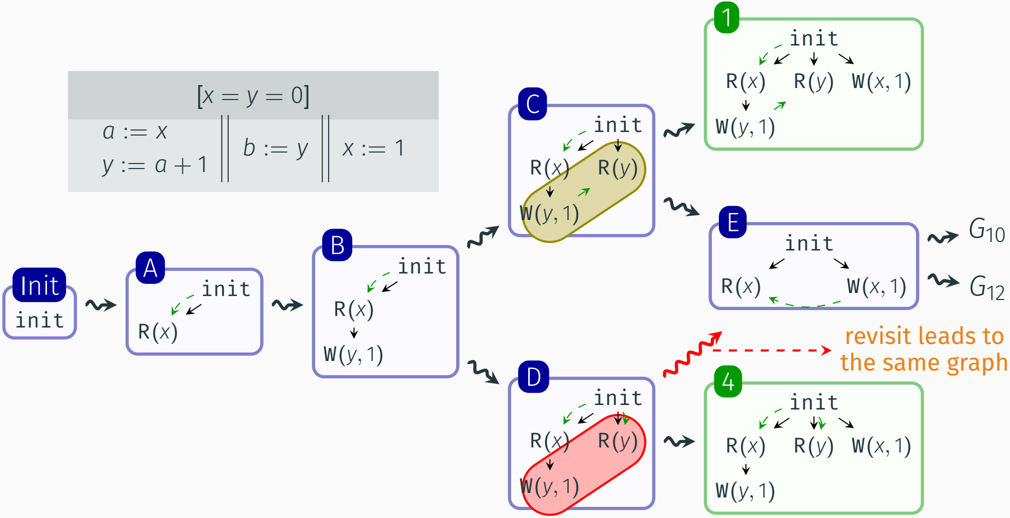
$[x = y = 0]$		
$a := x$ $y := a + 1$	$b := y$	$x := 1$



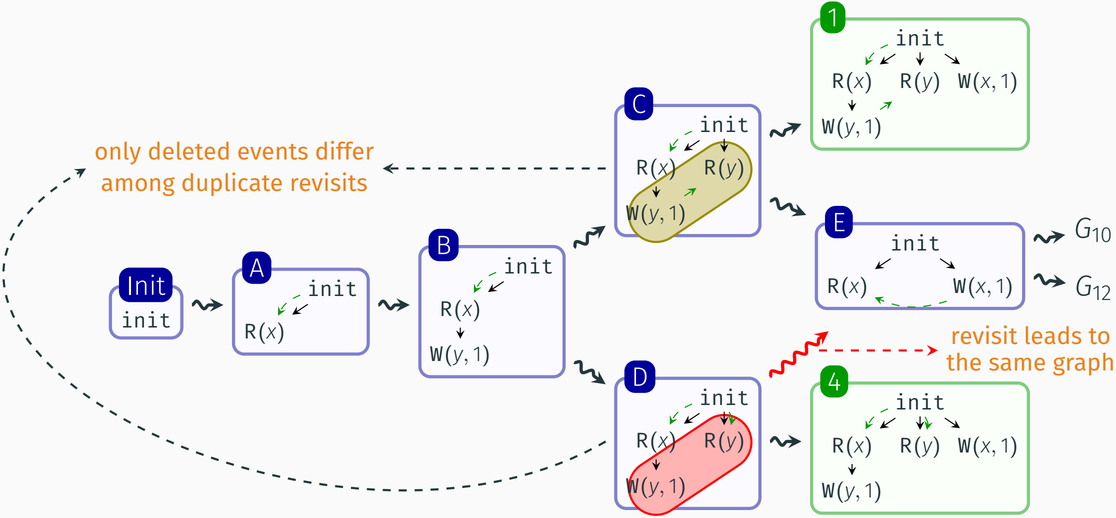
GENMC's algorithm: Example #4

$[x = y = 0]$

$a := x$ $y := a + 1$	$b := y$	$x := 1$



GenMC's algorithm: Example #4



Memory-model conditions

GenMC enumerates all consistent execution graphs for **any** memory model M, if

- $\text{cons}_M(\cdot)$ implies irreflexive($(\texttt{po} \cup \texttt{rf})^+$)
- $\text{cons}_M(\cdot)$ is **prefix-closed**
- $\text{cons}_M(\cdot)$ is **maximally extensible**

Memory-model conditions

GenMC enumerates all consistent execution graphs for **any** memory model M, if

- $\text{cons}_M(\cdot)$ implies irreflexive$((\text{po} \cup \texttt{rf})^+)$
- $\text{cons}_M(\cdot)$ is **prefix-closed**
- $\text{cons}_M(\cdot)$ is **maximally extensible**

These conditions hold for SC, TSO, PSO, RC11
(can be relaxed for POWER, ARM, IMM, LKMM)

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Demo #1

1. **Install Docker:**

   Debian/Ubuntu:
   ```
   apt install docker.io
   ```
   MacOS:
   ```
   brew install --cask docker
   ```
   Windows:
   ```
   wsl --install
   apt install docker.io
   ```

2. **Run GenMC container:**
   ```
   docker pull genmc/genmc
   docker run -it genmc/genmc:latest
   ```

3. **Download tutorial material:**
   ```
   wget https://plv.mpi-sws.org/genmc/popl2025/examples.tar.gz
   ```

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Approaches for state-space explosion

Partial order reduction (POR):

Avoid ordering independent actions

Approaches for state-space explosion

Partial order reduction (POR):

Avoid ordering independent actions

$a_1 = \ldots = a_N = 0$
$a_1 := 1 \parallel \ldots \parallel a_N := N$

$$\#\text{ of executions} \begin{cases} \text{SMC}: N! \\ \text{POR}: \quad 1 \\ \qquad \vdots \end{cases}$$

Approaches for state-space explosion

Partial order reduction (POR):

Avoid ordering independent actions

$a_1 = \ldots = a_N = 0$
$a_1 := 1 \parallel \ldots \parallel a_N := N$

$$\text{\# of executions} \begin{cases} \text{SMC}: & N! \\ \text{POR}: & 1 \\ & \vdots \end{cases}$$

Symmetry reduction (SR):

Avoid ordering symmetric threads

Approaches for state-space explosion

Partial order reduction (POR):

Avoid ordering independent actions

$a_1 = \ldots = a_N = 0$
$a_1 := 1 \parallel \ldots \parallel a_N := N$

$$\#\text{ of executions} \begin{cases} \text{SMC} : N! \\ \text{POR} : \ \ 1 \\ \text{SR} \ \ \ : N! \end{cases}$$

Symmetry reduction (SR):

Avoid ordering symmetric threads

Approaches for state-space explosion

Partial order reduction (POR):

Avoid ordering **independent actions**

$a_1 = \ldots = a_N = 0$
$a_1 := 1 \parallel \ldots \parallel a_N := N$

$$\# \text{ of executions} \begin{cases} \text{SMC}: N! \\ \text{POR}: 1 \\ \text{SR}: N! \end{cases}$$

Symmetry reduction (SR):

Avoid ordering **symmetric threads**

$x = 0$
`fetch_add`$(x, 1) \parallel \ldots \parallel$ `fetch_add`$(x, 1)$

$$\# \text{ of executions} \begin{cases} \text{SMC}: N! \\ \text{POR}: N! \\ \text{SR}: 1 \end{cases}$$

Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$$x = a[1] = \ldots = a[N] = 0$$

$$\begin{array}{l} i := \texttt{fetch\_add}(x, 1) \\ a[i] := i \end{array} \;\Big\|\; \ldots \;\Big\|\; \begin{array}{l} i := \texttt{fetch\_add}(x, 1) \\ a[i] := i \end{array}$$

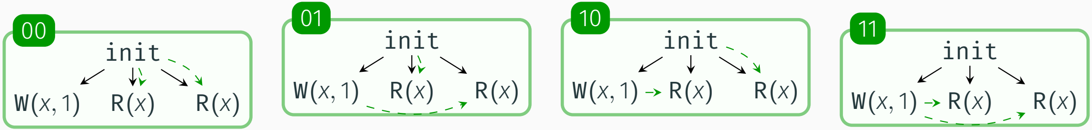
SMC	: $(2N)!/(N \cdot 2!)$
POR	: $N!$
SR	: $(2N-1)!!$
POR +SR	: 1

Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

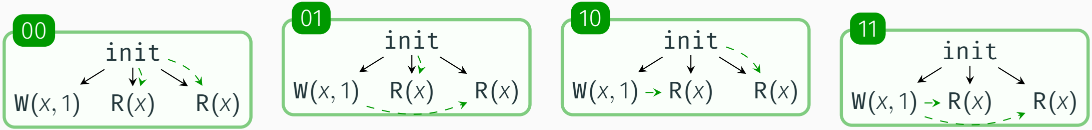
T1: $x := 1$ || T2: $r := x$ || T3: $r := x$



Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$		
T1: $x := 1$	T2: $r := x$	T3: $r := x$

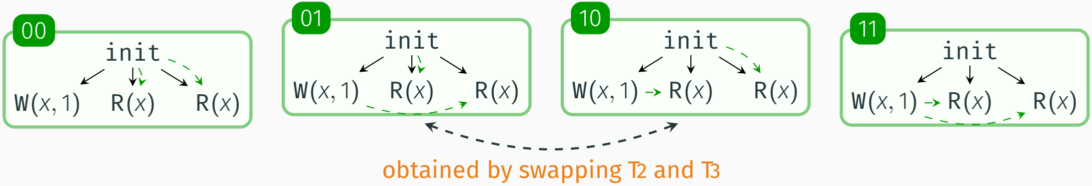


Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

Key Idea: Identify symmetries on the execution graphs

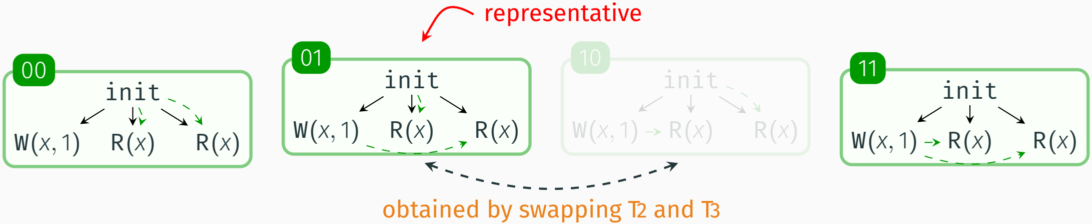


Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

Key Idea: Identify symmetries on the execution graphs

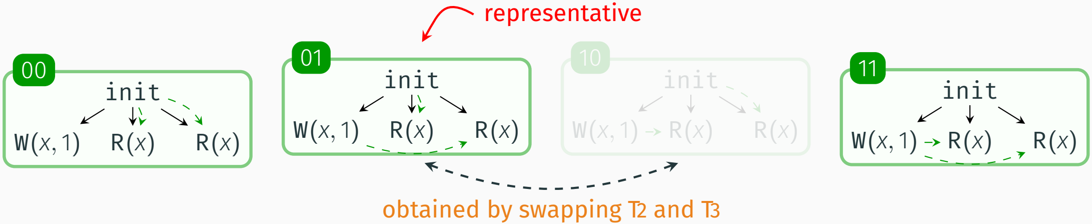


Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

Key Idea: Identify symmetries on the execution graphs
Only generate representative graphs

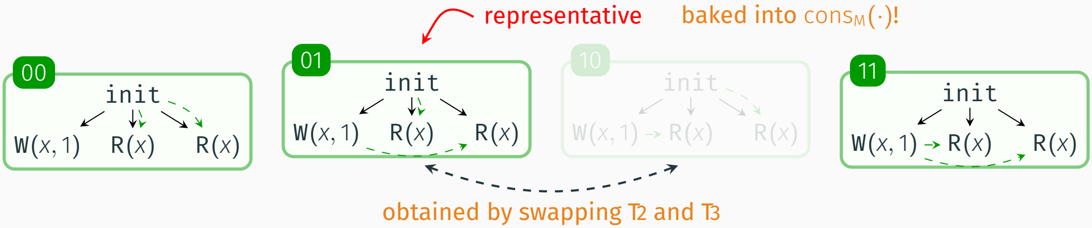


Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

Key Idea: Identify symmetries on the execution graphs
Only generate representative graphs



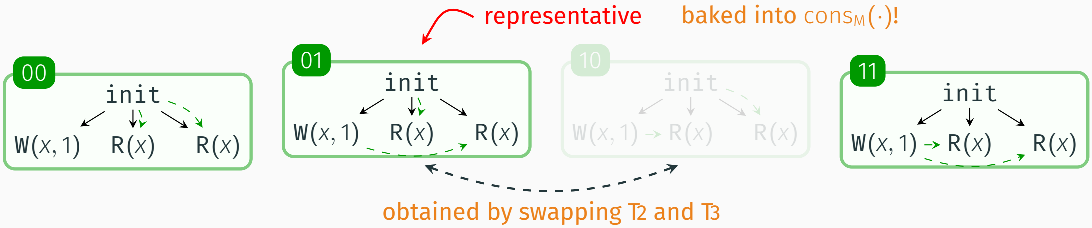
Combining SR and POR

[PLDI'24] SPORE: Combining symmetry and partial order reduction

$[x = 0]$

Key Idea: Identify symmetries on the execution graphs

Only generate **representative** graphs



There are many types of symmetries

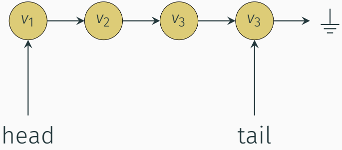
$\rightsquigarrow$ these can be incorporated into $\mathsf{cons}_M(\cdot)$

Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail)  continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```

Internal symmetries example: Michael-Scott queue

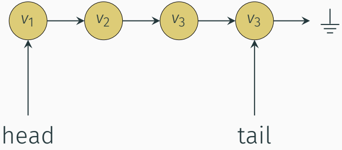
```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```



Internal symmetries example: Michael-Scott queue

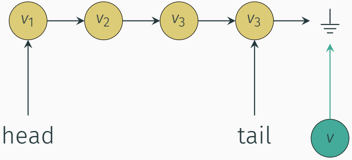
T1 ➡

```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```



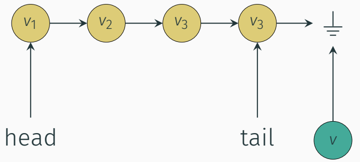
Internal symmetries example: Michael-Scott queue

```
   enqueue(v) ≜
T1   node := malloc(...)
     node.value := v
     node.next := NULL
     do
       t := tail
       next := t.next
       if (t ≠ tail) continue
       if (next ≠ NULL)
         CAS(tail, t, next)
         continue
     while (¬CAS(t.next, next, node))
     CAS(tail, t, node)
```



Internal symmetries example: Michael-Scott queue

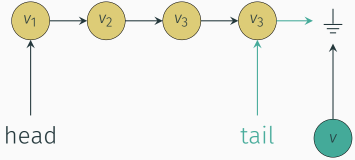
enqueue(v) $\triangleq$
  $node$:= malloc(...)
  $node.value$:= v
  $node.next$:= NULL
  do
T1 ➡ t := $\underline{tail}$
    $next$:= $t.next$
    if ($t \neq \underline{tail}$) **continue**
    if ($next \neq$ NULL)
      CAS($\underline{tail}$, t, $next$)
      continue
  while ($\neg$CAS($t.next$, $next$, $node$))
  CAS($\underline{tail}$, t, $node$)



Internal symmetries example: Michael-Scott queue

T1 ➡ (at line $t := \underline{tail}$)

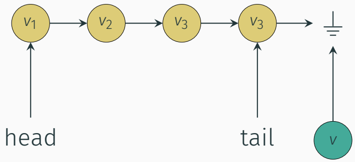
```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```



Internal symmetries example: Michael-Scott queue

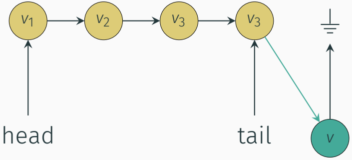
```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```

T1 → (at `while (¬CAS(t.next, next, node))`)



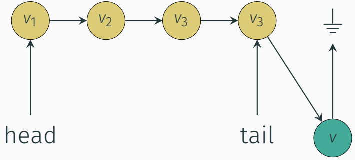
Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
T1 →  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)
```



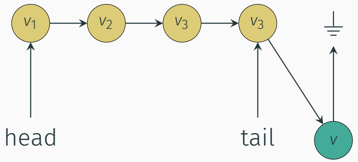
Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
T1 → CAS(tail, t, node)
```



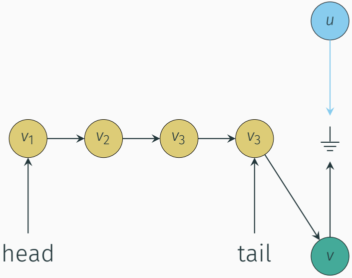
Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
T2 → node := malloc(...)
     node.value := v
     node.next := NULL
     do
        t := tail
        next := t.next
        if (t ≠ tail) continue
        if (next ≠ NULL)
           CAS(tail, t, next)
           continue
     while (¬CAS(t.next, next, node))
T1 → CAS(tail, t, node)
```



Internal symmetries example: Michael-Scott queue

```
    enqueue(v) ≜
T2 ➡  node := malloc(...)
      node.value := v
      node.next := NULL
      do
        t := tail
        next := t.next
        if (t ≠ tail) continue
        if (next ≠ NULL)
          CAS(tail, t, next)
          continue
      while (¬CAS(t.next, next, node))
T1 ➡  CAS(tail, t, node)
```



Internal symmetries example: Michael-Scott queue

enqueue(v) $\triangleq$

  $node$:= malloc(...)

  $node.value$:= v

  $node.next$:= NULL

  do

T2 → t := $\underline{tail}$

    $next$:= $t.next$

    if ($t \neq \underline{tail}$) **continue**

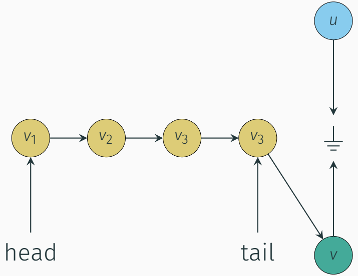
    if ($next \neq$ NULL)

      CAS($\underline{tail}$, t, $next$)

      continue

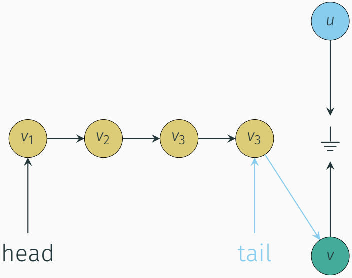
  while ($\neg$CAS($t.next$, $next$, $node$))

T1 → CAS($\underline{tail}$, t, $node$)



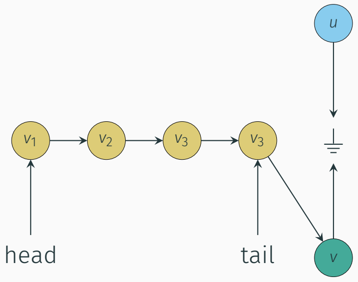
Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
    node := malloc(...)
    node.value := v
    node.next := NULL
    do
T2      t := tail
        next := t.next
        if (t ≠ tail) continue
        if (next ≠ NULL)
            CAS(tail, t, next)
            continue
    while (¬CAS(t.next, next, node))
T1  CAS(tail, t, node)
```



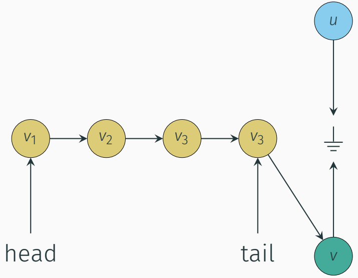
Internal symmetries example: Michael-Scott queue

enqueue(v) $\triangleq$
 $node$:= malloc(...)
 $node.value$:= v
 $node.next$:= NULL
 do
  t := $\underline{tail}$
  $next$:= $t.next$
  if ($t \neq \underline{tail}$) **continue**
  if ($next \neq$ NULL)
T2 → CAS($\underline{tail}$, t, $next$)
   continue
 while ($\neg$CAS($t.next$, $next$, $node$))
T1 → CAS($\underline{tail}$, t, $node$)



Internal symmetries example: Michael-Scott queue

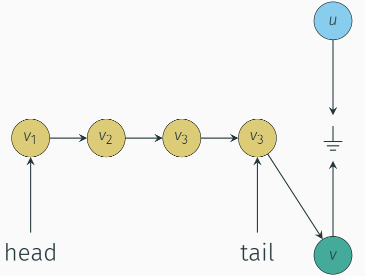
enqueue(v) $\triangleq$
  $node$:= malloc(...)
  $node.value$:= v
  $node.next$:= NULL
  do
    t := $\underline{tail}$
    $next$:= $t.next$
    if ($t \neq \underline{tail}$) **continue**
    if ($next \neq$ NULL)
T2   CAS($\underline{tail}$, t, $next$)
      continue
  while ($\neg$CAS($t.next$, $next$, $node$))
T1 CAS($\underline{tail}$, t, $node$)



Contention on tail hinders verification!

Internal symmetries example: Michael-Scott queue

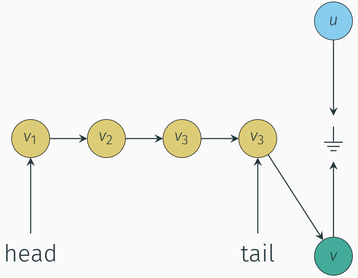
```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
      CAS(tail, t, next)      ← T2
      continue
  while (¬CAS(t.next, next, node))
  CAS(tail, t, node)          ← T1
```



Contention on tail hinders verification! → $K!$ overhead for K threads

Internal symmetries example: Michael-Scott queue

```
enqueue(v) ≜
  node := malloc(...)
  node.value := v
  node.next := NULL
  do
    t := tail
    next := t.next
    if (t ≠ tail) continue
    if (next ≠ NULL)
T2 ⇒  CAS(tail, t, next)
      continue
  while (¬CAS(t.next, next, node))
T1 ⇒ CAS(tail, t, node)
```



Contention on tail hinders verification! → $K!$ overhead for K threads

Observe: The conflicting operations are identical and idempotent

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires annotating the input program

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires annotating the input program

Key Idea: only explore executions where *main* succeeds

GenMC leverages idempotent operations by splitting them to *main* and *helping*
⇝ this requires **annotating** the input program

Key Idea: only explore executions where *main* succeeds

Is this sound?

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires annotating the input program

Key Idea: only explore executions where *main* succeeds

Is this sound?

$x = y = 0$

T1:	T2:	T3:
$y := 1$ CAS$(x, 0, 1)$ (main)	CAS$(x, 0, 1)$ (helping)	`if` $(x = 1)$ `assert`$(y = 1)$

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires **annotating** the input program

Key Idea: only explore executions where *main* succeeds

Is this sound?

reading only from *main*

misses the bug!

$x = y = 0$

T1:	T2:	T3:
$y := 1$ $\mathrm{CAS}(x, 0, 1)$	$\mathrm{CAS}(x, 0, 1)$	**if** $(x = 1)$ assert$(y = 1)$

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires **annotating** the input program

Key Idea: **only explore executions where *main* succeeds**

reading only from *main* misses the bug!

Is this sound?

$x = y = 0$

T1	T2	T3
$y := 1$ $\mathrm{CAS}(x, 0, 1)$ (*main*)	$\mathrm{CAS}(x, 0, 1)$ (*helping*)	**if** $(x = 1)$ `assert`$(y = 1)$

GenMC presents **sufficient conditions** for leveraging idempotent operations

⇝ *main* and *helping* can be functions …

⇝ but they have to satisfy certain conditions (e.g., induce same synchronization)

GenMC leverages idempotent operations by splitting them to *main* and *helping*

⇝ this requires **annotating** the input program

Key Idea: only explore executions where *main* succeeds

Is this sound?

reading only from *main* misses the bug!

$x = y = 0$

T1: $y := 1$	T2: $\mathrm{CAS}(x, 0, 1)$	T3: **if** $(x = 1)$
$\mathrm{CAS}(x, 0, 1)$		assert$(y = 1)$

GenMC presents **sufficient conditions** for leveraging idempotent operations

⇝ *main* and *helping* can be functions ...

⇝ but they have to satisfy certain conditions (e.g., induce same synchronization)

Internal symmetries can also be leveraged in non-symmetric programs!

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

How to estimate the state-space size?

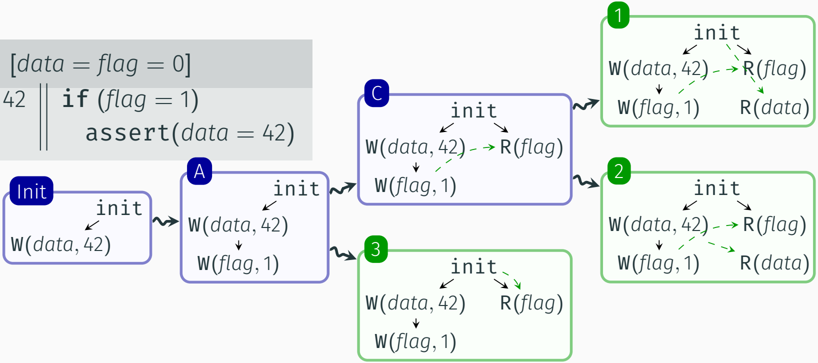
$$[data = flag = 0]$$

$data := 42$ $flag := 1$	$\parallel$	`if` $(flag = 1)$ `assert`$(data = 42)$

How to estimate the state-space size?

$[data = flag = 0]$

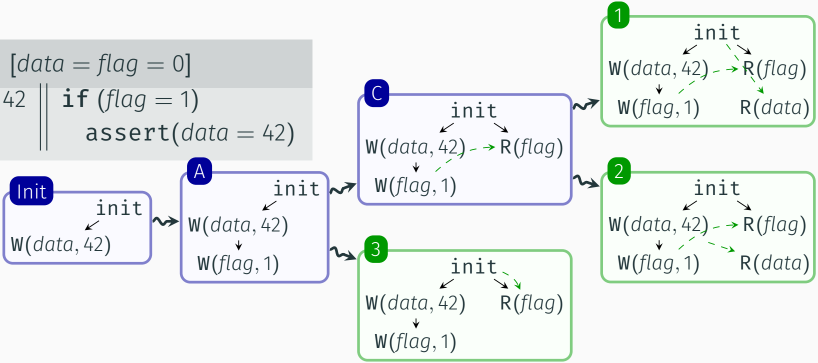
$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



How to estimate the state-space size?

$$[\mathit{data} = \mathit{flag} = 0]$$

$\mathit{data} := 42$	$\texttt{if}\ (\mathit{flag} = 1)$
$\mathit{flag} := 1$	$\quad\texttt{assert}(\mathit{data} = 42)$



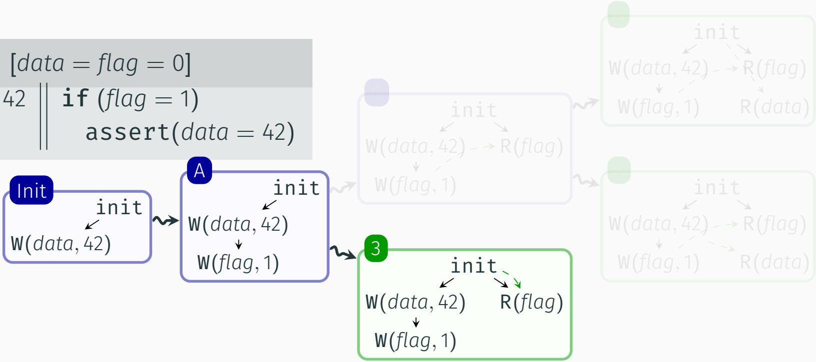
Naive “solution”:

- Assume symmetric state space
- Estimate based on explored space

How to estimate the state-space size?

$[data = flag = 0]$

$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



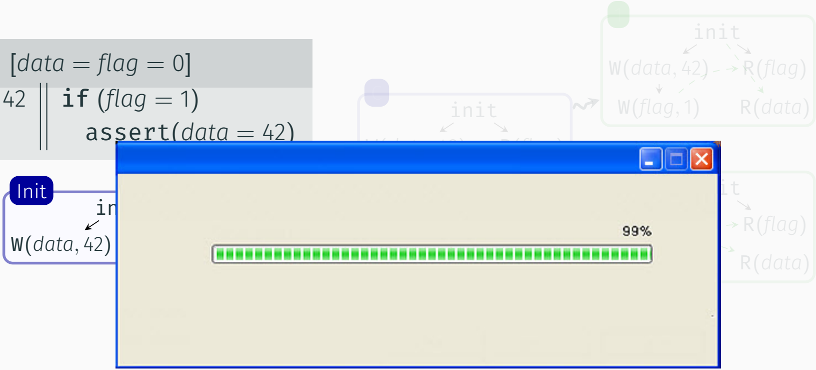
Naive “solution”:

- Assume symmetric state space
- Estimate based on explored space

How to estimate the state-space size?

$[data = flag = 0]$

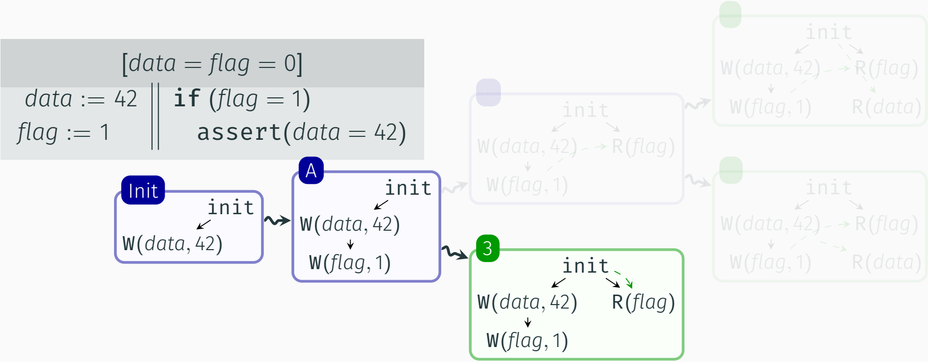
$data := 42$	`if` $(flag = 1)$
$flag := 1$	`assert`$(data = 42)$



Naive "solution":

- Assume symmetric state space
- Estimate based on explored space

How to estimate the state-space size?



Naive "solution":

- Assume symmetric state space
- Estimate based on explored space

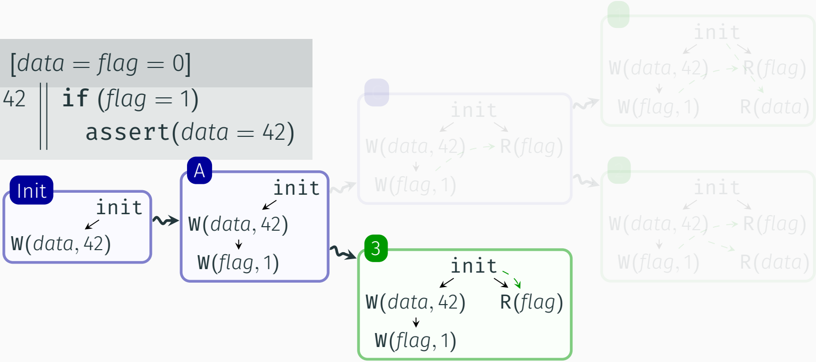
Our Idea: Monte Carlo Simulation before verification

- Take random samples (assuming symmetric space)
- Law of large numbers guarantees accuracy (if unbiased)

How to estimate the state-space size?

$[data = flag = 0]$

$data := 42$ $flag := 1$	`if` $(flag = 1)$ `assert`$(data = 42)$



Naive "solution":

- Assume symmetric state space
- Estimate based on explored space

Our Idea: Monte Carlo Simulation before verification

- Take random samples (assuming symmetric space)
- Law of large numbers guarantees accuracy (if unbiased)

Reducing bias in estimation

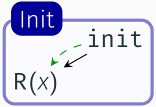
$$[x = 0]$$

$$\begin{array}{l} a := x \\ \textbf{if}\ (a > 0)\ b := x \end{array} \;\Big\|\; x := 1 \;\Big\|\; x := 2$$

Reducing bias in estimation

$$[x = 0]$$

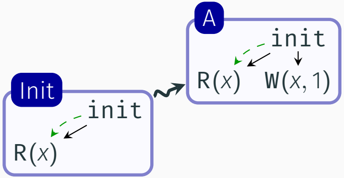
$$\begin{array}{l} a := x \\ \textbf{if } (a > 0)\ b := x \end{array} \;\Big\|\; x := 1 \;\Big\|\; x := 2$$



Reducing bias in estimation

$$[x = 0]$$

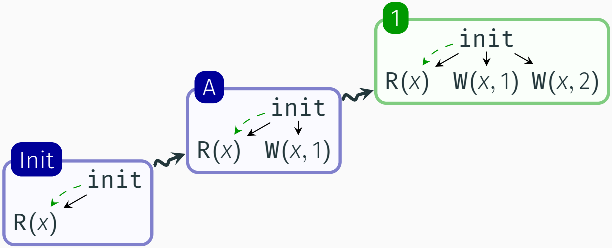
$$\begin{array}{l} a := x \\ \textbf{if } (a > 0)\ b := x \end{array} \;\Big\|\; x := 1 \;\Big\|\; x := 2$$



Reducing bias in estimation

$$[x = 0]$$

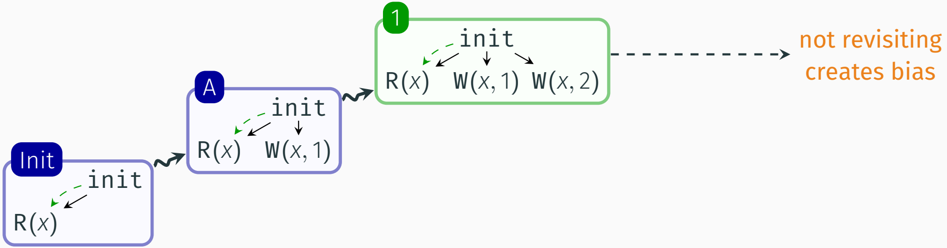
$$\begin{array}{l} a := x \\ \textbf{if } (a > 0)\ b := x \end{array} \;\Big\|\; x := 1 \;\Big\|\; x := 2$$



Reducing bias in estimation

$[x = 0]$

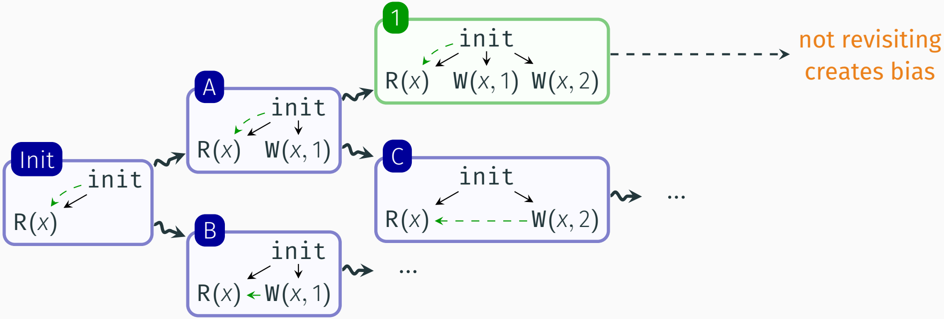
$a := x$; **if** $(a > 0)\ b := x$ $\parallel$ $x := 1$ $\parallel$ $x := 2$



Reducing bias in estimation

$[x = 0]$

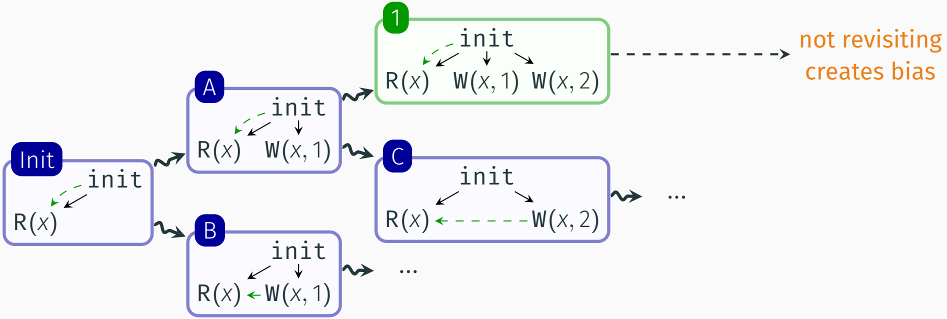
$a := x$ **if** $(a > 0)$ $b := x$	$x := 1$	$x := 2$



Reducing bias in estimation

$[x = 0]$

$a := x$ **if** $(a > 0)$ $b := x$	$x := 1$	$x := 2$

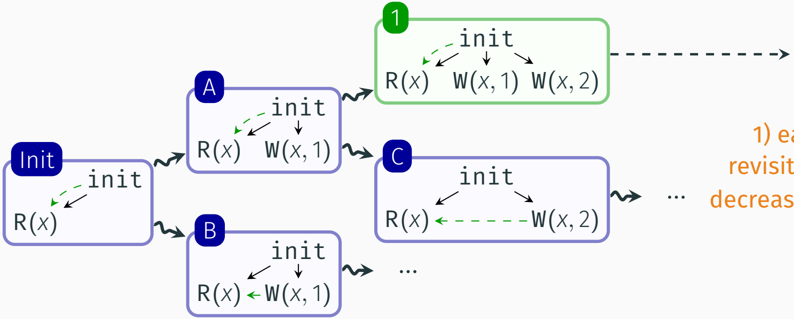


Problem: When to perform a revisit?

Reducing bias in estimation

$[x = 0]$

$a := x$; **if** $(a > 0)$ $b := x$ $\|$ $x := 1$ $\|$ $x := 2$



not revisiting creates bias

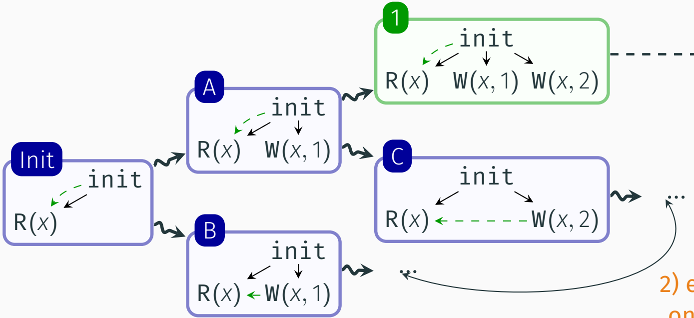
1) each read gets one revisit: revisit probability for later writes decreases exponentially ($p(1-p)\ldots$)

Problem: When to perform a revisit?

Reducing bias in estimation

$[x = 0]$

$a := x$ **if** $(a > 0)$ $b := x$	$x := 1$	$x := 2$



not revisiting creates bias

1) each read gets one revisit: revisit probability for later writes decreases exponentially ($p(1-p)\ldots$)

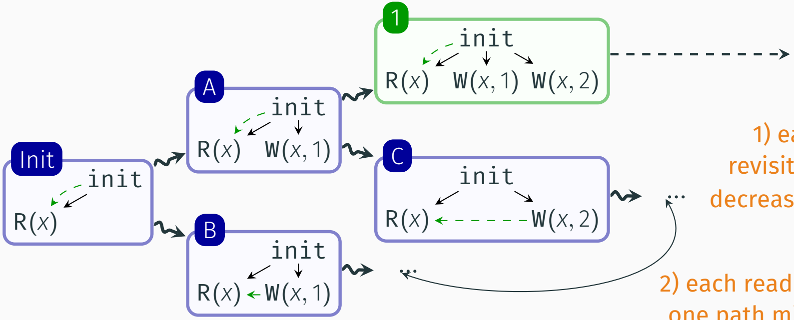
2) each read gets multiple revisits: one path might lead to the other $\rightsquigarrow$ long estimation time

Problem: When to perform a revisit?

Reducing bias in estimation

$[x = 0]$

$$\begin{array}{l} a := x \\ \textbf{if } (a > 0)\ b := x \end{array} \;\Big\|\; x := 1 \;\Big\|\; x := 2$$



not revisiting creates bias

1) each read gets one revisit: revisit probability for later writes decreases exponentially ($p(1-p)\ldots$)

2) each read gets multiple revisits: one path might lead to the other $\rightsquigarrow$ long estimation time

Problem: When to perform a revisit?

Our Solution: No revisits — random scheduler that prioritizes writes over reads

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

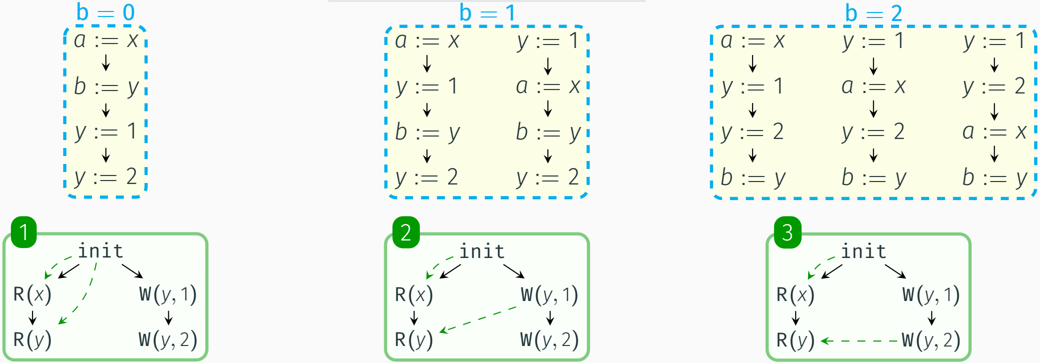
Bounded exploration and POR

$[x = y = 0]$		
$a := x$	$\|\|$	$y := 1$
$b := y$		$y := 2$

Bounded exploration and POR

$[x = y = 0]$

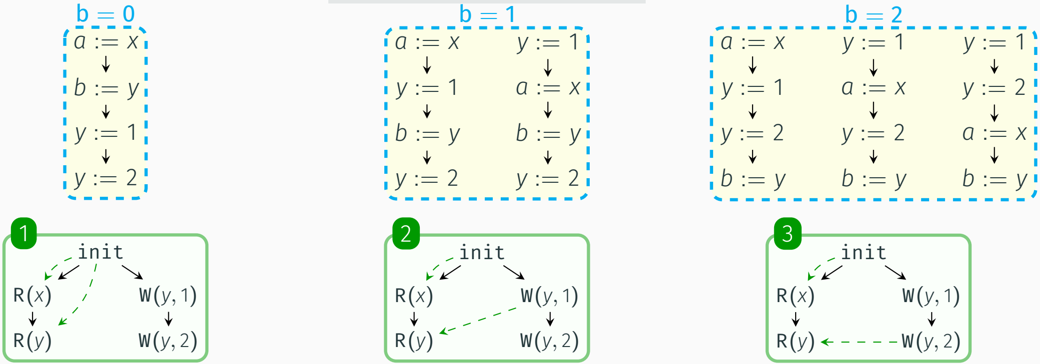
$a := x$	$y := 1$
$b := y$	$y := 2$



Bounded exploration and POR

$$[x = y = 0]$$

$$\begin{array}{r||l} a := x & y := 1 \\ b := y & y := 2 \end{array}$$

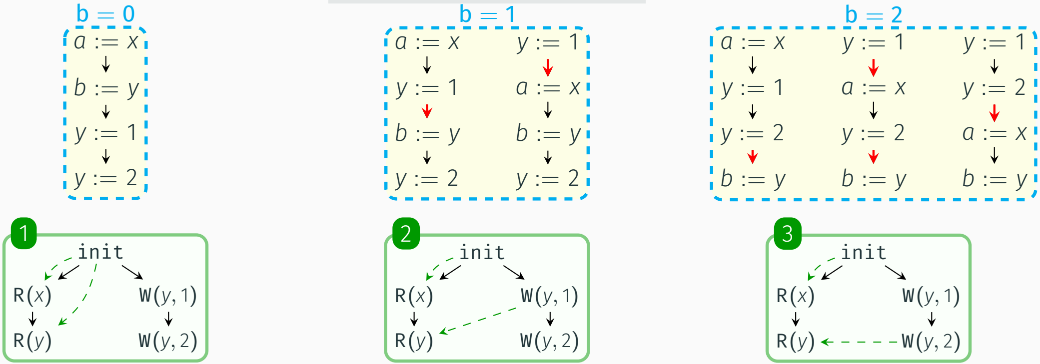


Goal: Only explore graphs $G \in [\![P]\!]$ where exists $t \in \mathrm{trace}(G)$ s.t. $B(t) \leq K$

Bounded exploration and POR

$[x = y = 0]$

$a := x$	$y := 1$
$b := y$	$y := 2$



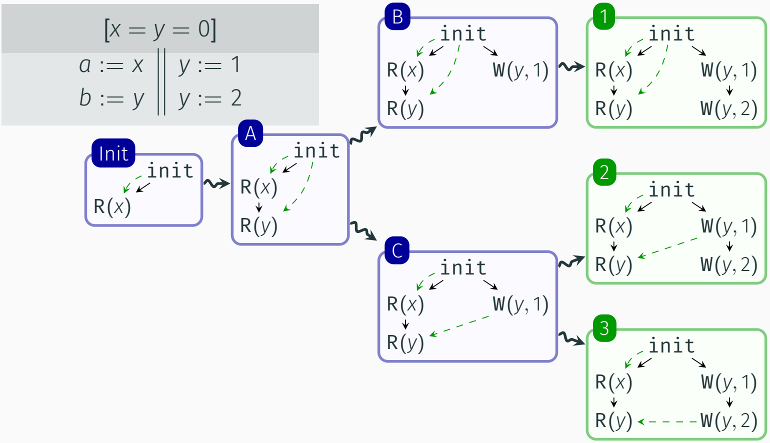
Goal: Only explore graphs $G \in [\![P]\!]$ where exists $t \in \text{trace}(G)$ s.t. $B(t) \leq K$

here: round-robin rounds

Bounded POR

$[x = y = 0]$		
$a := x$	$\|\|$	$y := 1$
$b := y$		$y := 2$

Bounded POR

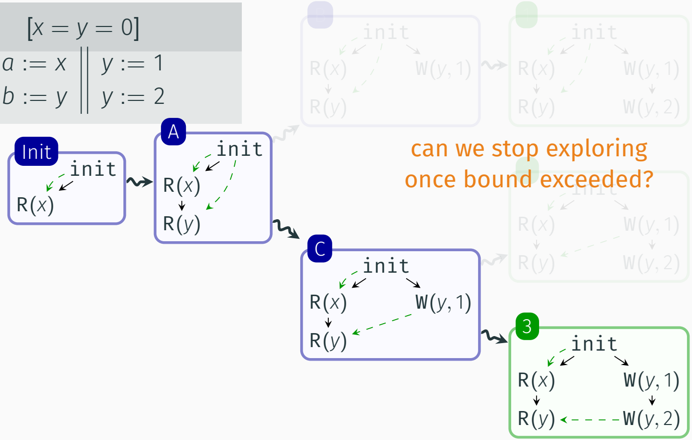


[x = y = 0]
a := x || y := 1
b := y || y := 2
Init
init
R(x)
A
init
R(x)
R(y)
B
init
R(x)
W(y, 1)
R(y)
C
init
R(x)
W(y, 1)
R(y)
1
init
R(x)
W(y, 1)
R(y)
W(y, 2)
2
init
R(x)
W(y, 1)
R(y)
W(y, 2)
3
init
R(x)
W(y, 1)
R(y)
W(y, 2)

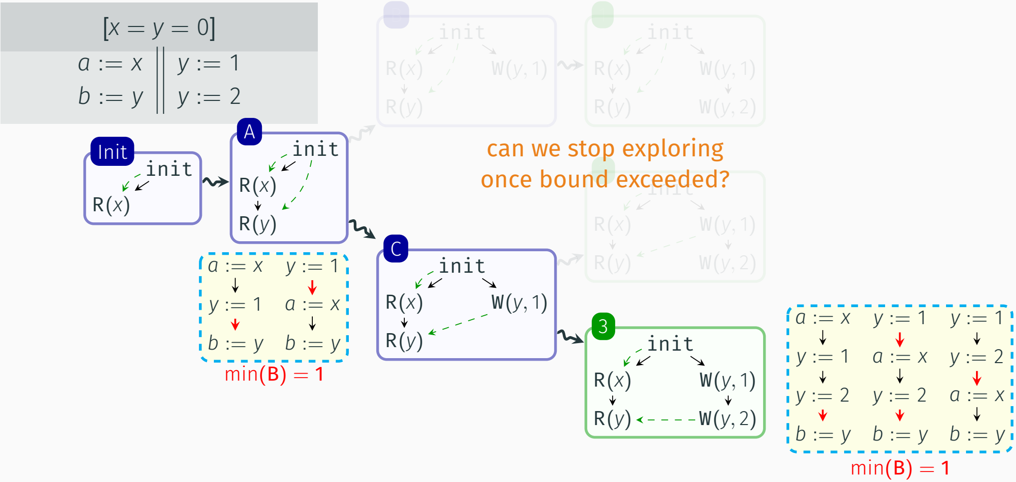
Bounded POR

$$[x = y = 0]$$

$a := x$	‖	$y := 1$
$b := y$	‖	$y := 2$



Bounded POR

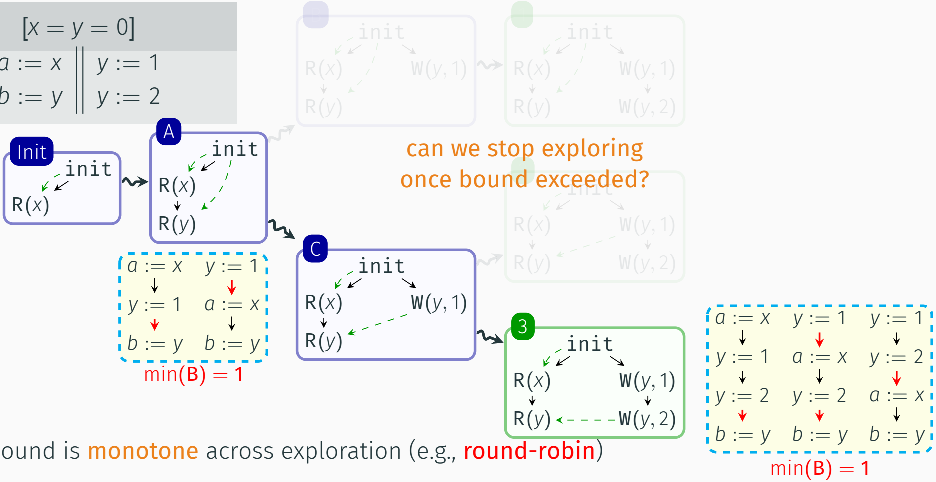


[x = y = 0]
a := x || y := 1
b := y || y := 2
Init
init
R(x)
A
init
R(x)
R(y)
a := x y := 1
y := 1 a := x
b := y b := y
min(B) = 1
C
init
R(x) W(y, 1)
R(y)
can we stop exploring
once bound exceeded?
3
init
R(x) W(y, 1)
R(y) W(y, 2)
a := x y := 1 y := 1
y := 1 a := x y := 2
y := 2 y := 2 a := x
b := y b := y b := y
min(B) = 1

Bounded POR

$[x = y = 0]$

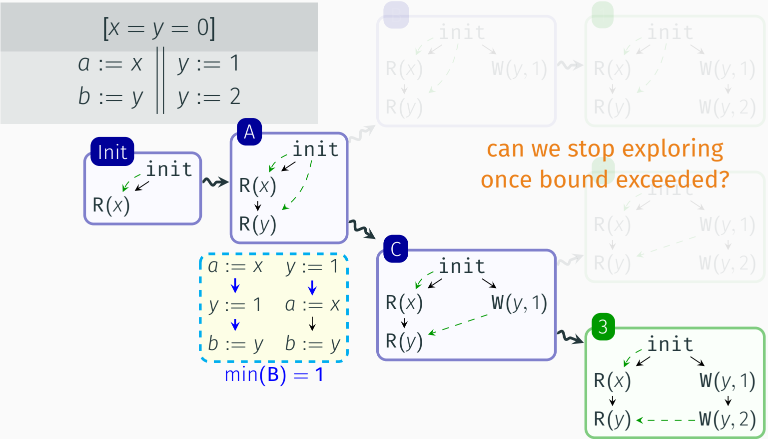
$a := x$	$y := 1$
$b := y$	$y := 2$

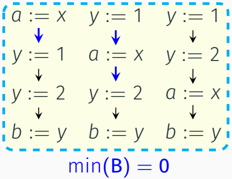


can we stop exploring once bound exceeded?

Yes, if bound is monotone across exploration (e.g., round-robin)

Bounded POR





Yes, if bound is monotone across exploration (e.g., round-robin)

$\rightsquigarrow$ **context bounding**: intermediate graphs might have bound $\leq T + K - 2$

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo

Outline

How does GenMC work?

- SMC basics
- Execution graphs
- Exploration algorithm

How to apply GenMC to our code?

- State-space reductions
- Estimating state-space size
- Exploration bounding

Each part will be followed by a demo