Towards Efficient Verification of Parallel Applications with Mc SimGrid

Joint work with Martin Quinson (Magellan) and Thierry Jéron (Devine)

Mathieu Laurent

February 20, 2025

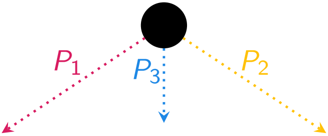
Distributed computing

- HPC applications are distributed and concurrent
- Data shared via messages (e.g. MPI) or synchronizations (e.g. thread)
- Causes non-deterministic bugs
- Software model checking covers all cases

Content of this talk

(with ongoing work 📢)

Exploring the transition systems



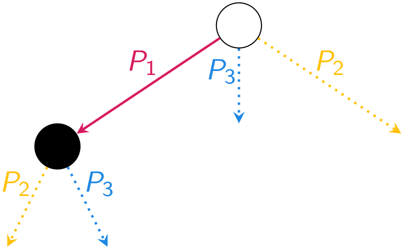
A small MPI example

P_1/P_2	P_3
$\texttt{Send}(P_3)$	$\texttt{Recv}()$
	$\texttt{Recv}()$

Exploring the transition systems

A small MPI example

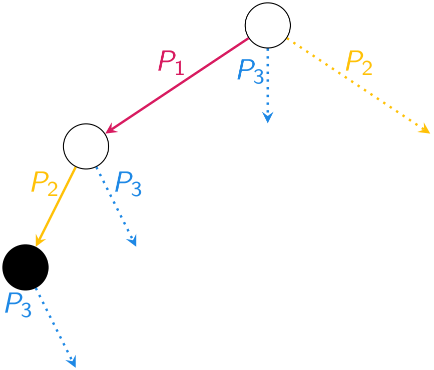
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

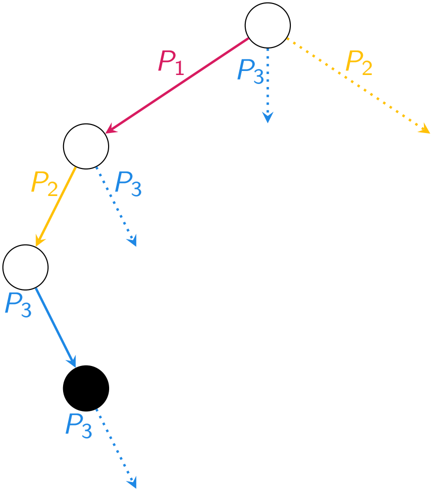
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

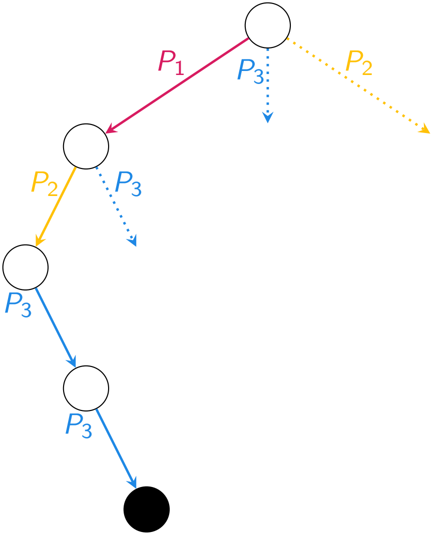
P_1/P_2	P_3
`Send`(P_3)	`Recv()`
	`Recv()`



Exploring the transition systems

A small MPI example

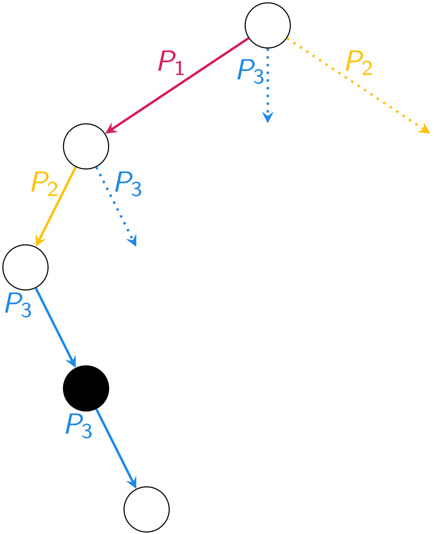
P_1 / P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

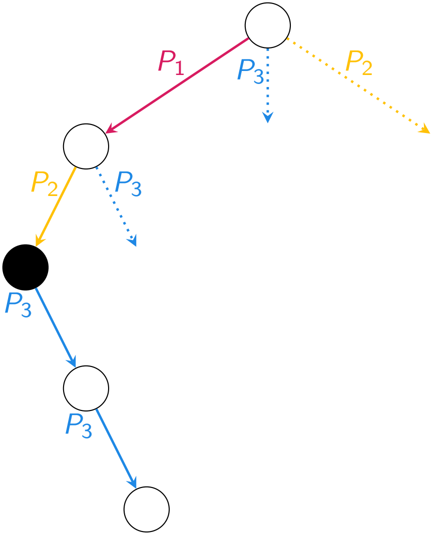
P_1/P_2	P_3
`Send`(P_3)	`Recv()`
	`Recv()`



Exploring the transition systems

A small MPI example

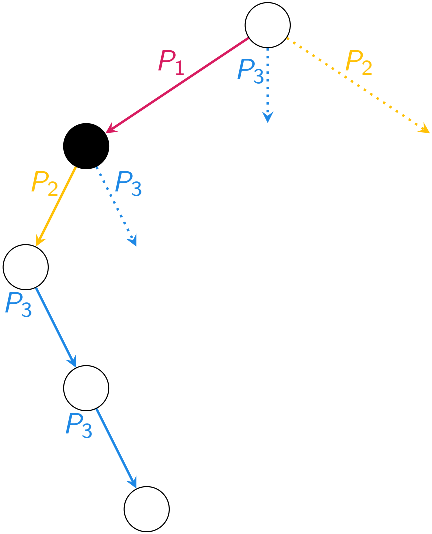
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

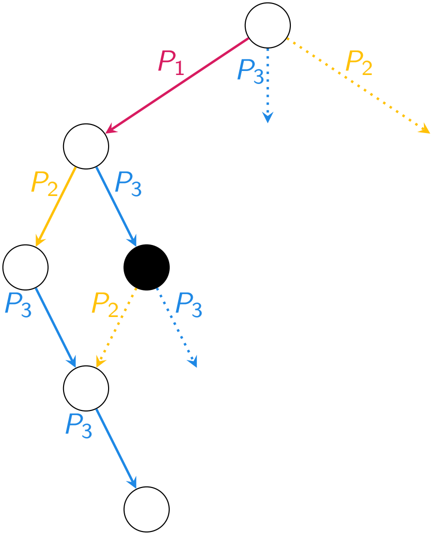
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

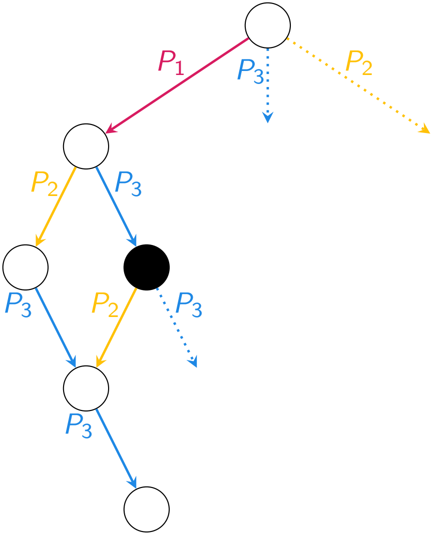
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

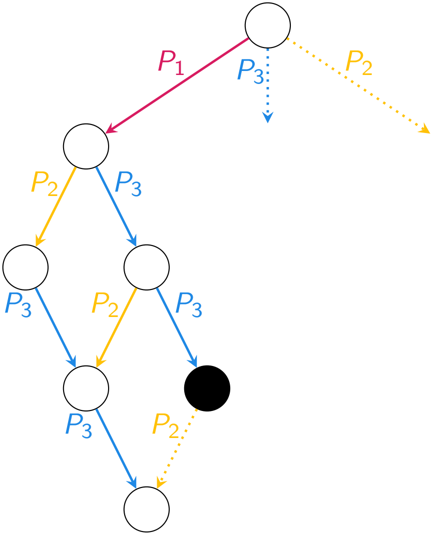
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()



Exploring the transition systems

A small MPI example

P_1/P_2	P_3
`Send`(P_3)	`Recv()`
	`Recv()`



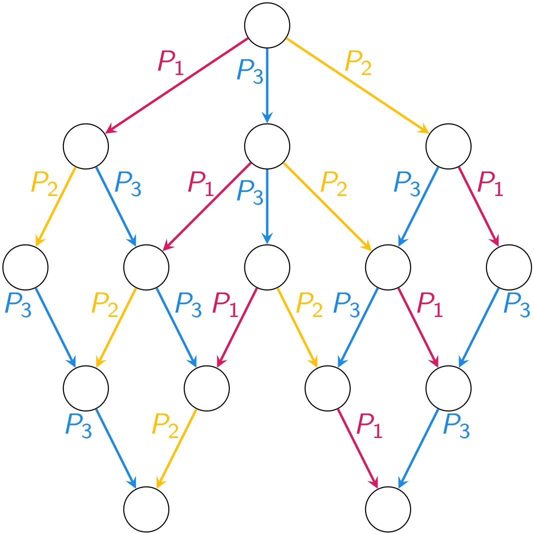
Exploring the transition systems

A small MPI example

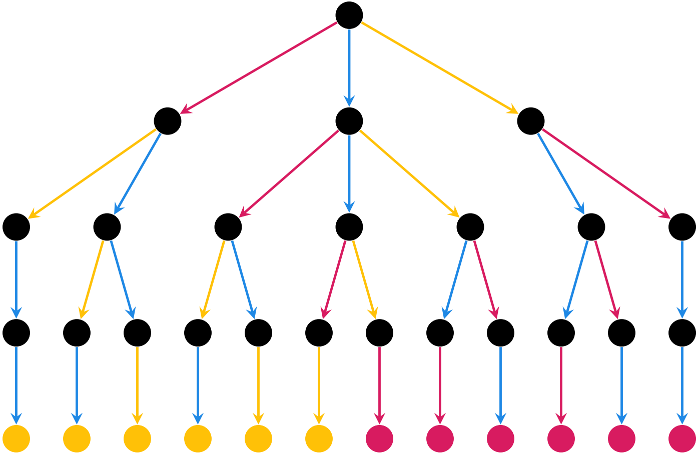
P_1/P_2	P_3
Send(P_3)	Recv()
	Recv()

Stateful exploration

15 states for 2 behaviors.



Stateless model checking

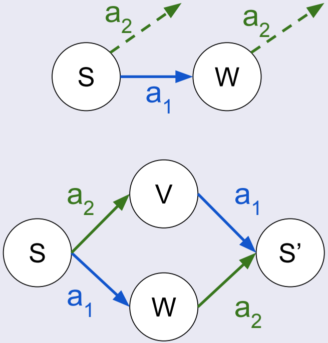


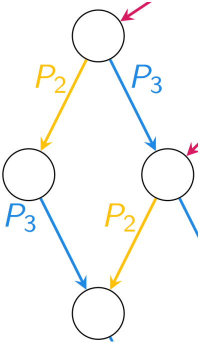
Stateless exploration

35 states for the same 2 behaviors.

Transition dependency

Two actions a_1, a_2 are independent if:



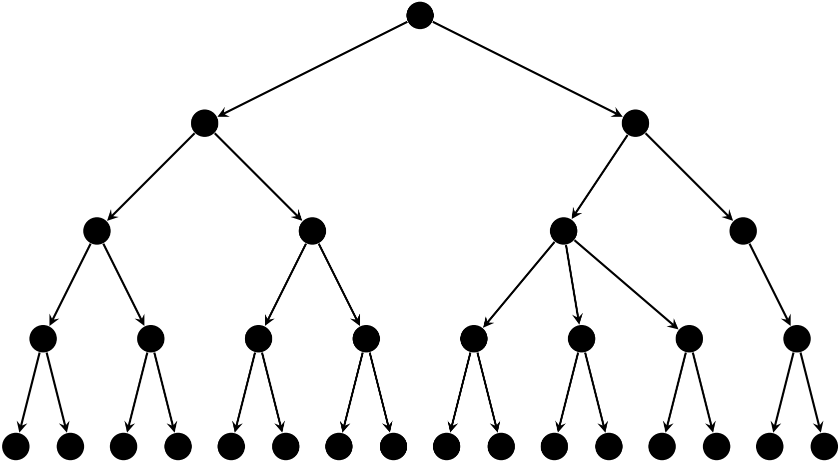


Example of two adjacent independent actions

Mazurkiewicz's traces [Maz'77]

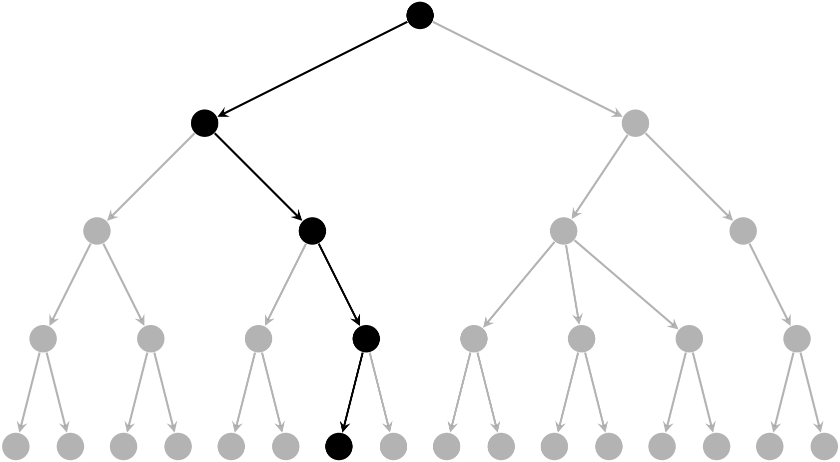
Equivalence class of executions with adjacent independent actions swapped

DPOR approach [Fla'05]



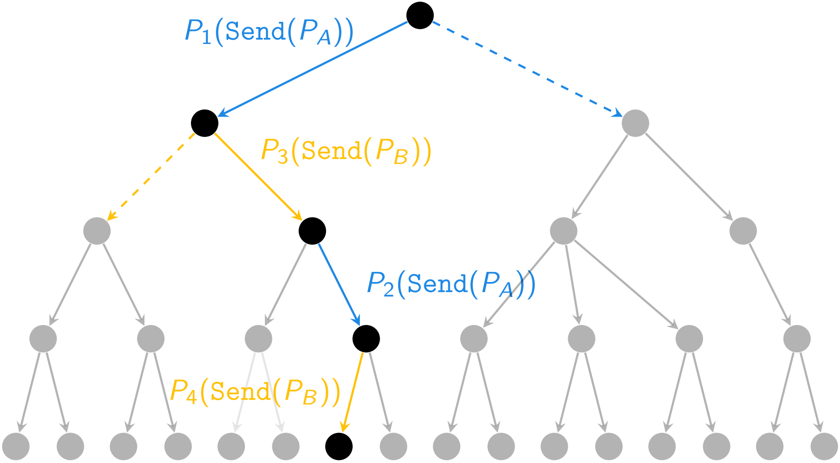
Classical depth first search algorithm

DPOR approach [Fla'05]



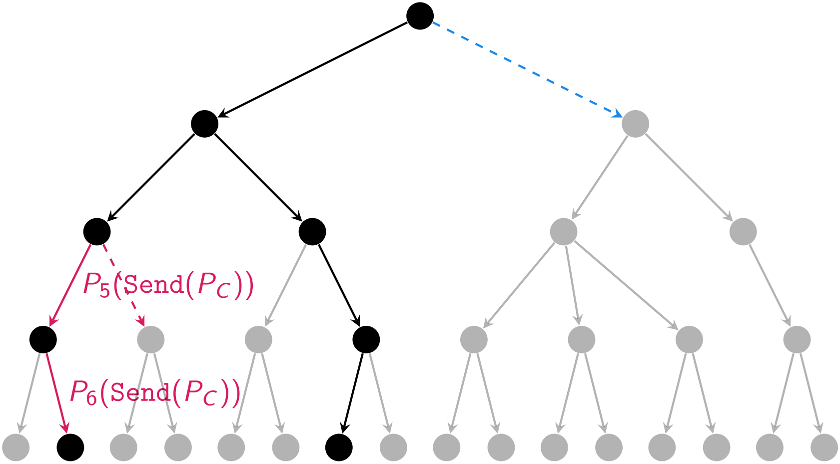
Start with an arbitrary execution

DPOR approach [Fla'05]



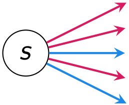
Discover dependencies

DPOR approach [Fla'05]



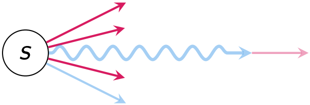
Recursive DFS exploration of what has been added

DPOR: persistent sets

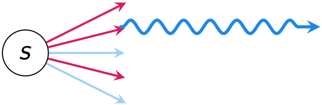


A set T of transitions from s is persistent iff for any sequence of transitions not in T starting from s, t' is independent with T

It is sufficient to only explore transitions in a persistent set:



equivalent to



DPOR: persistent sets

Five transitions enabled in s

$P_1(\texttt{Send}(P_A))$
$P_2(\texttt{Send}(P_A))$
$P_3(\texttt{Send}(P_B))$
$P_4(\texttt{Send}(P_B))$
$P_5(\texttt{LocalOp}())$

- $\{P_1, P_2\}$, $\{P_3, P_4, P_5\}$, $\{P_1, P_2, P_3, P_4\}, \ldots$ are persistent sets
- $\{P_1\}, \{P_2, P_3\}, \ldots$ are not persistent sets

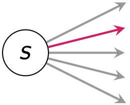
DPOR: persistent sets

Five transitions enabled in s

$P_1(\texttt{Send}(P_A))$
$P_2(\texttt{Send}(P_A))$
$P_3(\texttt{Send}(P_B))$
$P_4(\texttt{Send}(P_B))$
$P_5(\texttt{LocalOp}())$

- $\{P_1, P_2\}$, $\{P_3, P_4, P_5\}$, $\{P_1, P_2, P_3, P_4\}, \ldots$ are persistent sets
- $\{P_1\}, \{P_2, P_3\}, \ldots$ are not persistent sets

DPOR builds persistent sets iteratively for each state



Pick an arbitrary transition

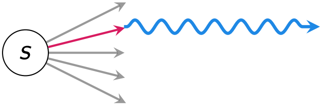
DPOR: persistent sets

Five transitions enabled in s

$P_1(\texttt{Send}(P_A))$
$P_2(\texttt{Send}(P_A))$
$P_3(\texttt{Send}(P_B))$
$P_4(\texttt{Send}(P_B))$
$P_5(\texttt{LocalOp}())$

- $\{P_1, P_2\}$, $\{P_3, P_4, P_5\}$, $\{P_1, P_2, P_3, P_4\}, \dots$ are persistent sets
- $\{P_1\}, \{P_2, P_3\}, \dots$ are not persistent sets

DPOR builds persistent sets iteratively for each state



Explore the corresponding subtree

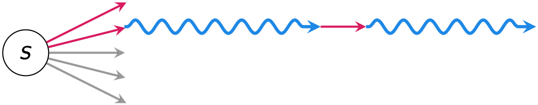
DPOR: persistent sets

Five transitions enabled in s

$P_1(\texttt{Send}(P_A))$
$P_2(\texttt{Send}(P_A))$
$P_3(\texttt{Send}(P_B))$
$P_4(\texttt{Send}(P_B))$
$P_5(\texttt{LocalOp}())$

- $\{P_1, P_2\}$, $\{P_3, P_4, P_5\}$, $\{P_1, P_2, P_3, P_4\}, \ldots$ are persistent sets
- $\{P_1\}, \{P_2, P_3\}, \ldots$ are not persistent sets

DPOR builds persistent sets iteratively for each state



If a dependent transition is found, add it to the persistent set

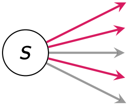
DPOR: persistent sets

Five transitions enabled in s

$P_1(\texttt{Send}(P_A))$
$P_2(\texttt{Send}(P_A))$
$P_3(\texttt{Send}(P_B))$
$P_4(\texttt{Send}(P_B))$
$P_5(\texttt{LocalOp}())$

- $\{P_1, P_2\}$, $\{P_3, P_4, P_5\}$, $\{P_1, P_2, P_3, P_4\}, \ldots$ are persistent sets
- $\{P_1\}, \{P_2, P_3\}, \ldots$ are not persistent sets

DPOR builds persistent sets iteratively for each state



Repeat until no more dependent transition not in the set is found

DPOR: sleep sets

Sleep set

For each prefix E

- *sleep*(E) contains the transitions already explored from E
- *sleep*$(E \cdot t)$ is initialized with
  $\{t' \mid t' \in$ *sleep*(E) and t' is independent with $t\}$

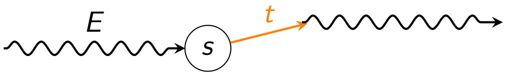
DPOR: sleep sets

Sleep set

For each prefix E

- $sleep(E)$ contains the transitions already explored from E
- $sleep(E \cdot t)$ is initialized with $\{t' \mid t' \in sleep(E) \text{ and } t' \text{ is independent with } t\}$

It is sound to never explore a transition in a sleep set:



After exploring the subtree starting with t, $sleep(E) = \{t\}$

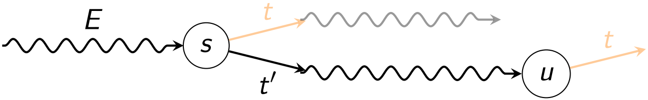
DPOR: sleep sets

Sleep set

For each prefix E

- $sleep(E)$ contains the transitions already explored from E
- $sleep(E \cdot t)$ is initialized with $\{t' \mid t' \in sleep(E) \text{ and } t' \text{ is independent with } t\}$

It is sound to never explore a transition in a sleep set:



At any time when exploring t', if t is still in the sleep set at u...

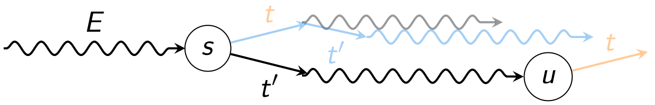
DPOR: sleep sets

Sleep set

For each prefix E

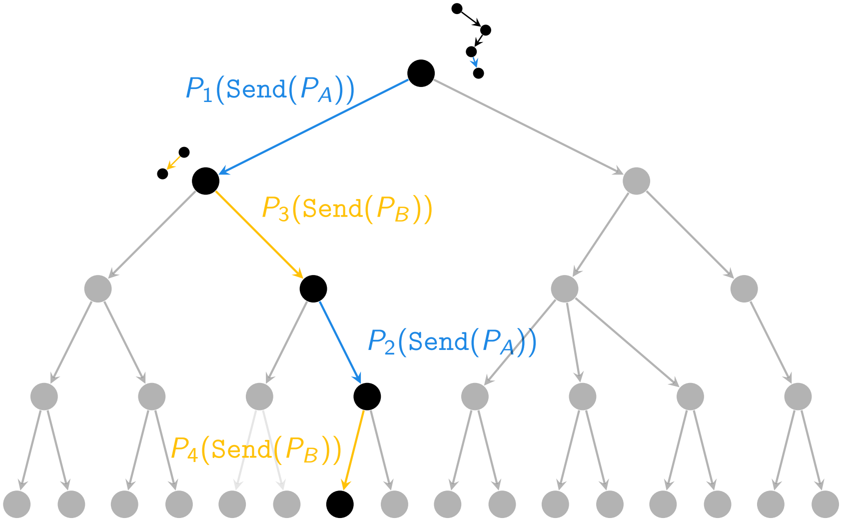
- $sleep(E)$ contains the transitions already explored from E
- $sleep(E \cdot t)$ is initialized with
  $\{t' \mid t' \in sleep(E) \text{ and } t' \text{ is independent with } t\}$

It is sound to never explore a transition in a sleep set:



for each execution starting by t from u, an equivalent has been explored

ODPOR approach [Abd'14]

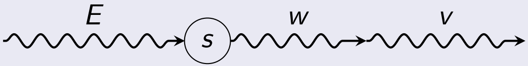


Compute initials and handle a tree of sequences instead of a single step

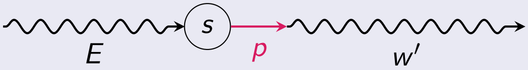
ODPOR: better sets

Weak Initials

$p \in WI_{[E]}(w)$ iff there exists v, w' such that



is equivalent to



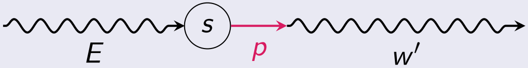
To explore w from E, we can start by any process in $WI_{[E]}(w)$

ODPOR: better sets

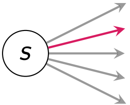
Weak Initials

$p \in WI_{[E]}(w)$ iff there exists v, w' such that

is equivalent to



To explore w from E, we can start by any process in $WI_{[E]}(w)$



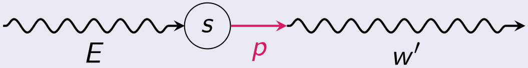
Source sets computation

ODPOR: better sets

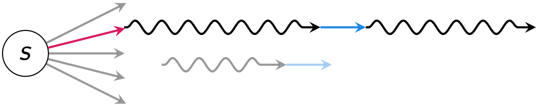
Weak Initials

$p \in WI_{[E]}(w)$ iff there exists v, w' such that

is equivalent to



To explore w from E, we can start by any process in $WI_{[E]}(w)$

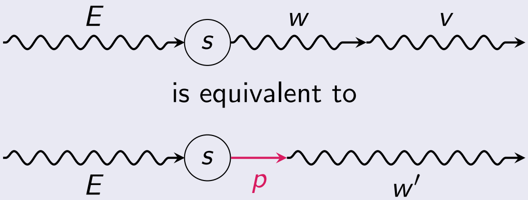


when finding a race, compute what the reversed race w looks like

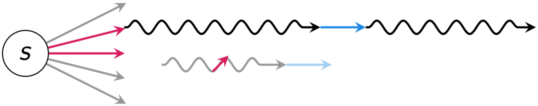
ODPOR: better sets

Weak Initials

$p \in WI_{[E]}(w)$ iff there exists v, w' such that



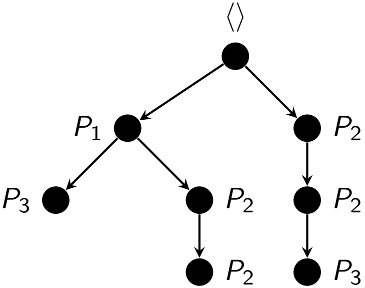
To explore w from E, we can start by any process in $WI_{[E]}(w)$



ensure some $p \in WI_{[E]}(w)$ is in the source set

ODPOR: wakeup trees

To avoid sleep set blocked executions, ODPOR stores trees

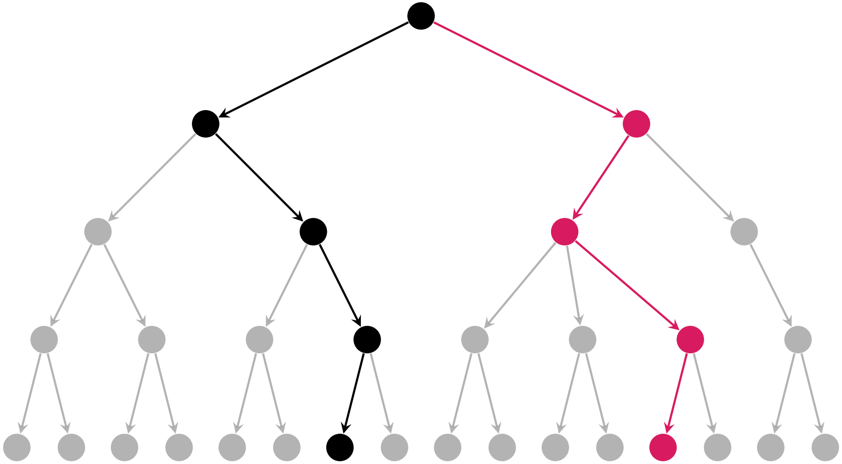


A wakeup tree containing sequences P_1P_3, $P_1P_2P_2$ and $P_2P_2P_3$

Insertion ensures that:

- the exploration of $P_1P_2P_2$ will not be blocked by $\{P_3\}$
- the exploration of $P_2P_2P_3$ will not be blocked by $\{P_1\}$

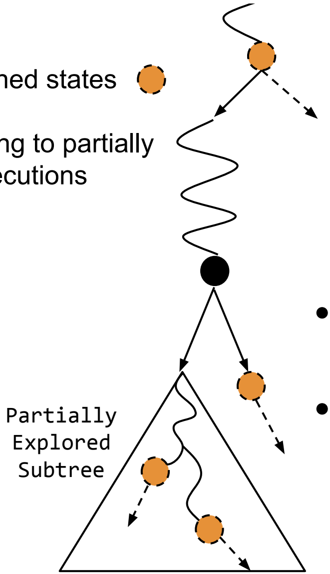
Limits of this approach



What if the only bug is far from the first guess?

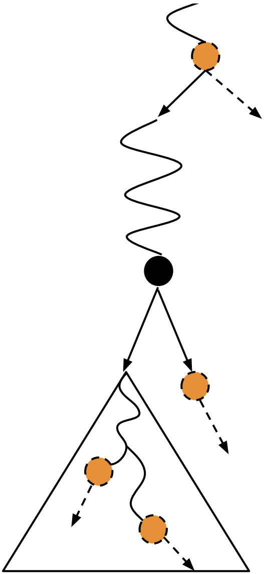
Contribution: Best First (O)DPOR

- Multiple opened states
- Corresponding to partially explored executions



- Keeps the optimality from ODPOR
- Allows more freedom in exploration order

BeFS ODPOR differences

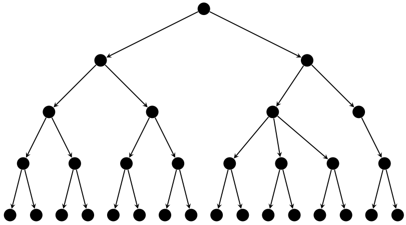


- The explored tree is saved as a wakeup tree
- Sleep sets are kept ordered and are updated at the right time
- A procedure garbages collected states when there are no longer needed

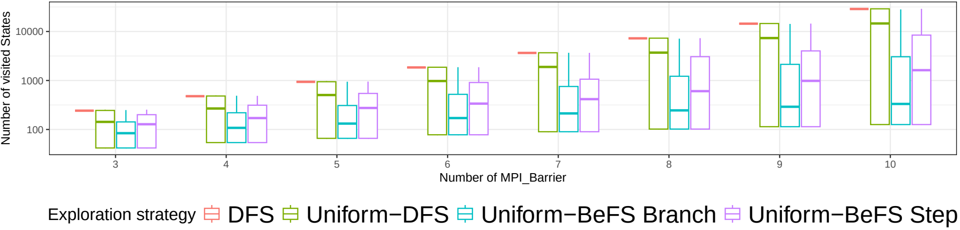
Experimental results

MPI example slightly modified

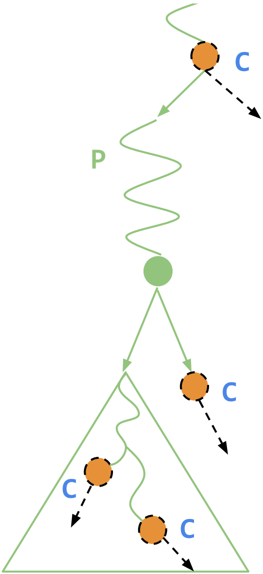
P_1	P_2	P_3
`Send(`P_3`)`	`Send(`P_3`)`	`MPI_Barrier()`
`MPI_Barrier()`	`MPI_Barrier()`	`Recv(Any)`
		`Recv(From `P_1`)`



In what order?



What's next? - Parallelized exploration



- One **producer** handling the tree
- Multiple **consumers** picking up
- Distinct explorations happening in parallel

What's next? - Exploration heuristics

Maximize dissimilarity using Fidge-Mattern vector clocks

- Each process stores a clock for each process ($VC \in \mathbb{N}^P$)
- $VC_i[i]$ updates when i does something
- $VC_i[j]$ updates when i and j synchronize over an operation
- States are abstracted as points in $(\mathbb{N}^P)^P$
- Use distance in that space to maximize dissimilarity

What's next? - Exploration heuristics

Maximize dissimilarity using Fidge-Mattern vector clocks

- Each process stores a clock for each process ($VC \in \mathbb{N}^P$)
- $VC_i[i]$ updates when i does something
- $VC_i[j]$ updates when i and j synchronize over an operation
- States are abstracted as points in $(\mathbb{N}^P)^P$
- Use distance in that space to maximize dissimilarity

Using incremental dependencies

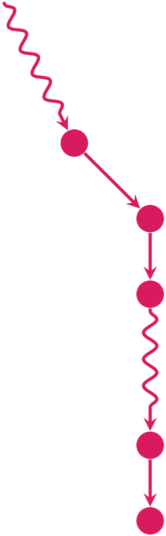
- Most bugs only concern 2 or 3 actors (e.g. A and B)
- Start by over-reducing the system as if only A and B were dependent
- Slowly increase the dependencies and repeat

Working on explainability: Why?

```
**************************
*** DEADLOCK DETECTED ***
**************************
] 1 actor is still active, awaiting something. Here is its status:
]  - pid 3 (2@node-10.simgrid.org) simcall CommWait(comm_id:20 src:-1 dst:3 mbox:SMPI-3(id:3))
Counter-example execution trace:
  Actor 2 in :0:() ==> simcall: iSend(mbox=3)
  Actor 1 in :0:() ==> simcall: iSend(mbox=3)
  Actor 1 in :0:() ==> simcall: iRecv(mbox=4)
  Actor 1 in :0:() ==> simcall: iRecv(mbox=4)
  Actor 2 in :0:() ==> simcall: iSend(mbox=4)
  Actor 1 in :0:() ==> simcall: WaitComm(from 2 to 1, mbox=4, no timeout)
  Actor 2 in :0:() ==> simcall: iRecv(mbox=5)
  Actor 3 in :0:() ==> simcall: iSend(mbox=4)
  Actor 1 in :0:() ==> simcall: WaitComm(from 3 to 1, mbox=4, no timeout)
  Actor 1 in :0:() ==> simcall: iSend(mbox=5)
  Actor 1 in :0:() ==> simcall: iSend(mbox=3)
  Actor 1 in :0:() ==> simcall: iRecv(mbox=4)
  Actor 1 in :0:() ==> simcall: iRecv(mbox=4)
  Actor 2 in :0:() ==> simcall: WaitComm(from 1 to 2, mbox=5, no timeout)
  Actor 2 in :0:() ==> simcall: iSend(mbox=4)
  Actor 1 in :0:() ==> simcall: WaitComm(from 2 to 1, mbox=4, no timeout)
  Actor 2 in :0:() ==> simcall: iRecv(mbox=5)
  Actor 3 in :0:() ==> simcall: iRecv(mbox=3)
  Actor 3 in :0:() ==> simcall: WaitComm(from 1 to 3, mbox=3, no timeout)
  Actor 3 in :0:() ==> simcall: iSend(mbox=4)
  Actor 1 in :0:() ==> simcall: WaitComm(from 3 to 1, mbox=4, no timeout)
  Actor 1 in :0:() ==> simcall: iSend(mbox=5)
  Actor 1 in :0:() ==> simcall: iSend(mbox=3)
  Actor 2 in :0:() ==> simcall: WaitComm(from 1 to 2, mbox=5, no timeout)
  Actor 3 in :0:() ==> simcall: iRecv(mbox=3)
  Actor 3 in :0:() ==> simcall: WaitComm(from 1 to 3, mbox=3, no timeout)
  Actor 3 in :0:() ==> simcall: iRecv(mbox=3)
  Actor 3 in :0:() ==> simcall: WaitComm(from 2 to 3, mbox=3, no timeout)
  Actor 3 in :0:() ==> simcall: iRecv(mbox=3)
```

Mc SimGrid output on a simple example with only two `MPI_Barrier()`.

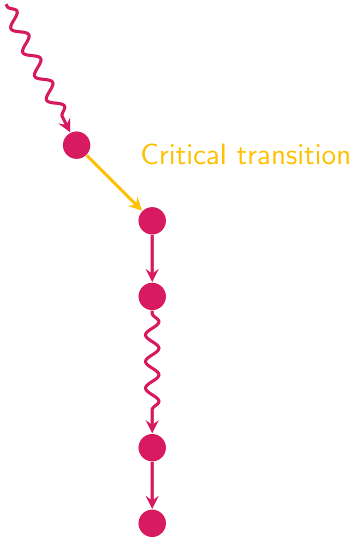
The critical transition



Critical transition

Let E be an incorrect execution,

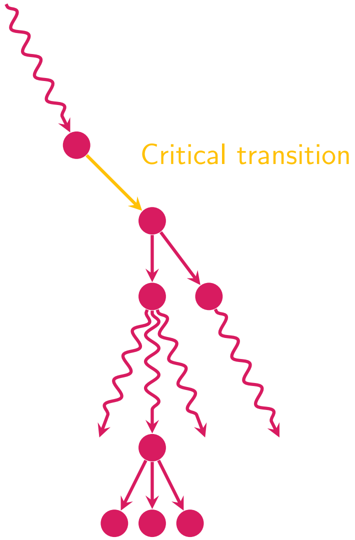
The critical transition



Critical transition

Let E be an incorrect execution,
the **critical transition** is the unique
$t = (s, a, s')$ $\in E$ s.t.

The critical transition

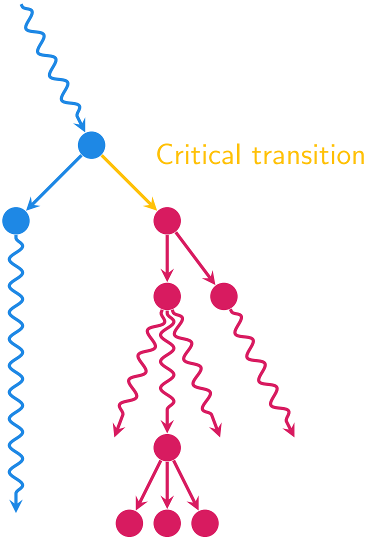


Critical transition

Let E be an incorrect execution,
the **critical transition** is the unique
$t = (s, a, s')$ $\in E$ s.t.

- every execution from s' is incorrect

The critical transition



Critical transition

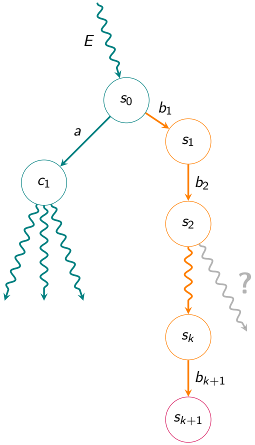
Let E be an incorrect execution,
the **critical transition** is the unique
$t = (s, a, s') \in E$ s.t.

- every execution from s' is incorrect
- there exists a correct execution from s

Critical transition: how to compute?

Use reduction and take a decision for the non-explored transitions

- s_{k+1} violates the property
- c_1 is the root of a correct subtree
- Hence, the critical transition is in $\{b_1, \ldots, b_{k+1}\}$



Critical transition: what are we missing?

A two process deadlock

P_1	P_2
$\texttt{Lock}(a)$	$\texttt{Lock}(b)$
$\texttt{Lock}(b)$	$\texttt{Lock}(a)$

- Executions starting by $P_1 P_2$ or $P_2 P_1$ **will** deadlock
- Critical transition is the last executed of $P_1 : \texttt{Lock}(a)$ and $P_2 : \texttt{Lock}(b)$
- Possible to retrieve both P_1 and P_2 locks

Critical transition: what are we missing?

A *four* process deadlock

P_1	P_2	P_3	P_4
$\texttt{Lock}(a)$	$\texttt{Lock}(b)$	$\texttt{Lock}(c)$	$\texttt{Lock}(d)$
$\texttt{Lock}(b)$	$\texttt{Lock}(a)$	$\texttt{Lock}(d)$	$\texttt{Lock}(c)$

- Executions starting by ... (24 combinations) **will** deadlock
- Critical transition is the last executed of ... (one of the processes' first action)
- No links between a/b and c/d deadlocks

Conclusion

Contributions

- New reduction algorithms allowing arbitrary search
- Defining and computing critical transitions
- Code integrated within McSimGrid

Future work

- Parallelize the implementation of BeFS ODPOR
- Develop a good benchmark to explore heuristics
- Simplify counter examples using critical sections
- Observe memory access and detect data races with McSimGrid

Time and memory performances

Benchmark Name	Traces	DPU (UDPOR) Time	DPU (UDPOR) Mem	Nidhugg (ODPOR) Time	Nidhugg (ODPOR) Mem	McSG(BeFS ODPOR) Time	McSG(BeFS ODPOR) Mem
DISP(5,3)	1482	0.629	55M	6.314	65M	2.080	54M
DISP(5,4)	15282	6.285	135M	65.034	65M	15.245	460M
DISP(5,5)	151032	203.785	973M	TO	65M	154.689	4387M
DISP(5,6)		ERR	1016M	TO	65M	TO	17219M
MPAT(5)	3840	1.860	80M	1.203	64M	3.927	154M
MPAT(6)	46080	51.283	420M	16.273	64M	51.426	1853M
MPAT(7)	645120	TO	1553M	255.109	64M	TO	19609M
MPAT(8)		TO	1603M	TO	64M	TO	23999M
MPC(2,5)	60	0.273	51M	1.038	65M	0.067	12M
MPC(3,5)	2958	0.937	61M	37.662	65M	2.510	81M
MPC(4,5)	313683	ERR	63M	TO	65M	308.723	6684M
MPC(5,5)		TO	1344M	TO	65M	TO	23495M
PI(5)	120	0.301	43M	ERR	66M	0.082	11M
PI(6)	720	0.468	47M	ERR	66M	0.441	19M
PI(7)	5040	1.950	66M	ERR	66M	3.201	77M
PI(8)	40320	28.748	273M	ERR	66M	26.796	573M
PI(9)	362880	TO	1128M	ERR	65M	291.884	5291M
POKE(7)	2440	1.247	84M	44.736	65M	3.057	118M
POKE(8)	3700	1.934	99M	146.232	65M	4.913	193M
POKE(9)	5332	2.913	124M	458.337	65M	7.653	302M
POKE(10)	7384	4.479	152M	TO	64M	11.310	446M
POKE(11)	9904	6.674	193M	TO	65M	16.247	649M
POKE(12)	12940	9.969	242M	TO	65M	22.676	910M
POKE(13)	16540	14.506	310M	ERR	64M	30.774	1252M

More results