Efficient state-space exploration for asynchronous distributed programs

The Anh Pham

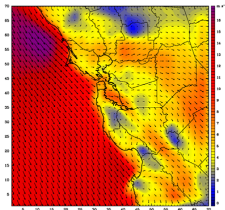
Supervisors: Thierry Jéron, Martin Quinson

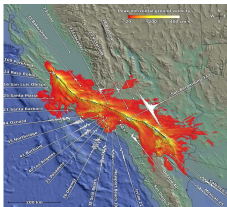
Jury :

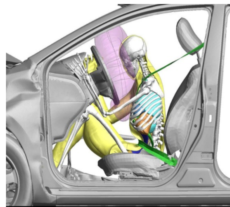
Radu Mateescu	Research Director, Inria Grenoble – Rhône-Alpes (R).
Laure Petrucci	Professor, IUT de Villetaneuse, Université Paris 13 (R).
Stefan Leue	Professor, Universität Konstanz (Germany).
Stephan Merz	Research Director, Inria Nancy – Grand Est.
François Taïani	Professor, ESIR, Université de Rennes 1.
Thierry Jéron	Research Director, Inria Rennes – Bretagne Atlantique (S).
Martin Quinson	Professeur, ENS Rennes (S).

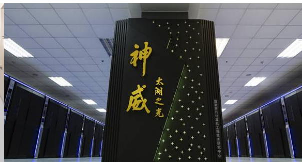
6th December 2019

Distributed programs









- Distributed applications are widespread in the HPC community,
- MPI libraries (e.g. MPICH) are widely used to develop HPC applications,
- Distributed applications are hard to design.

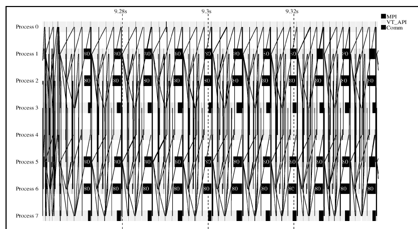
Why is it difficult to design distributed programs

- Concurrency: many processes running in parallel
  - How to split an algorithm into several operations executed concurrently?
  - How to synchronize processes effectively?
- Data distribution: data locality
  - How to efficiently write and store data?
  - How to efficiently process (e.g. communicate, combine, visualize) data?
- Nondeterminism: many execution scenarios $\rightarrow$ hard to avoid unwanted scenarios.

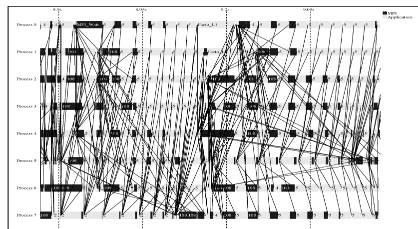
Classical approach to write distributed programs

Rigid communication patterns

- Avoid complex synchronization scenarios.
- Scale poorly $\rightarrow$ a strong need for dynamic communication patterns.



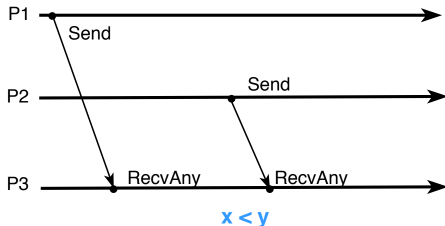
Rigid communication patterns

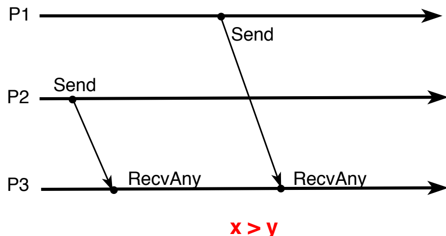


Dynamic communication patterns

Testing distributed programs

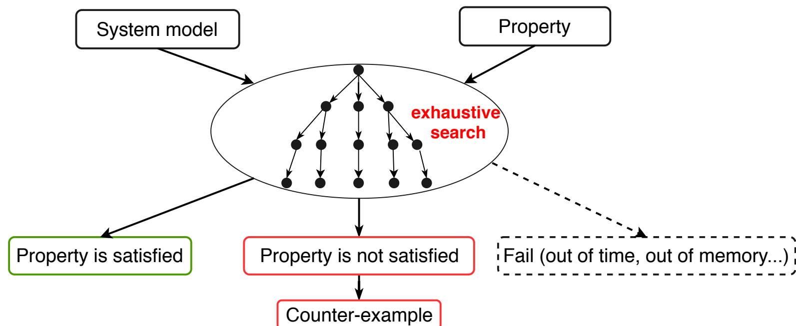
```
P1 ( ) {
    Send (1, P3);
}

P2 ( ) {
    Send (2, P3);
}

P3 ( ) {
    RecvAny(&x, *);

    RecvAny(&y, *);

    assert(x < y);
}
```





- Testing is incomplete $\Rightarrow$ formal methods can help.

Model checking



- Exhaustive exploration of the state space
- Check if the property is true at every state (for safety properties)
- A counterexample denotes an execution leading to the bug.

State space explosion because of concurrency

- Exhaustive search of all possible states ⇒ state space explosion.
- Sources of state space explosion: concurrency, non-determinism, unbounded data...
- Number of states can grow exponentially with respect to the number of processes.

Example:

```
Master(){
 for(i:=1; i<=nbWorker; i++)
    c_i= Isend(to Worker_i);

 for(i:=1; i<=nbWorker; i++)
    wait(c_i);

 for(i:=1; i<=nbWorker; i++){
    c'_i= Irecv(from Worker_i);
    wait(c'_i); }
}
```

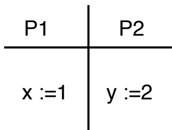
```
Worker() {

   c1= Irecv(from Master);
   wait(c1);
   Computation
   c2= Isend(to Master);
   wait(c2);

}
```

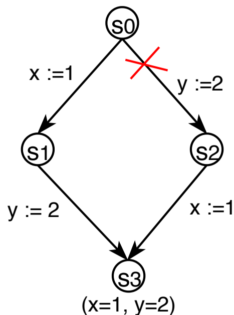
nbWorker = 5: more than one million of states.

Main concern

Can we remain exhaustive (preserve properties) but partially explore the state space?

- **Partial order reduction**[a] (POR) efficiently mitigates the state space explosion problem.



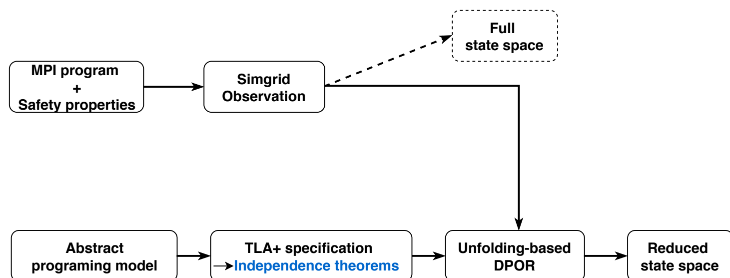


- Most studies of POR focused mostly on shared memory programs.
- Applying POR to distributed programs remains challenging.

[a] Patrice Godefroid, Partial-Order Methods for the Verification of Concurrent Systems, 1996

The main goal of the thesis

Efficiently adapting Unfolding-based Dynamic partial order reduction to verify MPI programs.



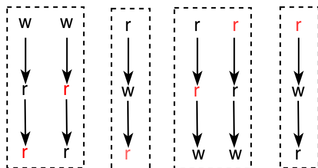
1. Context
2. Partial Order reduction
3. Abstract programming model of asynchronous distributed programs
4. Adapting UDPOR
5. Evaluation and Conclusion

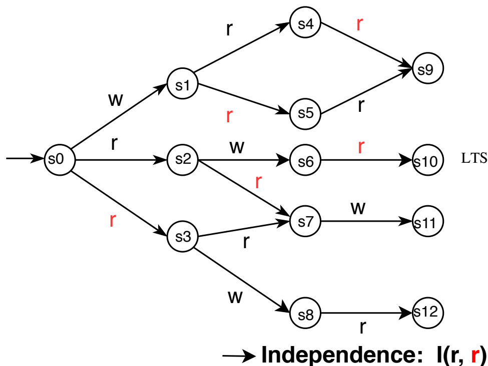
Interleaving semantics & POR

Model: Labelled transition system (LTS)

Independence: two actions a and b are independent if they commute:

1. Executing one action does not enable nor disable the other one,
2. Their execution order does not change the overall result.



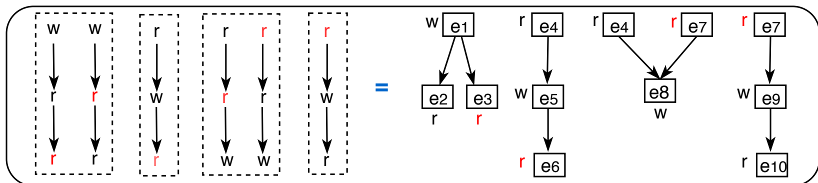


- Mazurkiewicz trace = an equivalence class of executions.
- POR explores **at least one execution per** Mazurkiewicz trace.

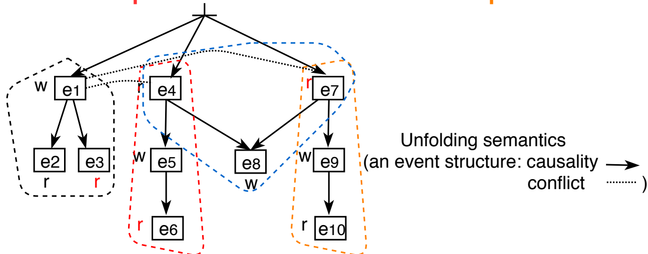
Unfolding semantics[a]

w	r	r
write(x)	read(x)	read(x)

Independence: I(r, r)



Mazurkiewicz trace = equivalence class of executions = partial order

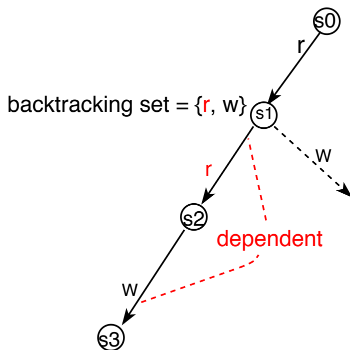


[a] César Rodríguez et al.,Unfolding-based Partial Order Reduction, CONCUR 2015

Dynamic partial order reduction (*DPOR*)[a]

W	r	r
write(x)	read(x)	read(x)

Independence: I(r, r)

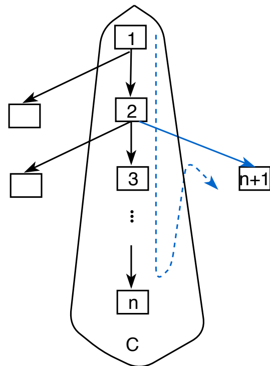


- Dynamicity: computing independence at run-time to build backtracking sets for states.
- **Optimal DPOR**: exploring **only one execution** per Mazurkiewicz trace.

[a] Flanagan and Godefroid, Dynamic partial-order reduction for model checking software, POPL 2005

Unfolding-based Dynamic partial order reduction (*UDPOR*)[a]

1 **Procedure** ***Explore***(C, D, A)
2 Compute extensions of C ($ex(C)$)
3 Add all events in $ex(C)$ to U
4 **if** $en(C) \subseteq D$ **then**
5 Return
6 **if** $(A = \emptyset)$ **then**
7 choose e from $en(C) \setminus D$
8 **else**
9 choose e from $A \cap en(C)$
10 **Explore**$(C \cup \{e\}, D, A \setminus \{e\})$
11 **if** $\exists J \in Alt(C, D \cup \{e\})$ **then**
12 **Explore**$(C, D \cup \{e\}, J \setminus C)$
13 $U := U \cap Q_{C,D}$

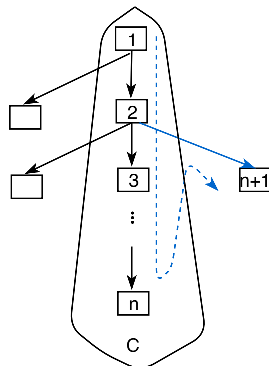


- Combining strengths of unfolding semantics and DPOR.
- Visiting every configuration (partially ordered of events)

[a] César Rodríguez et al.,Unfolding-based Partial Order Reduction, CONCUR 2015

Alternatives

```
1  Procedure Explore(C, D, A)
2  |  Compute extensions of C (ex(C))
3  |  Add all events in ex(C) to U
4  |  if en(C) ⊆ D then
5  |  |  Return
6  |  if (A = ∅) then
7  |  |  choose e from en(C) \ D
8  |  else
9  |  |  choose e from A ∩ en(C)
10 |  Explore(C ∪ {e}, D, A \ {e})
11 |  if ∃J ∈ Alt(C, D ∪ {e}) then
12 |  |  Explore(C, D ∪ {e}, J \ C)
13 |  U := U ∩ Q_{C,D}
```

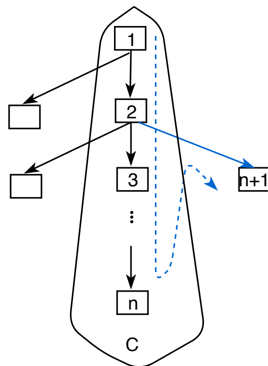


- Each alternative (roughly) corresponds to a backtracking point.
- Computing alternatives is an NP-Complete problem in optimal DPORs.
- Quasi-Optimal *POR*[a]: tuning between an optimal or a quasi-optimal algorithm (may be more efficient) by using a constant *k* (k-partial alternative)

[a] Huyen T.T Nguyen et al.,Quasi-Optimal Partial Order Reduction, CAV 2018.

Extensions

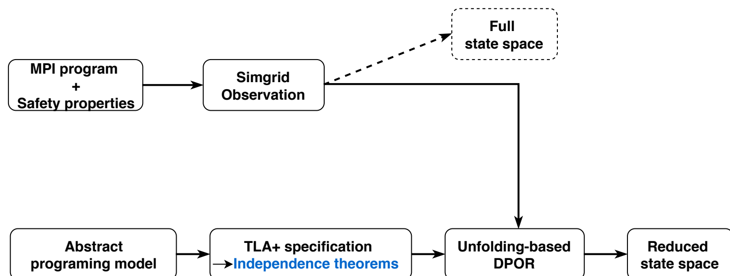
```
1  Procedure Explore(C, D, A)
2      Compute extensions of C (ex(C))
3      Add all events in ex(C) to U
4      if en(C) ⊆ D then
5          Return
6      if (A = ∅) then
7          choose e from en(C) \ D
8      else
9          choose e from A ∩ en(C)
10     Explore(C ∪ {e}, D, A \ {e})
11     if ∃J ∈ Alt(C, D ∪ {e}) then
12         Explore(C, D ∪ {e}, J \ C)
13     U := U ∩ Q_{C,D}
```



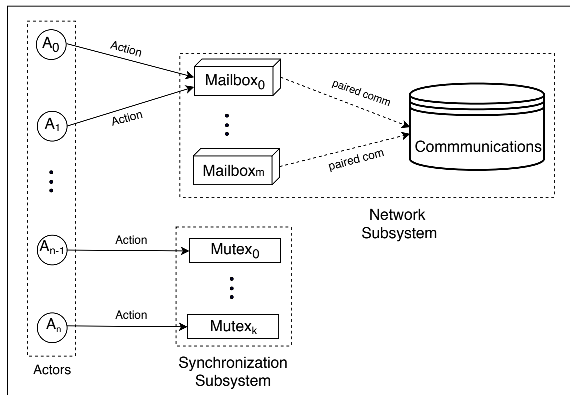
- A configuration (partially ordered of events) = an equivalence class of executions.
- Extensions: direct states reachable from some states of these executions.
- Computing extensions may be costly (e.g. NP-complete for Petri Nets) $\rightarrow$ should be computed efficiently.

Agenda

1. Context
2. Partial Order reduction
3. Abstract programming model of asynchronous distributed programs
4. Adapting UDPOR
5. Evaluation and Conclusion



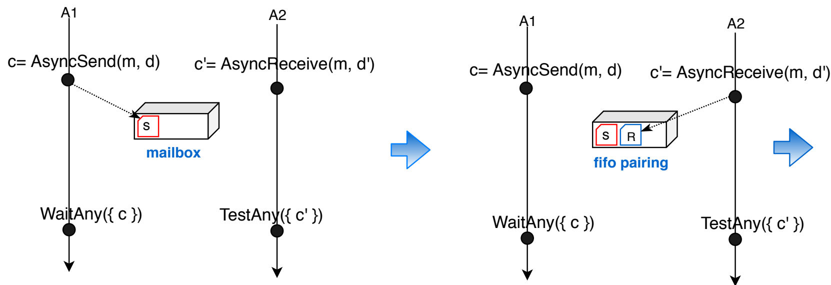
Abstract programming model

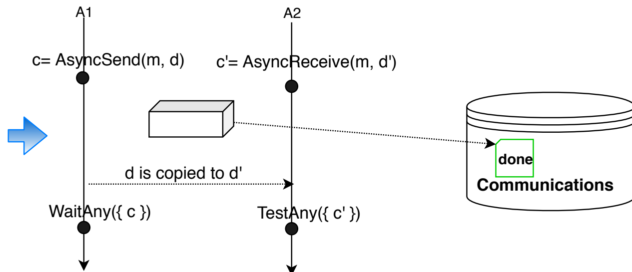


Actions

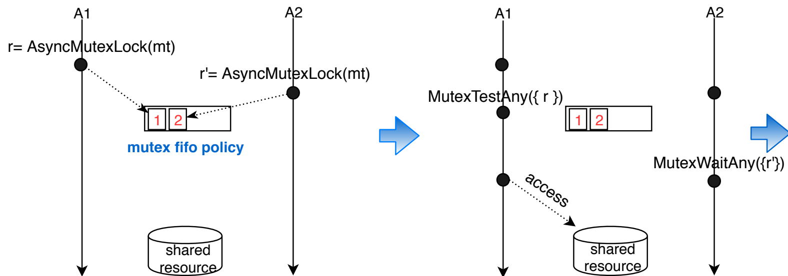
- Communication: *AsyncSend*, *AsyncReceive*, *TestAny*, *WaitAny*
- Synchronization: *AsyncMutexLock*, *MutexUnlock*, *MutexTestAny*, *MutexWaitAny*
- Local computation: *LocalComp*

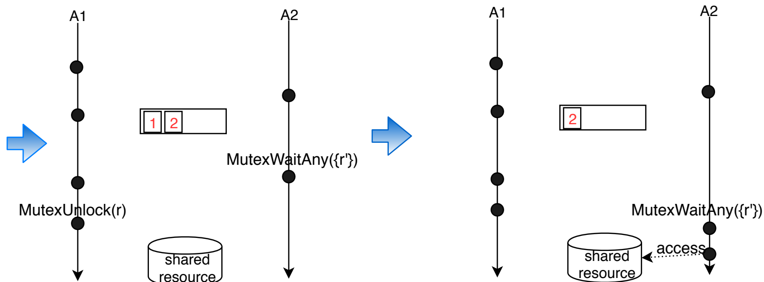
Communication actions





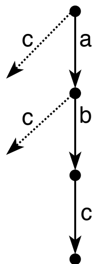
Synchronization actions





Persistence, a key property for efficient UDPOR

An enabled action is *persistent* if it cannot be disabled by performing other actions.



Lemma: All actions are persistent in our model

- Contrary to usual models of mutex, where locks (= *AsyncMutexLock* + *MutexWaitAny*) are atomic
- Persistence is essential in the efficiency of UDPOR.

TLA+ specification of the programming model

Model specification (in TLA+)

```
AsyncReceive(aId, mbId, data_addr, comm_addr) ==
  /\ aId \in ActorsIds
  /\ mbId\in MailboxesIds
  /\ data_addr \in Addresses
  /\ comm_addr \in Addresses
  /\ pc[aId] \in ReceiveIns
  /\ \/ /\ \/ Len(Mailboxes[mbId]) = 0
           \/ /\ Len(Mailboxes[mbId]) > 0
              /\ Head(Mailboxes[mbId]).status = "receive"
        /\ LET comm ==
                 [id |-> commId,
                  status |-> "receive",
                  src |-> NoActor,
                  dst |-> aId,
                  data_src |-> NoAddr,
                  data_dst |-> data_addr]
           IN
             /\ Mailboxes' = [Mailboxes EXCEPT ![mbId] = Append(Mailboxes[mbId],
                                                                  comm)]
             /\ Memory' = [Memory EXCEPT ![aId][comm_addr] = comm.id]
             /\ UNCHANGED <<Communications>>
             /\ commId' = commId+1
```

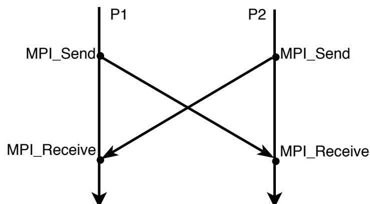
Independence theorems expressed in TLA+, used in UDPOR

Example : An *AsyncSend* action and an *AsyncReceive* action are independent if they are performed by different actors.

```
THEOREM \forall a1, a2 \in ActorsIds, mbId1, mbId2 \in MailboxesIds, data1, data2,
comm1, comm2 \in Addresses: a1 /= a2
        => I(AsyncSend(a1, mbId1, data1, comm1), AsyncReceive(a2, mbId2, data2, comm2))
```

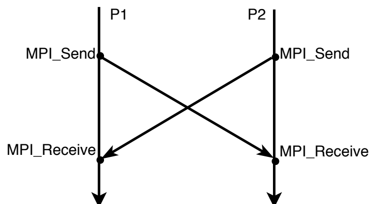
Example of an interesting property: deadlock

Deadlock or deadlock free?

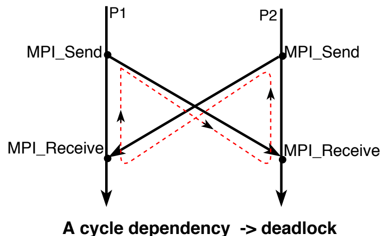


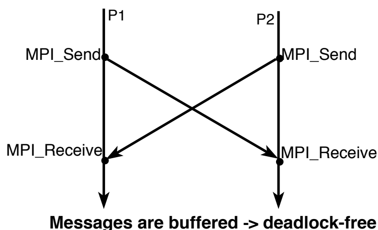
Example of an interesting property: deadlock

Deadlock or deadlock free?



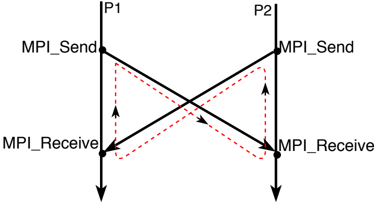
Deadlock depends on zero-buffering or infinite-buffering.

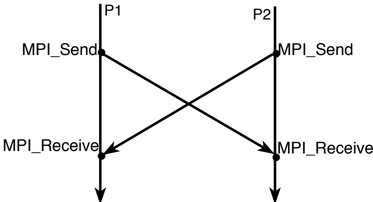




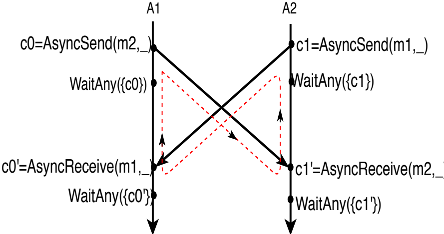
Encoding MPI programs

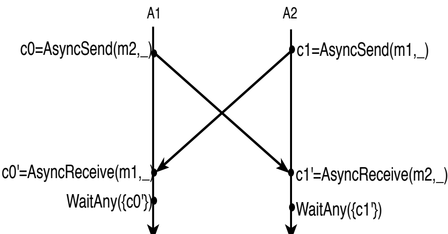
Deadlock depends on zero-buffering or infinite-buffering.





Encoding





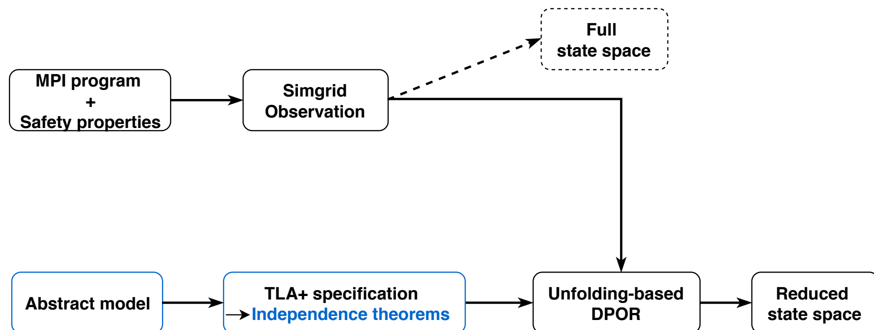
Encoding MPI functions

MPI functions	Infinite buffering	Zero buffering
MPI_Send	*AsyncSend*	*AsyncSend + WaitAny*
MPI_Isend		*AsyncSend*
MPI_Recv	*AsyncReceive + WaitAny*	
MPI_Irecv	*AsyncReceive*	
MPI_Test	*TestAny*	
MPI_Testany		
MPI_Wait	*WaitAny*	
MPI_Waitany		
MPI_Win_lock	*AsyncMutexLock + MutexWaitAny*	
MPI_Win_unlock	*MutexUnlock*	

160 MPI functions are simulated by using this model in SimGrid.

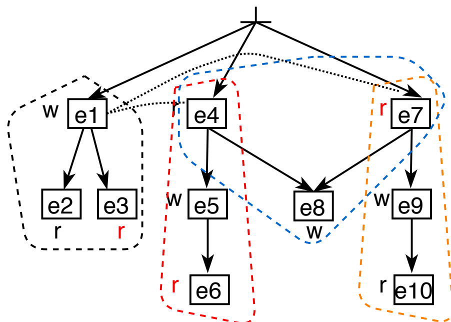
Agenda

1. Context
2. Partial Order reduction
3. Abstract programming model of asynchronous distributed programs
4. Adapting UDPOR
5. Evaluation and Conclusion



How to compute extensions of a configuration efficiently?

Configuration



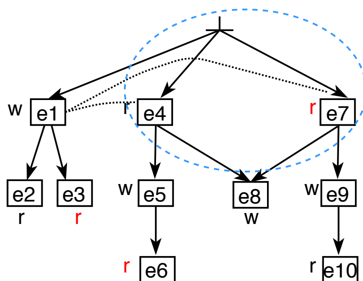
- C = set of events, conflict free and causally closed (represents an equivalence class of executions).
- C can be identified by its **maximal events set**:
  $maximalEvt(C)$= { events in C that are not causal predecessors of any other event in C }.
  Example: $maximalEvt(\{e_4, e_7, e_8\}) = \{e_8\}$; $maximalEvt(\{e_1, e_2, e_3\}) = \{e_2, e_3\}$

Extensions

- $ex(C) = \{$ events outside C whose causal predecessors are all in C $\}$.

  Example: $ex(\{e_4, e_7\}) = \{e_1, e_5, e_8, e_9\}$

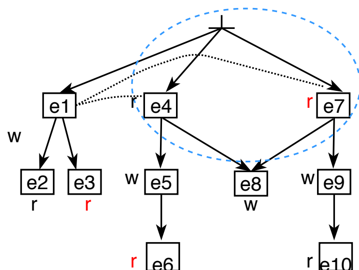
- C represents an equivalence class of executions
  $\rightarrow ex(C) \simeq \{$ states that are directly reachable from some states of these executions $\}$.



Computing ex(C) (naive and expensive method)

e = $\langle a, H \rangle \in ex(C) \iff$

- H is a configuration, H⊆C
- a is enabled at state(H)
- $\forall e' \in$ maximalEvt(H) : $D(a, \lambda(e'))$



Combining every subset of C with every action

Example: ex($\{e_4, e_7\}$) =

Event set	enabled actions	events
$\emptyset$	w, r, r	e_1, e_4, e_7
$\{e_4\}$	w, r	e_5
$\{e_7\}$	w, r	e_9
$\{e_4, e_7\}$	w	e_8

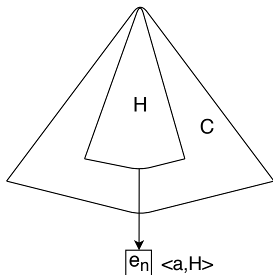
$\Rightarrow$ Exponential number of subsets.

Computing extensions: proposed method

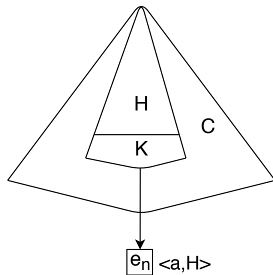
$e = \langle a, H \rangle \in ex(C) \iff$

$$\begin{cases} \text{H is a configuration, } H \subseteq C \\ a \text{ is enabled at state(H)} \\ \forall e' \in \text{maximalEvt(H)} : D(a, \lambda(e')) \end{cases}$$

a depends on the actions of very few and easily identifiable events in C.



Checking all subsets of C
= exponential time



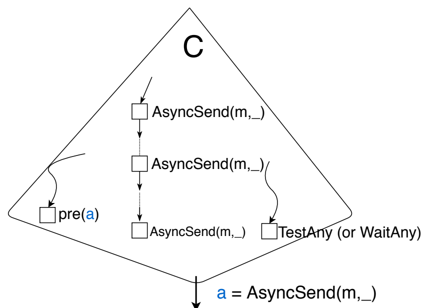
Computing all sets K,
|K| <= 3 (thanks to the persistence property)

Possible values of K according to the type of action a

Type of action	Description of K
AsyncSend	K $\subseteq$ { $preEvt(a, C)$, *AsyncSend*, *TestAny* }
AsyncReceive	K $\subseteq$ { $preEvt(a, C)$, *AsyncReceive*, *TestAny* }
TestAny	K $\subseteq$ { $preEvt(a, C)$, *AsyncSend* (or *AsyncReceive*) }
WaitAny	
AsyncMutexLock	K $\subseteq$ { $preEvt(a, C)$, *AsyncMutexLock* }
MutexUnlock	K $\subseteq$ { $preEvt(a, C)$, *MutexTestAny* }
MutexTestAny	K $\subseteq$ { $preEvt(a, C)$, *MutexUnlock* }
MutexWaitAny	
LocalComp	K $\subseteq$ { $preEvt(a, C)$}

Computing extensions labelled by an *AsyncSend*

If a is $AsyncSend(m, _)$ $\Rightarrow$ resources of dependency: $pre(a)$, $AsyncSend(m, _)$, $TestAny/WaitAny$.



- $pre(a)$ is unique.
- All $AsyncSend(m, _)$ events are causally related.
- a depends on only one *TestAny* in a configuration.
- a always happens after *WaitAny* if they are dependent.

$\Rightarrow K \subseteq \{$ preEvt(a,C), $AsyncSend(m, _)$, *TestAny* $\}$

Computing extensions labelled by an *AsyncSend*

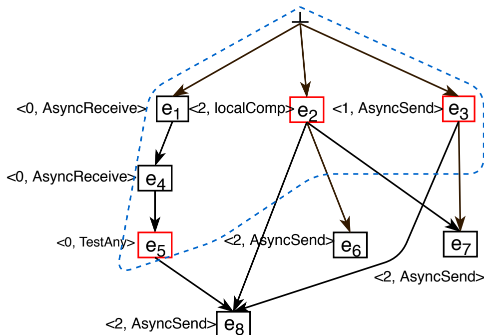
e = $\langle a, H\rangle \in ex(C) \iff$

- H is a configuration, H⊆C
- a is enabled at state(H)
- $\forall e' \in$ maximalEvt(H) : $D(a, \lambda(e'))$

$K \subseteq$ { preEvt(a,C), *AsyncSend*, *TestAny* }

Example:

Actor0	c0 = AsyncReceive(m, _) c0' = AsyncReceive(m, _) TestAny({ c0' })
Actor1	c1 = AsyncSend(m, _)
Actor2	localComp c2 = AsyncSend(m, _)



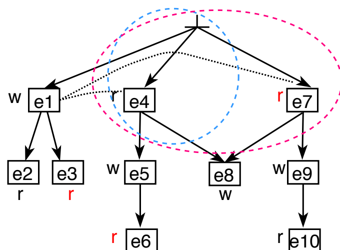
$K_1 = \{e_2\}$; $K_2 = \{e_2, e_3\}$; $K_3 = \{e_2, e_3, e_5\}$

Computing extensions incrementally

UDPOR is recursive $\rightarrow$ Recomputation of many events

1 **Procedure *Explore***(C, D, A)
2 Compute extensions of C $(ex(C))$
3 Add all events in $ex(C)$ to U
4 **if** $en(C) \subseteq D$ **then**
5 Return
6 **if** $(A = \emptyset)$ **then**
7 choose e from $en(C) \setminus D$
8 **else**
9 choose e from $A \cap en(C)$
10 **Explore**$(C \cup \{e\}, D, A \setminus \{e\})$
11 **if** $\exists J \in Alt(C, D \cup \{e\})$ **then**
12 **Explore**$(C, D \cup \{e\}, J \setminus C)$
13 $U := U \cap Q_{C,D}$

Example:



ex({ e4 }) = {e1, e5, e7}

ex({e4, e7}) = {e1, e5, e8, e9}

Eliminating redundant computations thanks to the persistence property

if $C' = C \cup \{e\}$ then $ex(C') = (ex(C) \cup \bigcup_{a \in enab(C')} \{< a, H >\}) \setminus \{e\}$

Evaluation

Benchmarks	#P	Deadlock	Exhaustive search		UDPOR	
			#Traces	Time(s)	#Traces	Time(s)
wait-deadlock	2	yes	2	<0.01	1	<0.01
send-recv-ok	2	no	24	0.03	1	<0.01
sendrecv-deadlock	3	yes	105	0.06	1	0.01
complex-deadlock	3	yes	36	0.03	1	<0.01
waitall-deadlock	3	yes	1458	1.2	1	<0.01
no-error-wait-any-src	3	no	21	0.02	1	0.01
any-src-waitall-deadlock	3	no	105	0.05	1	0.01
any-src-can-deadlock3	3	yes	999	0.65	2	0.03
DTG	5	yes	-	TO	2	0.07
RMQ-receiving	4	no	20064	8.15	6	0.2
	5	no	-	TO	24	2.52
	6	no	-	TO	120	47
Master-worker	3	no	1356444	> 17 (m)	2	0.2
	4	no	-	TO	6	2.5
	5	no	-	TO	24	60

TO: timeout after 30 minutes;

Variations on k (k-partial alternative)

Benchmark	k	run time	number of traces
RMQ-receiving #P = 5	7	2.5	24
	4	2.3	24
	3	2	25
	2	TO	> 9000
RMQ-receiving #P = 6	11	47	120
	5	34	120
	4	28	121
	3	TO	> 3000
Master-worker #P = 5	7	60	24
	5	57	24
	4	51	24
	3	TO	> 450

TO: time out after 10 minutes

$\Rightarrow$ UDPOR can still be optimal with a low value of k;
or it can have redundant explorations, but the run time decreases.

Conclusion

Efficient state-space exploration for asynchronous distributed programs

- An abstract model of asynchronous distributed programs
- Formal specification of the programming model in TLA+
  - extraction of the independence relation, used in UDPOR,
  - identification of the persistence property.
- Computing extensions in polynomial time and incrementally.

- The Anh Pham, Thierry Jéron, Martin Quinson, Verifying MPI applications with SimGridMC, CORRECTNESS@SC 2017.
- The Anh Pham, Thierry Jéron, Martin Quinson, Unfolding-Based Dynamic Partial Order Reduction of asynchronous distributed programs, FORTE 2019.

Future work

Integrating UDPOR in the SimGrid simulator

- verifying large and more complicated MPI programs,
- comparing UDPOR with state of the art tools.

Improving the performance of UDPOR

- Refining the independence relation: the more precise, the less Mazurkiewicz traces exist,
- Parallelization/distribution of UDPOR.

Checking liveness property