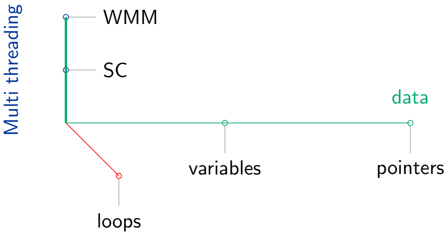
Verifying Multithreaded Software with Impact

Björn Wachter, Daniel Kroening and Joël Ouaknine

University of Oxford

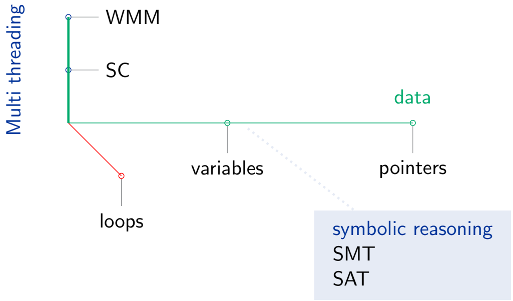
Intro

- Multi-threading
  - C/C++ with POSIX/WIN 32 threads
  - event processing, device drivers, web servers, databases, ...
  - coming to embedded systems
- Verification Challenges



Intro

- Multi-threading
  - C/C++ with POSIX/WIN 32 threads
  - event processing, device drivers, web servers, databases, ...
  - coming to embedded systems
- Verification Challenges



Multi threading

WMM

SC

data

variables

pointers

loops

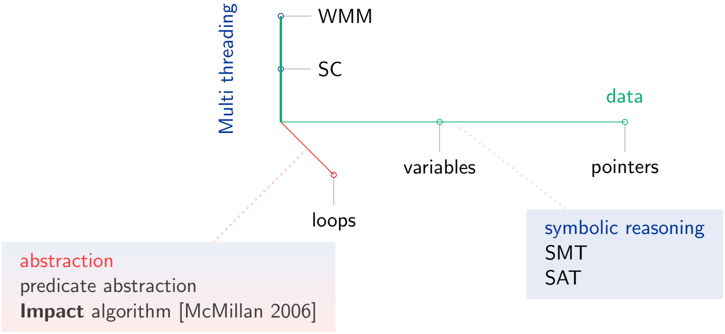
symbolic reasoning

SMT

SAT

Intro

- Multi-threading
  - C/C++ with POSIX/WIN 32 threads
  - event processing, device drivers, web servers, databases, ...
  - coming to embedded systems
- Verification Challenges

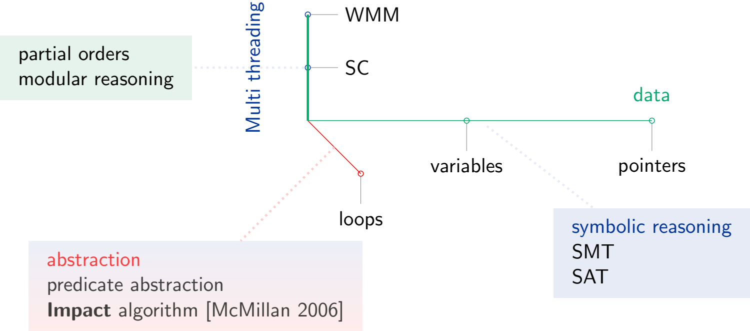


abstraction
predicate abstraction
Impact algorithm [McMillan 2006]

symbolic reasoning
SMT
SAT

Intro

- Multi-threading
  - C/C++ with POSIX/WIN 32 threads
  - event processing, device drivers, web servers, databases, ...
  - coming to embedded systems
- Verification Challenges



Multi threading

WMM

SC

partial orders
modular reasoning

data

variables

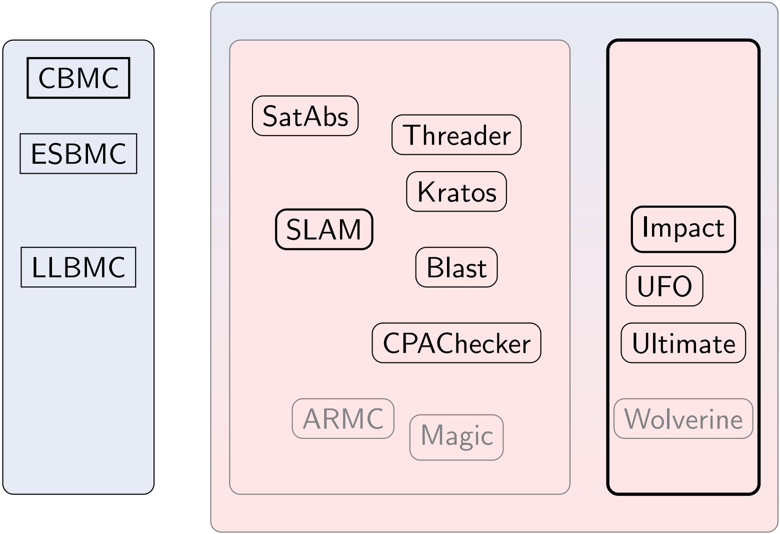
pointers

loops

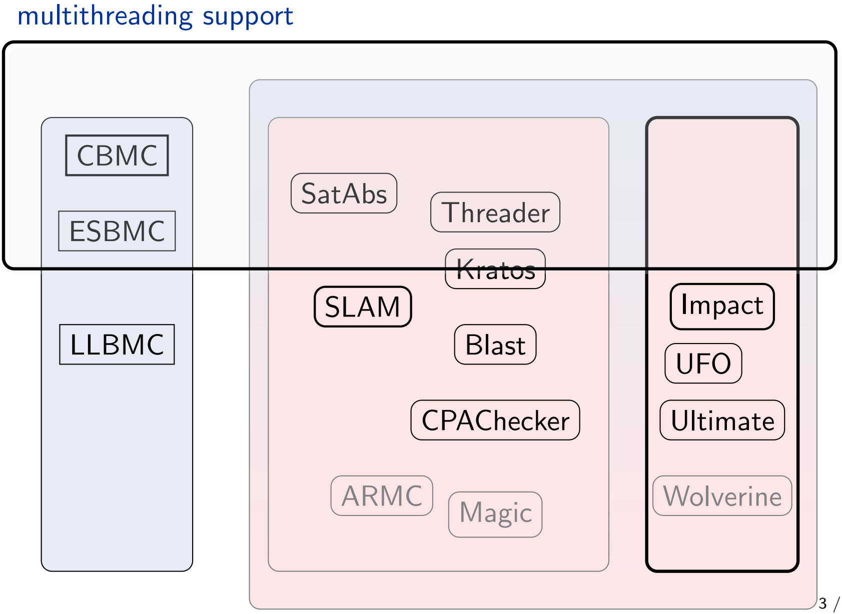
abstraction
predicate abstraction
Impact algorithm [McMillan 2006]

symbolic reasoning
SMT
SAT

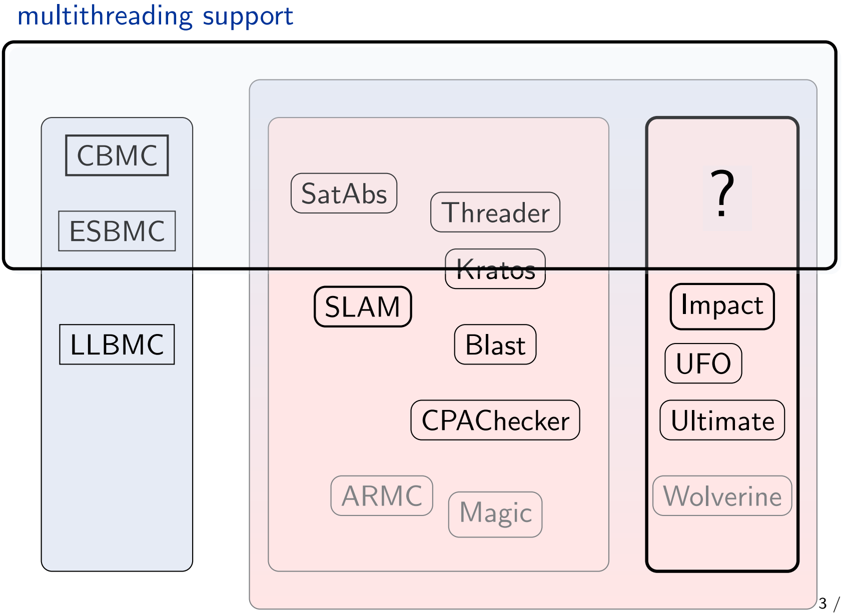
Software model checkers



Software model checkers

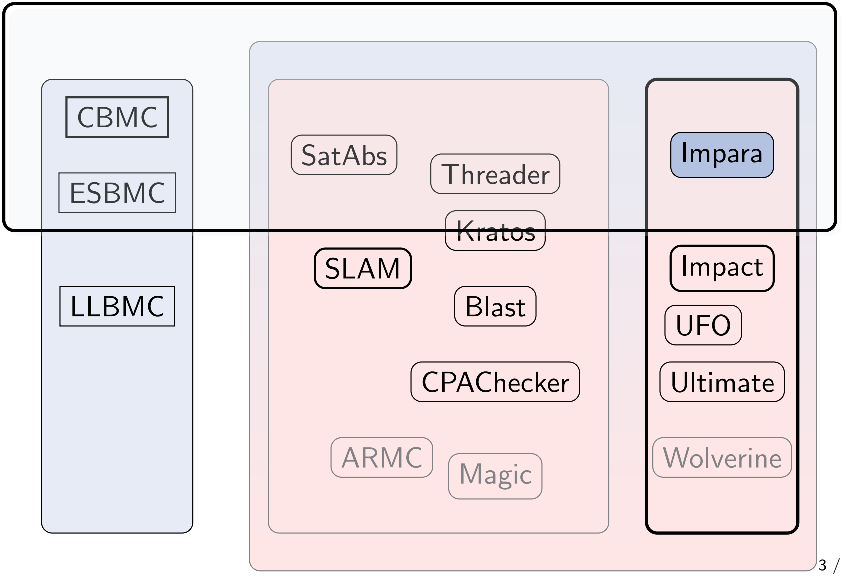


Software model checkers

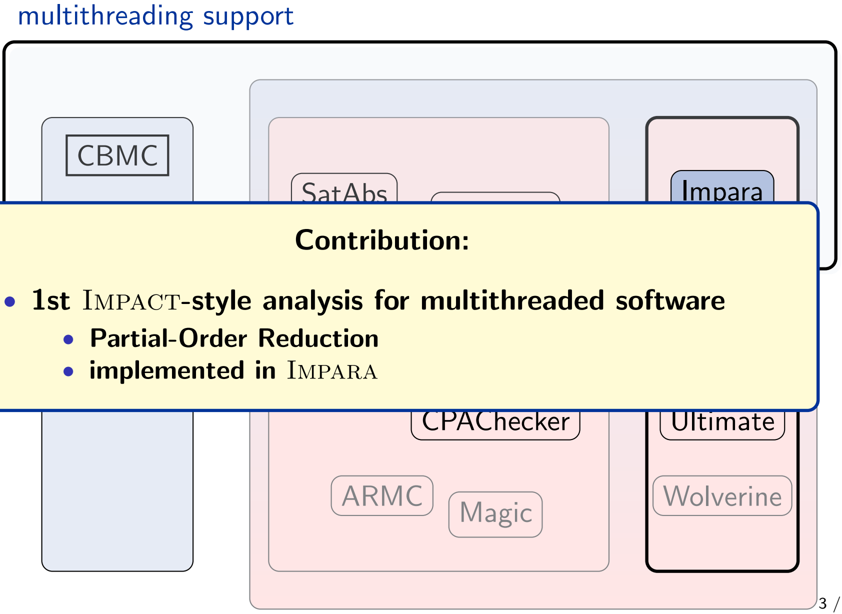


Software model checkers

multithreading support



Software model checkers



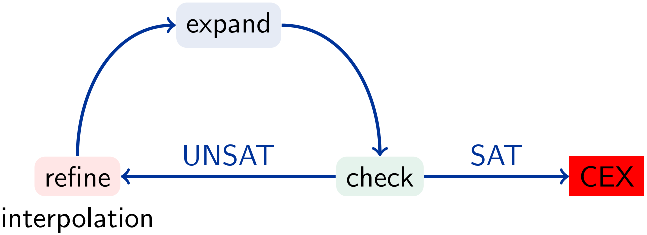
Contribution:

- **1st Impact-style analysis for multithreaded software**
  - **Partial-Order Reduction**
  - **implemented in Impara**

Outline

- Recap: Impact for Sequential Software
- Impact for Multithreaded Software
    - Partial order reduction
- Experiments with our tool Impara

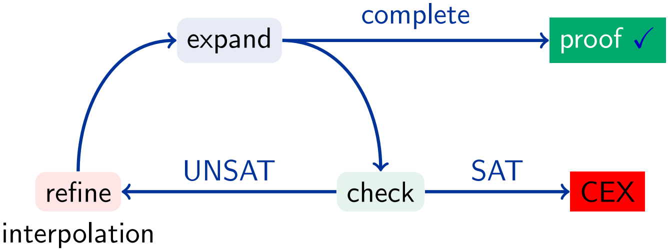
Impact algorithm



- maintain abstract reachability tree
  - node labels
  - covering relation $\triangleright$

$$v \triangleright w \text{ implies } label(v) \Rightarrow label(w)$$

Impact algorithm



- maintain abstract reachability tree
  - node labels
  - covering relation $\triangleright$

$$v \triangleright w \text{ implies } label(v) \Rightarrow label(w)$$

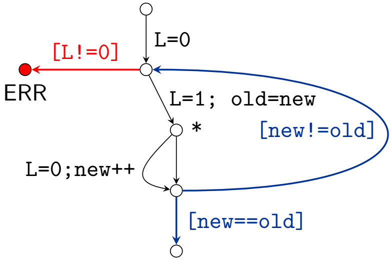
- complete iff all nodes either
  - covered
  - expanded

Classical SLAM example

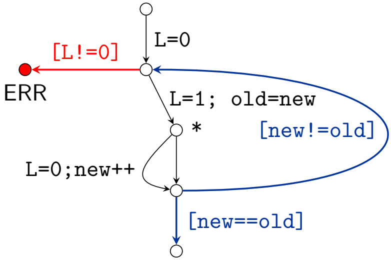
```
do {
  lock();
  old=new;
  if(*) {
    unlock();
    new++;
  }
} while (new!=old);
```

Classical SLAM example

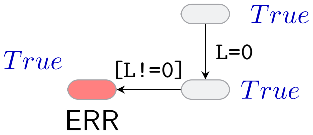
```
do {
  lock();
  old=new;
  if(*) {
    unlock();
    new++;
  }
} while (new!=old);
```



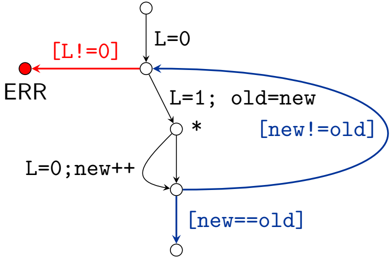
- reachable states ⊆ label



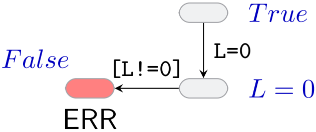
Abstract Reachability Tree



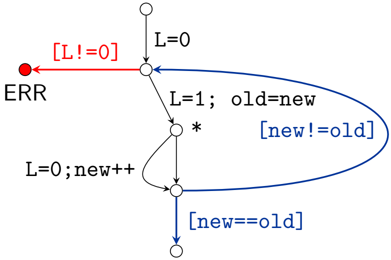
- reachable states ⊆ label



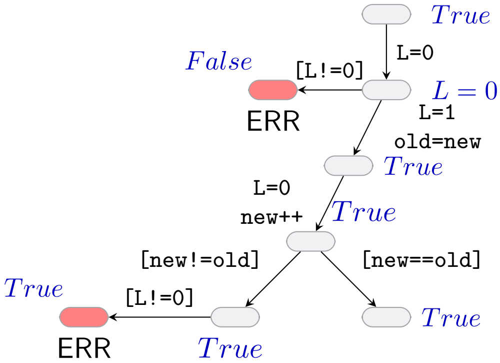
Abstract Reachability Tree



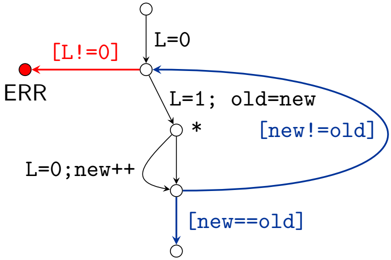
- reachable states ⊆ label



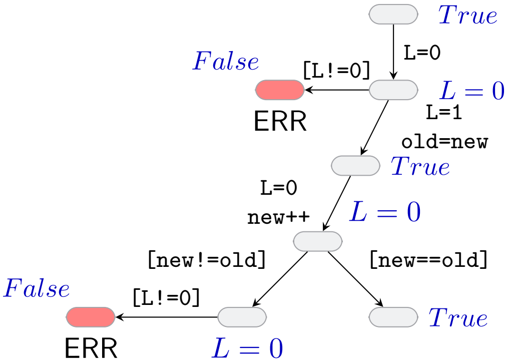
Abstract Reachability Tree



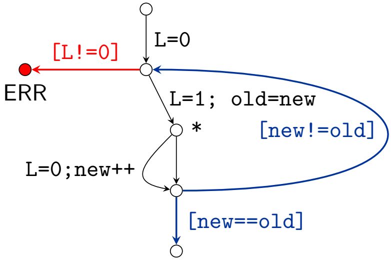
- reachable states $\subseteq$ label



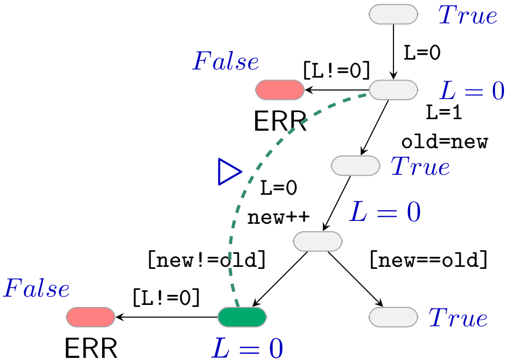
Abstract Reachability Tree



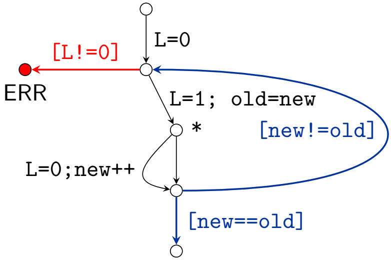
- reachable states ⊆ label



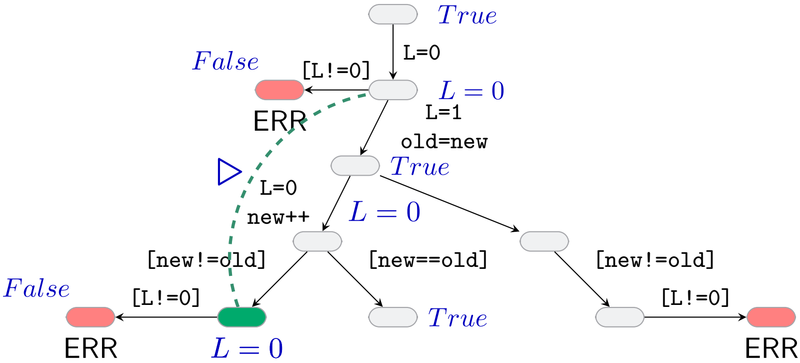
Abstract Reachability Tree



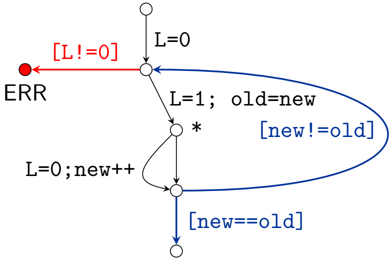
- reachable states ⊆ label



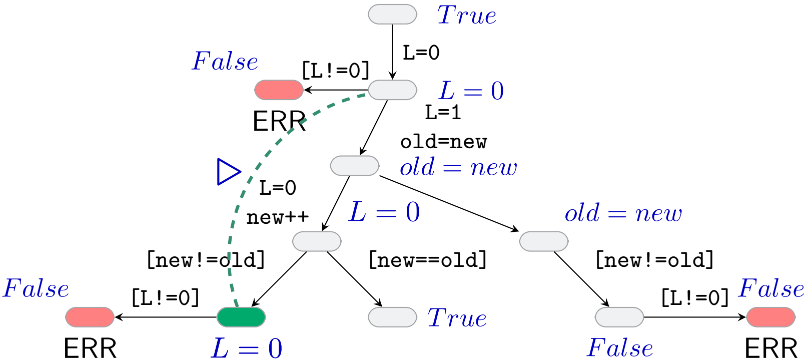
Abstract Reachability Tree



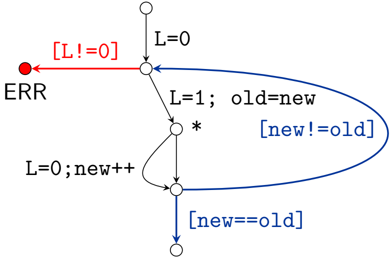
- reachable states $\subseteq$ label



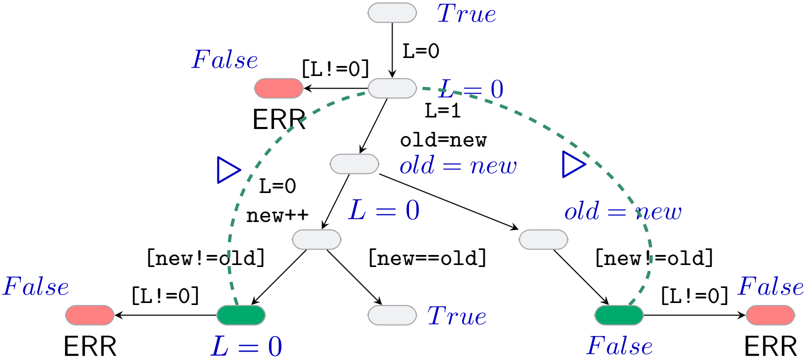
Abstract Reachability Tree



- reachable states ⊆ label
- terminates if all nodes
  - covered
  - or fully expanded



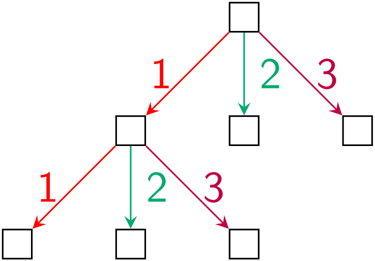
Abstract Reachability Tree



Impact for Multithreaded Software

Naive Impact for Multi-threading

- interleave at every step

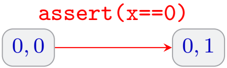


threads 1,2,3

Example

`int x=0;`

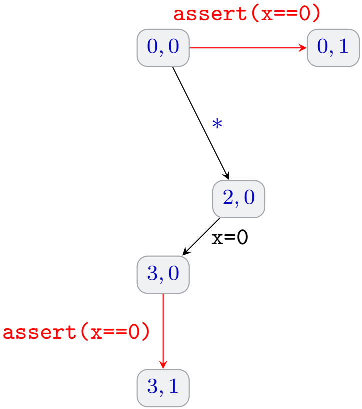
thread 1	thread 2
`0:  assert(x==0);`	`0:  if(*)`
`1:`	`1:    x=1;`
	`2:  x=0;`
	`3:`



Example

`int x=0;`

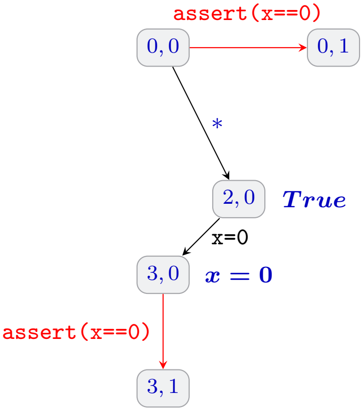
thread 1	thread 2
`0:  assert(x==0);`	`0:  if(*)`
`1:`	`1:    x=1;`
	`2:  x=0;`
	`3:`



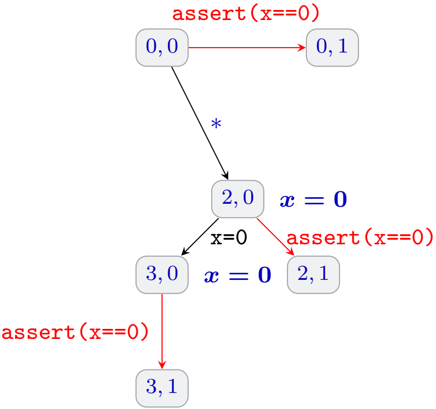
Example

`int x=0;`

thread 1	thread 2
`0:   assert(x==0);`	`0:   if(*)`
`1:`	`1:      x=1;`
	`2:   x=0;`
	`3:`



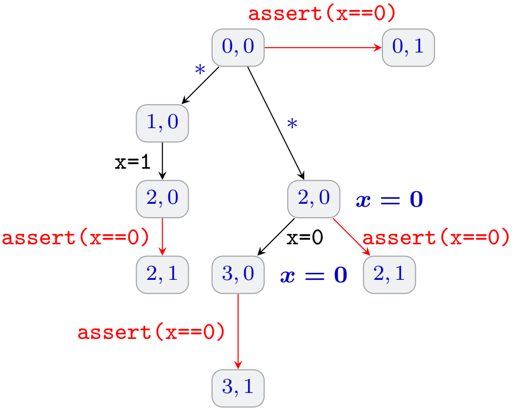
Example



```
int x=0;
```

thread 1	thread 2
0: assert(x==0);	0: if(*)
1:	1: x=1;
	2: x=0;
	3:

Example



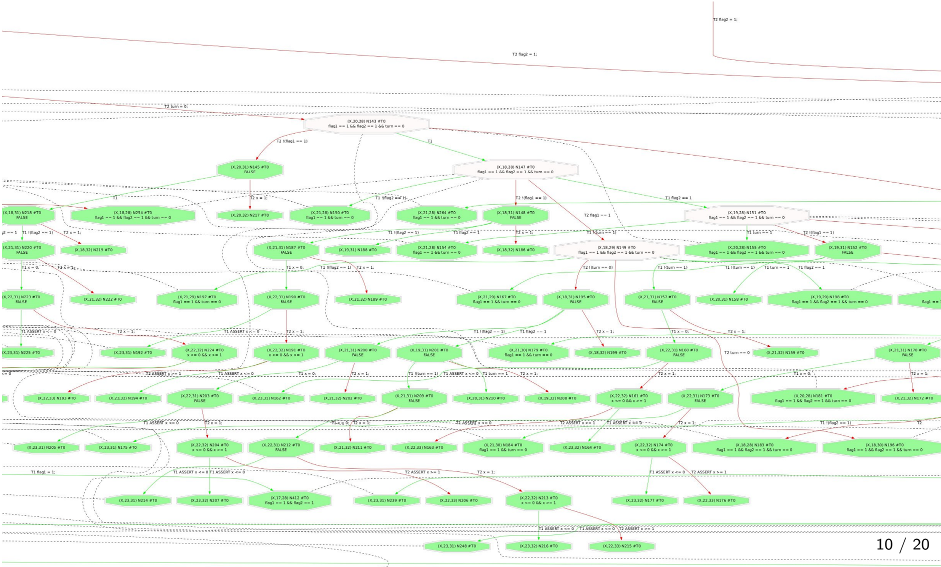
`int x=0;`

thread 1	thread 2
`0:  assert(x==0);`	`0:  if(*)`
`1:`	`1:    x=1;`
	`2:  x=0;`
	`3:`

CEX

Naive Impact blows up

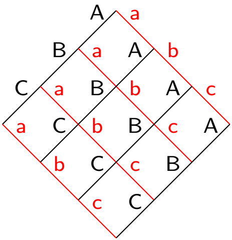
ART from a concrete case study (Peterson's algorithm)



Partial-Order Reduction [Godefroid'94, Peled'93, Valmari'90]

avoid unnecessary interleavings resulting in same state

main()	thread 1	thread 2
assume(i!=j);		
v[i]=0; v[j]=0;	A : v[i]=1;	a : v[j]=-2;
pthread_create(T_1);	B : v[i]=v[i]+1;	b : v[j]=v[j]+1;
pthread_create(T_2);	C : v[i]=v[j];	c : v[i]=v[i]+1;
pthread_join(T_1);		
pthread_join(T_2);		
assert(v[j] $\geq$ 0);		



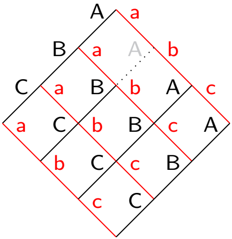
$A \mid\mid a$ and $TID(A) < TID(a)$

Partial-Order Reduction [Godefroid'94, Peled'93, Valmari'90]

avoid unnecessary interleavings resulting in same state

main()	thread 1	thread 2

`assume(i!=j);`
`v[i]=0; v[j]=0;`
`pthread_create(`T_1`);`
`pthread_create(`T_2`);`
`pthread_join(`T_1`);`
`pthread_join(`T_2`);`
`assert(v[j]` $\geq$ `0);`
	A : `v[i]=1;`
	B : `v[i]=v[i]+1;`
	C : `v[i]=v[j];`
		a : `v[j]=-2;`
		b : `v[j]=v[j]+1;`
		c : `v[i]=v[i]+1;`



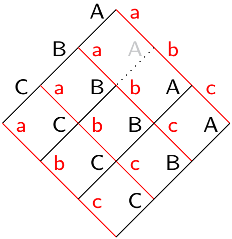
$A \mid\mid a$ and $TID(A) < TID(a)$

Partial-Order Reduction [Godefroid'94, Peled'93, Valmari'90]

avoid unnecessary interleavings resulting in same state

main()	thread 1	thread 2
`assume(i!=j);`		
`v[i]=0; v[j]=0;`	A : `v[i]=1;`	a : `v[j]=-2;`
`pthread_create(`T_1`);`	B : `v[i]=v[i]+1;`	b : `v[j]=v[j]+1;`
`pthread_create(`T_2`);`	C : `v[i]=v[j];`	c : `v[i]=v[i]+1;`
`pthread_join(`T_1`);`		
`pthread_join(`T_2`);`		
`assert(v[j]` $\geq$ `0);`		

consecutive independent actions only occur in the order of increasing thread ids, e.g., Aa but not aA



$A \parallel a$ and $TID(A) < TID(a)$

$B \parallel b$ and $TID(B) < TID(b)$

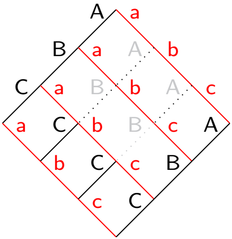
$A \parallel b$ and $TID(A) < TID(b)$

Partial-Order Reduction [Godefroid'94, Peled'93, Valmari'90]

avoid unnecessary interleavings resulting in same state

main()	thread 1	thread 2
assume(i!=j);		
v[i]=0; v[j]=0;	A : v[i]=1;	a : v[j]=-2;
pthread_create(T_1);	B : v[i]=v[i]+1;	b : v[j]=v[j]+1;
pthread_create(T_2);	C : v[i]=v[j];	c : v[i]=v[i]+1;
pthread_join(T_1);		
pthread_join(T_2);		
assert(v[j] $\geq$ 0);		

consecutive independent actions only occur in the order of increasing thread ids, e.g., Aa but not aA



$A \mid\mid a$ and $TID(A) < TID(a)$

$B \mid\mid b$ and $TID(B) < TID(b)$

$A \mid\mid b$ and $TID(A) < TID(b)$

Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

```
procedure $\textsc{Expand}_{\Diamond}(v)$
    for $T \in \mathcal{T}$ with $\neg\textsc{Skip}_{\Diamond}(v, T)$ do
        $\textsc{Expand-thread}(T, v)$
```

Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

```
procedure $\textsc{Expand}_{\Diamond}(v)$
    for $T \in \mathcal{T}$ with $\neg\textsc{Skip}_{\Diamond}(v, T)$ do
        $\textsc{Expand-thread}(T, v)$

procedure $\textsc{Skip}_{\Diamond}(v, T)$
    select unique parent action $T', a'$ s.t. $u \xrightarrow{T',a'} v$
    return $\left( T < T' \wedge \underbrace{\textsc{Action}(v, T) \mid\mid a'}_{\text{dependence check}} \right)$
```

Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

```
procedure $\textsc{Expand}_\Diamond(v)$
    for $T \in \mathcal{T}$ with $\neg\textsc{Skip}_\Diamond(v, T)$ do
        $\textsc{Expand-thread}(T, v)$

procedure $\textsc{Skip}_\Diamond(v, T)$
    select unique parent action $T', a'$ s.t. $u \xrightarrow{T', a'} v$
    return $\left( T < T' \wedge \underbrace{\textsc{Action}(v, T) \mid\mid a'}_{\text{dependence check}} \right)$
```

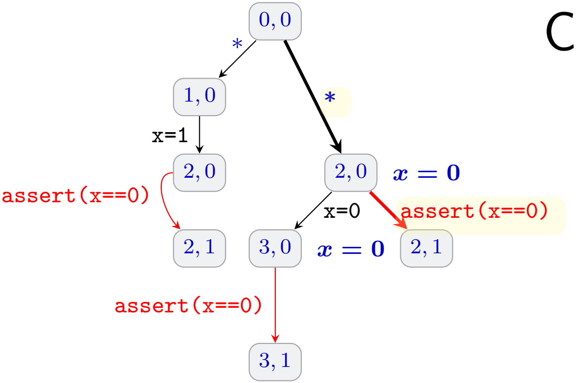
Is that sound?

Impact + **POR**

`int x=0;`

thread 1	thread 2
`0:   assert(x==0);`	`0:   if(*)`
`1:`	`1:      x=1;`
	`2:   x=0;`
	`3:`

CEX

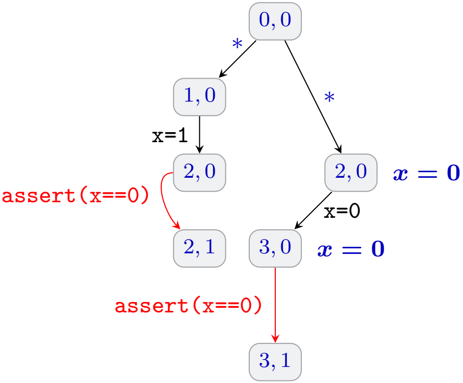


- $*$ and `assert(x==0)` independent

Impact + POR

`int x=0;`

thread 1	thread 2
`0:  assert(x==0);`	`0:  if(*)`
`1:`	`1:     x=1;`
	`2:  x=0;`
	`3:`



CEX

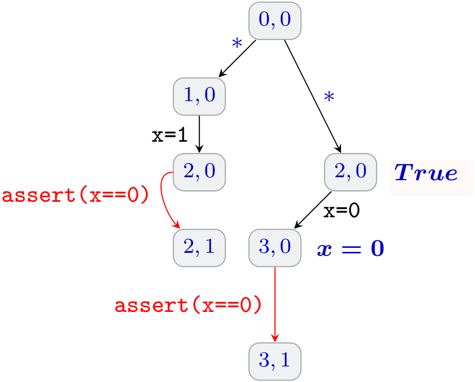
- $*$ and `assert(x==0)` independent
- reduction

Impact + **POR**

int x=0;

thread 1	thread 2
0: assert(x==0);	0: if(*)
1:	1: x=1;
	2: x=0;
	3:

CEX



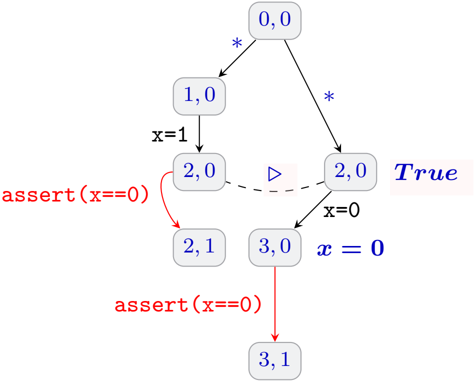
- $*$ and assert(x==0) independent
- reduction

Impact + **POR**

```
int x=0;
```

thread 1	thread 2
`0:  assert(x==0);`	`0:  if(*)`
`1:`	`1:      x=1;`
	`2:  x=0;`
	`3:`

CEX

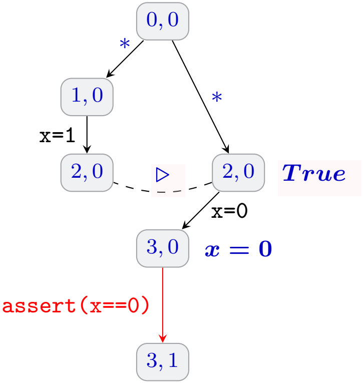


- $*$ and assert(x==0) independent
- reduction

Impact + **POR**

`int x=0;`

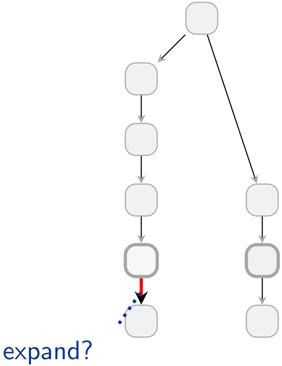
thread 1	thread 2
`0:   assert(x==0);`	`0:   if(*)`
`1:`	`1:      x=1;`
	`2:   x=0;`
	`3:`



CEX

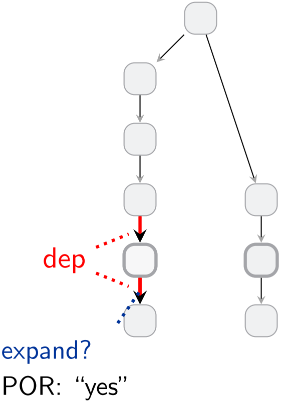
- $*$ and `assert(x==0)` independent
- reduction

Let's take a step back



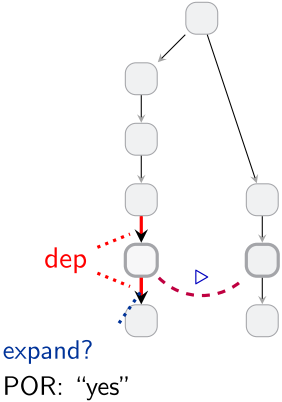
Let's take a step back

- POR inspects node history



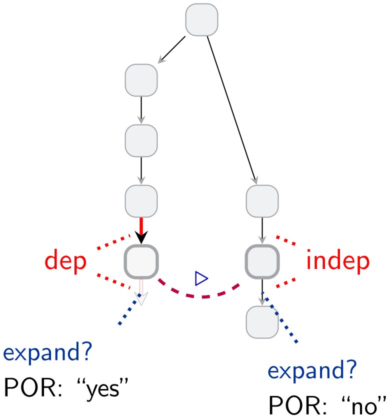
Let's take a step back

- POR inspects node history
- covers merge distinct histories



Let's take a step back

- POR inspects node history
- covers merge distinct histories

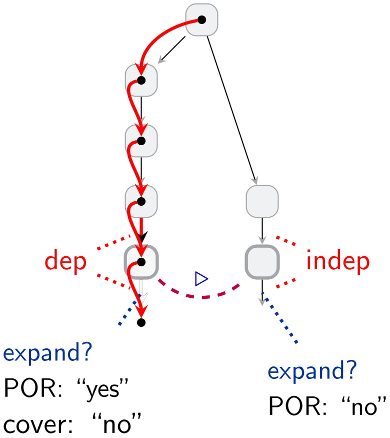


Let's take a step back

- POR inspects node history
- covers merge distinct histories

⇒ incomplete: lost program path

  - no corresponding ART path



expand?

POR: “yes”

cover: “no”

expand?

POR: “no”

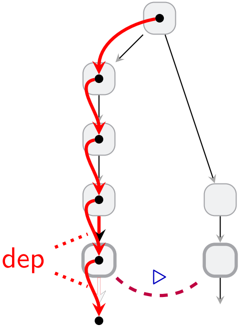
Let's take a step back

- POR inspects node history
- covers merge distinct histories

⇒ incomplete: lost program path

  - no corresponding ART path

- How to fix this?
  - corresponding path?
  - allow cover edges
  - jump to more abstract node



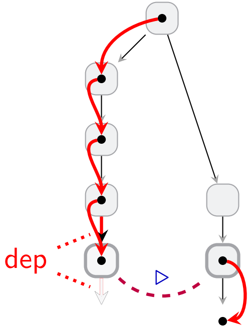
Let's take a step back

- POR inspects node history
- covers merge distinct histories

⇒ incomplete: lost program path

  - no corresponding ART path

- How to fix this?
  - corresponding path?
  - allow cover edges
  - jump to more abstract node

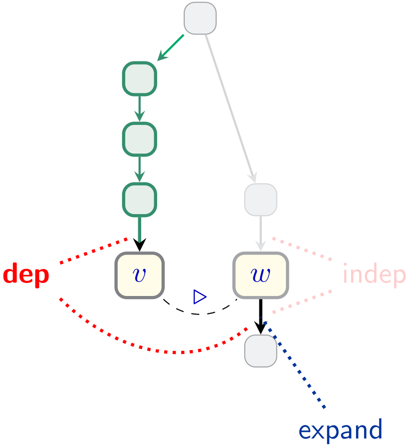


Complete Algorithm

- $v \rhd w$

$\Rightarrow$ consider **both** **histories**

- v's and w's



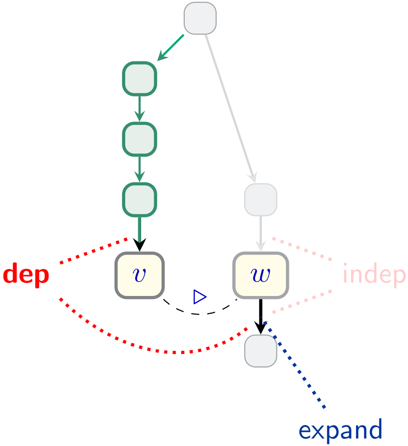
Complete Algorithm

- $v \rhd w$

$\Rightarrow$ consider **both** **histories**

  - v's and w's

- Note: we're still doing POR



Π-completeness

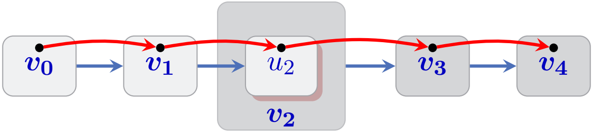
Π determined by POR strategy

Definition (Π-complete ART)

ART $\mathcal{A}$ is Π-complete iff:

for every $\boldsymbol{\pi} \in \prod$, there is a corresponding path $v_0, \ldots, v_n$.

Π-completeness



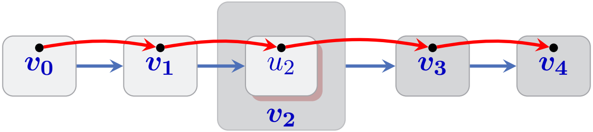
Π determined by POR strategy

Definition (Π-complete ART)

ART $\mathcal{A}$ is Π-complete iff:

for every $\boldsymbol{\pi} \in \prod$, there is a corresponding path $v_0, \ldots, v_n$.

Π-completeness



Π determined by POR strategy

Definition (Π-complete ART)

ART $\mathcal{A}$ is Π-complete iff:

for every $\boldsymbol{\pi} \in \prod$, there is a corresponding path $v_0, \ldots, v_n$.

$\Rightarrow$ **Soundness**

Impara

- C++ implementation
- CBMC frontend
- bit-precise interpolation
    - unsatisfiable cores + weakest preconditions

Impara vs. other tools

	CBMC 4.5	ESMBC	SatAbs	Threader	Impara
technique	BMC	BMC	Pred. Abs.	Pred. Abs.	Interpolation
threads	PO encoding	POR	POR	Modular Reasoning	POR
unbounded loops			✓	✓	✓
bit-precise	✓	✓	✓		✓
weak memory	✓		✓*		✓*

SVCOMP'13 multi-threading benchmarks

program	safe	CBMC	ESBMC	SatAbs	Threader	Impara
`dekker`	y	0.6*	2.2*	0.2	TO	**0.1**
`lamport`	y	12.4*	18.1*	**0.3**	38.1	**0.3**
`peterson`	y	0.2*	2.0*	0.3	4.8	**0.1**
`szymanski`	y	0.5*	4.7*	**0.2**	13.5	**0.2**
`read_write_u`	n	**0.2**	TO	0.8	58.4	0.6
`read_write_s`	y	**0.4**	TO	0.8	58.1	0.9
`time_var_mutex`	y	0.2	110.3	95.4	4.3	**0.1**
`stack_u`	n	1.0	TO	TO	80.6	**0.5**
`stack_s`	y	**33.5**	TO	TO	250.1	38.8

Conclusion

- IMPACT abstraction + POR
    - take-home message: **look at both histories**
- Experiments
    - SVCOMP'13
    - weak memory benchmarks (low-lock algorithms)
        - IMPARA gives correct results
        - which gives us confidence
- Binary & benchmarks at:

`http://www.cprover.org/concurrent-impact/`

Thank you!