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

• Verification Challenges

symbolic reasoning
SMT
SAT

loops
variables
pointers
data
WMM
SC
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
  loops
  variables
  pointers
  data

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

- Impact algorithm [McMillan 2006]
Software model checkers

- CBMC
- ESBMC
- LLBMC
- SatAbs
- Threader
- Kratos
- SLAM
- Blast
- CPAChecker
- ARMC
- Magic
- Impact
- UFO
- Ultimate
- Wolverine
Software model checkers

multithreading support

- CBMC
- ESBMC
- LLBMC
- SatAbs
- Threader
- Kratos
- SLAM
- Blast
- CPAChecker
- ARMC
- Magic
- Impact
- UFO
- Ultimate
- Wolverine
Software model checkers

multithreading support

- CBMC
- ESBMC
- LLBMC
- SatAbs
- Threader
- Kratos
- SLAM
- Blast
- CPAChecker
- ARMC
- Magic
- Impact
- UFO
- Ultimate
- Wolverine
Software model checkers

multithreading support

- CBMC
- ESBMC
- LLBMC
- IMPARA
- Impala
- SatAbs
- Threader
- Kratos
- SLAM
- Blast
- CPAChecker
- ARMC
- Magic
- Wolverine
- Impact
- UFO
- Ultimate
Software model checkers

multithreading support

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

- maintain abstract reachability tree
  - node labels
  - covering relation $\triangleright$
    
    $v \triangleright w$ implies $\text{label}(v) \Rightarrow \text{label}(w)$

- 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 $\subseteq$ label

Abstract Reachability Tree

True

ERR

True

[\[L!\neq 0\]]

L=0

L=0; new++

L=1; old=new

[\[new!\neq old\]]

[\[new==old\]]
- reachable states $\subseteq$ label

Abstract Reachability Tree

**False**

$L = 0$

ERR

**True**

$L = 0$

$L = 1$; old=new

[new!=old]

[new==old]

ERR
Classical SLAM example

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

Abstract Reachability Tree

- reachable states ⊆ label

```
L=0
[ L!=0 ]
ERR

L=1; old=new
[ new!=old ]
L=0; new++
[ new==old ]

L=0
[ L!=0 ]

L=1
old=new
True

L=0
new++
True

[ new!=old ]

True

[ new==old ]

False
[ L!=0 ]
ERR

L=0
True

L=1
old=new
True
```
Classical SLAM example

```c

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

Abstract Reachability Tree

- reachable states ⊆ label

Diagram:

```
ERR
```

```
L=0;new++
```

```
L=1; old=new
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[L!=0]*
```

```
L=0
```

```
L=1
```

```
old=new
```

```
L=0
new++
```

```
*[new!=old]*
```

```
L=0
```

```
*[L!=0]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```

```
*[new==old]*
```

```
L=0
```

```
*[new!=old]*
```

```
L=0
```
- reachable states $\subseteq$ label

**Abstract Reachability Tree**

- **False**
  - $[L! = 0]$  
  - $L = 0$
  - $L = 1$
  - $old = new$
  - $L = 0$
  - $new++$

- **False**
  - $[L! = 0]$
  - $L = 0$

- **False**
  - $[new! = old]$

- **True**
  - $[new = old]$

- **ERR**
  - $L = 0$
  - $L = 1$
  - $old = new$
  - $L = 0$
  - $new++$
- reachable states $\subseteq$ label

Abstract Reachability Tree

- $L = 0$
  - $[L = 0]$ (True)
  - $[L = 1; \text{old} = \text{new}]$ (False)
  - $[\text{old} = \text{new}]$ (True)

- $L = 0$
  - $[L = 0]$ (True)
  - $[\text{new} = \text{old}]$ (False)

- $\text{new}++$
  - $[\text{new} = \text{old}]$ (False)
  - $[L = 0]$ (True)

- $\text{ERR}$
• reachable states $\subseteq$ label
- 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

```c
int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1: x=1;</td>
<td>1: x=1;</td>
</tr>
<tr>
<td>2: x=0;</td>
<td>2: x=0;</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
</tbody>
</table>
```

```
thread 1:
0: assert(x==0);
1: x=0;

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

```
assert(x==0)
```

```c
0, 0 → 0, 1
```
Example

int x = 0;

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

```
int x = 0;
thread 1
0: assert(x == 0);
1: 
3: 
```

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

```c
int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1: x=1;</td>
<td>1: x=1;</td>
</tr>
<tr>
<td>2: x=0;</td>
<td>2: x=0;</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
</tbody>
</table>
```

![Diagram](image)

```c
int x=0;

assert(x==0)
0, 0
*
0, 0
*
2, 0
x=0
3, 0
x = 0
3, 0
assert(x==0)
0, 1
```

**Example**

```c
int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1: x=1;</td>
<td>1: x=1;</td>
</tr>
<tr>
<td>2: x=0;</td>
<td>2: x=0;</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
</tbody>
</table>
```
Example

```
int x=0;

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

```
int x=0;

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

```
int x=0;

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

```
int x=0;

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

```
int x=0;

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

```
int x=0;

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

```
int x=0;

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

```
assert(x==0)
1,0
```

```
2,0
x=1
2,1
assert(x==0)
```

```
0,0
* 0,1
```

```
2,0
* 2,1
```

```
x = 0
2,0
```

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

```
int x=0;
```
Naive Impact blows up

ART from a concrete case study (Peterson's algorithm)
Partial-Order Reduction \cite{Godefroid'94, Peled'93, Valmari'90}

avoid unnecessary interleavings resulting in same state

<table>
<thead>
<tr>
<th>main()</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>assume(i!=j);</td>
<td>A: v[i]=1;</td>
<td>a: v[j]=-2;</td>
</tr>
<tr>
<td>v[i]=0; v[j]=0;</td>
<td>B: v[i]=v[i]+1;</td>
<td>b: v[j]=v[j]+1;</td>
</tr>
<tr>
<td>pthread_create(T1);</td>
<td>C: v[i]=v[j];</td>
<td>c: v[i]=v[i]+1;</td>
</tr>
<tr>
<td>pthread_create(T2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_join(T1);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_join(T2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assert(v[j] ≥ 0);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

avoid unnecessary interleavings resulting in same state

<table>
<thead>
<tr>
<th>main()</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>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] ≥ 0);</td>
<td>A: v[i]=1; B: v[i]=v[i]+1; C: v[i]=v[j];</td>
<td>a: v[j]=-2; b: v[j]=v[j]+1; c: v[i]=v[i]+1;</td>
</tr>
</tbody>
</table>

A || a and TID(A) < TID(a)
Partial-Order Reduction [Godefroid’94, Peled’93, Valmari’90]

avoid unnecessary interleavings resulting in same state

<table>
<thead>
<tr>
<th>main()</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>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] ≥ 0);</td>
<td>A: v[i]=1; B: v[i]=v[i]+1; C: v[i]=v[j];</td>
<td>a: v[j]=-2; b: v[j]=v[j]+1; c: v[i]=v[i]+1;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>a</td>
</tr>
<tr>
<td>C</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>C</td>
</tr>
<tr>
<td>b</td>
<td>C</td>
</tr>
<tr>
<td>c</td>
<td>C</td>
</tr>
</tbody>
</table>

A || a and TID(A) < TID(a)
B || b and TID(B) < TID(b)
A || b and TID(A) < TID(b)
Partial-Order Reduction  [Godefroid’94, Peled’93, Valmari’90]

avoid unnecessary interleavings resulting in same state

<table>
<thead>
<tr>
<th>main()</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>assume((i\neq j));</td>
<td>(A : v[i]=1;)</td>
<td>(a : v[j]=-2;)</td>
</tr>
<tr>
<td>(v[i]=0; \ v[j]=0;)</td>
<td>(B : v[i]=v[i]+1;)</td>
<td>(b : v[j]=v[j]+1;)</td>
</tr>
<tr>
<td>pthread_create((T_1));</td>
<td>(C : v[i]=v[j];)</td>
<td>(c : v[i]=v[i]+1;)</td>
</tr>
<tr>
<td>pthread_create((T_2));</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_join((T_1));</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pthread_join((T_2));</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assert((v[j] \geq 0));</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[ A \parallel a \text{ and } \text{TID}(A) < \text{TID}(a) \]
\[ B \parallel b \text{ and } \text{TID}(B) < \text{TID}(b) \]
\[ A \parallel b \text{ and } \text{TID}(A) < \text{TID}(b) \]
Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

1: procedure $\text{EXPAND}_\Diamond(v)$
2: for $T \in \mathcal{T}$ with $\neg\text{SKIP}_\Diamond(v, T)$ do
3: $\text{EXPAND-THREAD}(T, v)$
Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

1: procedure $\text{EXPAND}^\Diamond(v)$
2: for $T \in \mathcal{T}$ with $\neg \text{SKIP}^\Diamond(v, T)$ do
3: \hspace{1em} $\text{EXPAND-THREAD}(T, v)$
4: 
5: procedure $\text{SKIP}^\Diamond(v, T)$
6: \hspace{1em} select unique parent action $T', a'$ s.t. $u^{T',a'} \rightarrow v$
7: \hspace{1em} return $\left( T < T' \land \text{ACTION}(v, T) \parallel a' \right)$

\hspace{1em} dependence check
Algorithm: POR+Impact (First Attempt)

- POR restricts expansion

1: procedure $\text{EXPAND}_\diamond(v)$
2: for $T \in \mathcal{T}$ with $\neg \text{SKIP}_\diamond(v, T)$ do
3: \hspace{1em} $\text{EXPAND-THREAD}(T, v)$
4: 
5: procedure $\text{SKIP}_\diamond(v, T)$
6: \hspace{1em} select unique parent action $T', a'$ s.t. $u^{T', a'} \rightarrow v$
7: \hspace{1em} return $\begin{pmatrix} T < T' \land \text{ACTION}(v, T) \parallel a' \\ \text{dependence check} \end{pmatrix}$

Is that sound?
Impact + POR

int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1:</td>
<td>1: x=1;</td>
</tr>
<tr>
<td></td>
<td>2: x=0;</td>
</tr>
<tr>
<td></td>
<td>3:</td>
</tr>
</tbody>
</table>

thread 1

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

thread 2

0: assert(x==0);
1: x=1;
2: x=0;
3: assert(x==0);

• * and assert(x==0) independent

CEX

```c
int x=0;

thread 1          | thread 2          |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1:</td>
<td>1: x=1;</td>
</tr>
<tr>
<td></td>
<td>2: x=0;</td>
</tr>
<tr>
<td></td>
<td>3:</td>
</tr>
</tbody>
</table>

thread 1

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

thread 2

0: assert(x==0);
1: x=1;
2: x=0;
3: assert(x==0);
```
Impact + POR

int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1:</td>
<td>1: x=1;</td>
</tr>
<tr>
<td></td>
<td>2: x=0;</td>
</tr>
<tr>
<td></td>
<td>3:</td>
</tr>
</tbody>
</table>

CEX

- * and assert(x==0) independent
- reduction
Impact + POR

int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1: x=1;</td>
<td>1:</td>
</tr>
<tr>
<td>2: x=0;</td>
<td>2: x=0;</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
</tbody>
</table>

CEX

- * and assert(x==0) independent
- reduction
Impact + POR

Impact + POR

int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1:</td>
<td>1: x=1;</td>
</tr>
<tr>
<td></td>
<td>2: x=0;</td>
</tr>
<tr>
<td></td>
<td>3:</td>
</tr>
</tbody>
</table>

thread 1

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

thread 2

int x=0;

bullet: * and assert(x==0) independent
bullet: reduction
Impact + POR

```
int x=0;

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert(x==0);</td>
<td>0: if(*)</td>
</tr>
<tr>
<td>1:</td>
<td>1: x=1;</td>
</tr>
<tr>
<td></td>
<td>2: x=0;</td>
</tr>
<tr>
<td></td>
<td>3:</td>
</tr>
</tbody>
</table>
```

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

CEX

```
assert(x==0)
```

```
0, 0

1, 0

x=1

2, 0

> True

2, 0

x=0

3, 0

x = 0

assert(x==0)

3, 1
```
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

dep
indep

POR: “no”
cover: “no”
expand?

expand?
Let’s take a step back

- POR inspects node history

POR: “no”

expand? POR: “yes”

dep
Let’s take a step back

- POR inspects node **history**
- **covers** merge distinct histories

POR: “no”

expand?

POR: “yes”

dep

expand?
Let’s take a step back

- POR inspects node history
- covers merge distinct histories

```
POR: “no”
```

```
POR: “yes”
```
Let’s take a step back

- POR inspects node **history**
- **covers** merge distinct histories
  ⇒ **incomplete**: lost program path
  - no corresponding ART path

- POR: “no”
- expand?
- dep
- indep
- cover: “no”
- POR: “yes”
- expand?
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

- $v \succ w$

$\Rightarrow$ consider both histories
  - $v$'s and $w$'s
Complete Algorithm

- $v \triangleright w$
- $\Rightarrow$ 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 \( \Pi \)-complete iff:

for every \( \pi \in \Pi \), 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 $\pi \in \Pi$, there is a corresponding path $v_0, \ldots, v_n$. 
Definition (\(\Pi\)-complete ART)

ART \(A\) is \(\Pi\)-complete iff:

for every \(\pi \in \Pi\), there is a corresponding path \(v_0, \ldots, v_n\).
IMPARA

- C++ implementation
- CBMC frontend
- bit-precise interpolation
  - unsatisfiable cores + weakest preconditions
**IMPARA vs. other tools**

<table>
<thead>
<tr>
<th>technique</th>
<th>CBMC 4.5</th>
<th>ESMBC</th>
<th>SatAbs</th>
<th>Threader</th>
<th>Impara</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>BMC</td>
<td>BMC</td>
<td>Pred. Abs.</td>
<td>Pred. Abs.</td>
<td>Interpolation</td>
</tr>
<tr>
<td>PO encoding</td>
<td>POR</td>
<td>POR</td>
<td>POR</td>
<td>Modular Reasoning</td>
<td>POR</td>
</tr>
<tr>
<td>unbounded loops</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bit-precise</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>weak memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**SVCOMP’13 multi-threading benchmarks**

<table>
<thead>
<tr>
<th>program</th>
<th>safe</th>
<th>CBMC</th>
<th>ESBMC</th>
<th>SatAbs</th>
<th>THREAVER</th>
<th>IMPARA</th>
</tr>
</thead>
<tbody>
<tr>
<td>dekker</td>
<td>y</td>
<td>0.6*</td>
<td>2.2*</td>
<td>0.2</td>
<td>TO</td>
<td>0.1</td>
</tr>
<tr>
<td>lamport</td>
<td>y</td>
<td>12.4*</td>
<td>18.1*</td>
<td>0.3</td>
<td>38.1</td>
<td>0.3</td>
</tr>
<tr>
<td>peterson</td>
<td>y</td>
<td>0.2*</td>
<td>2.0*</td>
<td>0.3</td>
<td>4.8</td>
<td>0.1</td>
</tr>
<tr>
<td>szymanski</td>
<td>y</td>
<td>0.5*</td>
<td>4.7*</td>
<td>0.2</td>
<td>13.5</td>
<td>0.2</td>
</tr>
<tr>
<td>read_write_u</td>
<td>n</td>
<td>0.2</td>
<td>TO</td>
<td>0.8</td>
<td>58.4</td>
<td>0.6</td>
</tr>
<tr>
<td>read_write_s</td>
<td>y</td>
<td>0.4</td>
<td>TO</td>
<td>0.8</td>
<td>58.1</td>
<td>0.9</td>
</tr>
<tr>
<td>time_var_mutex</td>
<td>y</td>
<td>0.2</td>
<td>110.3</td>
<td>95.4</td>
<td>4.3</td>
<td>0.1</td>
</tr>
<tr>
<td>stack_u</td>
<td>n</td>
<td>1.0</td>
<td>TO</td>
<td>TO</td>
<td>80.6</td>
<td>0.5</td>
</tr>
<tr>
<td>stack_s</td>
<td>y</td>
<td>33.5</td>
<td>TO</td>
<td>TO</td>
<td>250.1</td>
<td>38.8</td>
</tr>
</tbody>
</table>
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!