

Local Nontermination Detection for Parallel C++ Programs

Vladimír Štill Jiří Barnat



Masaryk University
Brno, Czech Republic

6th December 2019



“Would you trust a program which was verified, but not tested?”



“Would you trust a program which was verified, but not tested?”

... at the very least, we should not blindly trust safety checking



- targeting assertion violations, memory corruption, data races
- primarily caused by thread interleaving
- or by relaxed memory



- targeting assertion violations, memory corruption, data races
- primarily caused by thread interleaving
- or by relaxed memory
- if the program might not terminate. . .
 - the tool might not terminate
 - or it might report there are no safety violations



- targeting assertion violations, memory corruption, data races
- primarily caused by thread interleaving
- or by relaxed memory
- if the program might not terminate. . .
 - the tool might not terminate
 - or it might report there are no safety violations (correctly)



- targeting assertion violations, memory corruption, data races
- primarily caused by thread interleaving
- or by relaxed memory
- if the program might not terminate. . .
 - the tool might not terminate
 - or it might report there are no safety violations (correctly)
- not enough for parallel programs



- check that the whole program terminates



- check that the whole program terminates
- or checks that certain parts of it terminate
 - critical sections
 - waiting for condition variables, threads. . .
 - user-defined parts



- we aim at nontermination caused by unintended parallel interactions



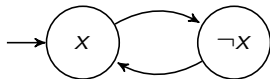
- we aim at nontermination caused by unintended parallel interactions
- not at complex control flow & loops



- we aim at nontermination caused by unintended parallel interactions
- not at complex control flow & loops
- should be easy to specify
- should not report nontermination spuriously
- should be useful for analysis of services/servers

- we aim at nontermination caused by unintended parallel interactions
 - not at complex control flow & loops
 - should be easy to specify
 - should not report nontermination spuriously
 - should be useful for analysis of services/servers
-
- builds on explicit-state model checking \rightarrow finite-state programs (with possibly infinite behaviour)
 - user can specify what to check

```
bool x = true;  
while (true) { x = !x; }
```





What is Nontermination?

```
mutex mtx;  
void w() { mutex.lock(); x++; mutex.unlock(); }  
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate?



What is Nontermination?

```
mutex mtx;  
void w() { mutex.lock(); x++; mutex.unlock(); }  
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate? ... yes

What is Nontermination?

```
atomic< bool > spin_lock = false;
void w() {
    while (spin_lock.exchange(true)) { /* wait */ }
    x++;
    spin_lock = false;
}
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate?

What is Nontermination?

```
atomic< bool > spin_lock = false;
void w() {
    while (spin_lock.exchange(true)) { /* wait */ }
    x++;
    spin_lock = false;
}
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate? ... yes

What is Nontermination?

```
atomic< bool > spin_lock = false;
void w() {
    while (spin_lock.exchange(true)) { /* wait */ }
    x++;
    spin_lock = false;
}
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate? ... yes

But there is an *infinite run*:

[t0: spin_lock.exchange(true) → false]

[t1: spin_lock.exchange(true) → true]^ω (repeats infinitely)

What is Nontermination?

```
atomic< bool > spin_lock = false;
void w() {
    while (spin_lock.exchange(true)) { /* wait */ }
    x++;
    spin_lock = false;
}
int main() { thread t0(w), t1(w); t0.join(); t1.join(); }
```

Does this program terminate? ... yes

But there is an *infinite run*:

[t0: spin_lock.exchange(true) → false]

[t1: spin_lock.exchange(true) → true]^ω (repeats infinitely)

but only because t0 is not allowed to run

What is Nontermination?

```
void w() {  
    while (true) {  
        while (spin_lock.exchange(true)) { /* wait */ }  
        x++;  
        spin_lock = false;  
    }  
}
```

Does every *wait* end?

What is Nontermination?

```
void w() {  
    while (true) {  
        while (spin_lock.exchange(true)) { /* wait */ }  
        x++;  
        spin_lock = false;  
    }  
}
```

Does every *wait* end? yes

What is Nontermination?

```
void w() {  
    while (true) {  
        while (spin_lock.exchange(true)) { /* wait */ }  
        x++;  
        spin_lock = false;  
    }  
}
```

Does every *wait* end? yes?

What is Nontermination?

```
void w() {
    while (true) {
        while (spin_lock.exchange(true)) { /* wait */ }
        x++;
        spin_lock = false;
    }
}
```

Does every *wait* end? yes?

```
[t0: spin_lock.exchange(true) → false]
([t1: spin_lock.exchange(true) → true]
[t0: x++]
[t0: spin_lock = false]
[t0: spin_lock.exchange(true) → false])ω
```

both threads can run

What is Nontermination?

```
[t0:  spin_lock.exchange(true) → false]
([t1:  spin_lock.exchange(true) → true]
 [t0:  x++]
 [t0:  spin_lock = false]
 [t0:  spin_lock.exchange(true) → false])ω
```

- this run requires a scheduler which allows t1 to run only if t0 is in the critical section

What is Nontermination?

```
[t0:  spin_lock.exchange(true) → false]
([t1:  spin_lock.exchange(true) → true]
 [t0:  x++]
 [t0:  spin_lock = false]
 [t0:  spin_lock.exchange(true) → false])ω
```

- this run requires a scheduler which allows t1 to run only if t0 is in the critical section
- does not happen in reality

What is Nontermination?

```
[t0:  spin_lock.exchange(true) → false]
([t1:  spin_lock.exchange(true) → true]
 [t0:  x++]
 [t0:  spin_lock = false]
 [t0:  spin_lock.exchange(true) → false])ω
```

- this run requires a scheduler which allows t1 to run only if t0 is in the critical section
- does not happen in reality
- for realistic schedulers an infinite run does not imply nontermination

What is Nontermination?

Nontermination

- a program does not terminate if it can reach a point from which it cannot reach its end

What is Nontermination?

Nontermination

- a program does not terminate if it can reach a point from which it cannot reach its end

Resource Section

- a block of code with an identifier
- delimited in the source code

What is Nontermination?

Nontermination

- a program does not terminate if it can reach a point from which it cannot reach its end

Resource Section

- a block of code with an identifier
- delimited in the source code

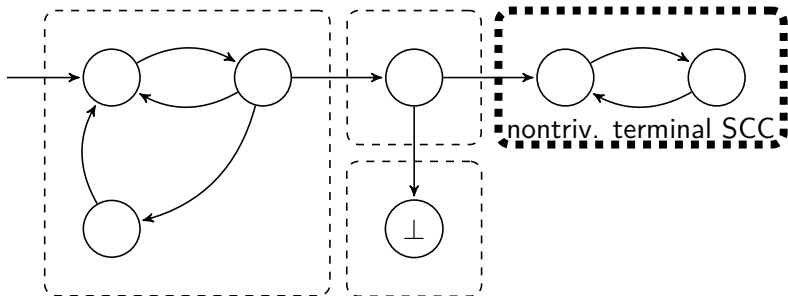
Local Nontermination

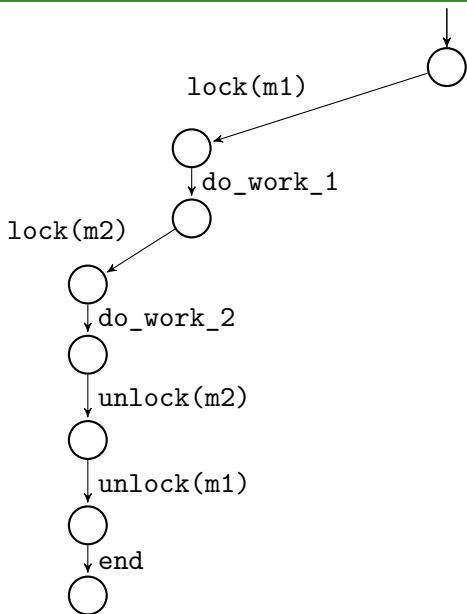
- a resource section does not terminate if the program can reach a point *in the resource section* from which it cannot reach *the corresponding resource section end*

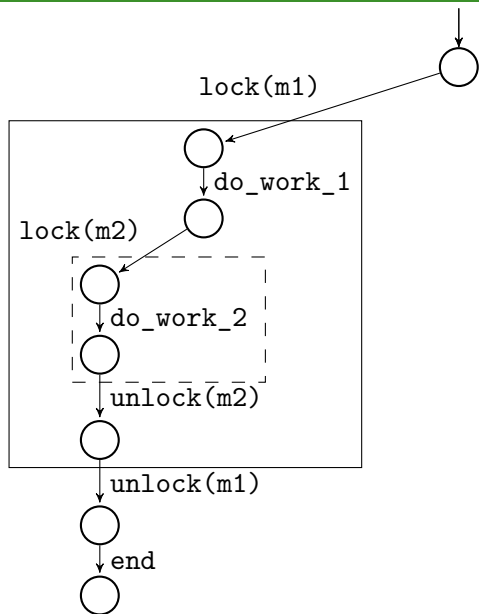


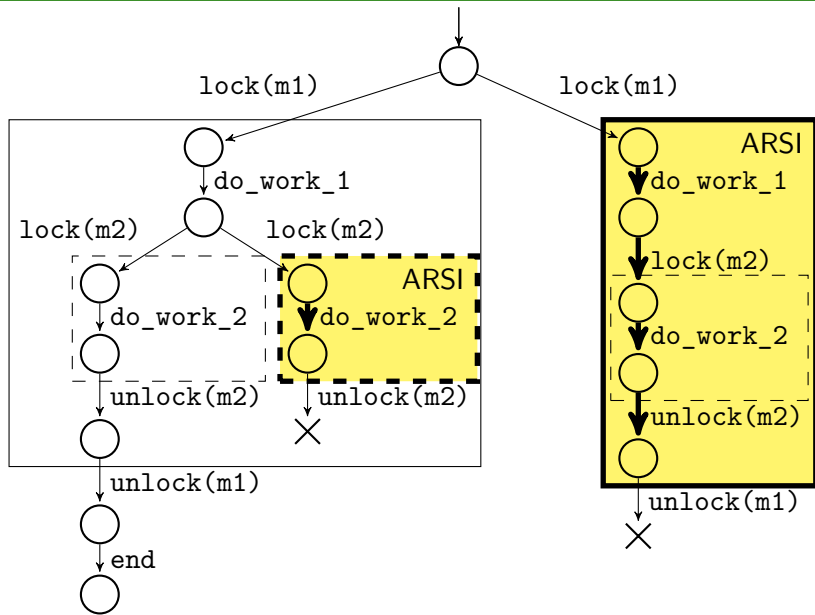
- a program does not terminate if it can reach a point from which it cannot reach its end

- a program does not terminate if it can reach a point from which it cannot reach its end
- detect nontrivial terminal strongly connected components





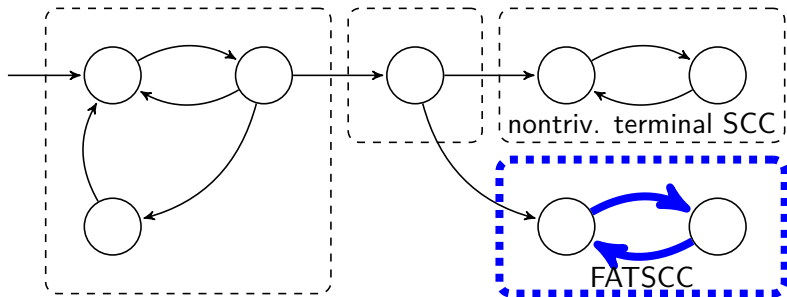






- a resource section does not terminate if the program can reach a point in the section from which it cannot reach the corresponding resource section end

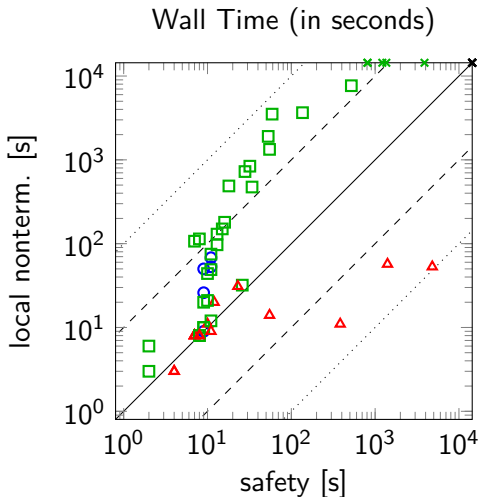
- a resource section does not terminate if the program can reach a point in the section from which it cannot reach the corresponding resource section end
- mark edges in ARSIs as **accepting**
- detect **fully accepting** terminal strongly connected components (**FATSCC**)





- modified Tarjan's algorithm for SCC decomposition: $\mathcal{O}(|G|)$
- global nontermination has no overhead
- for local nontermination the graph can get bigger

- modified Tarjan's algorithm for SCC decomposition: $\mathcal{O}(|G|)$
- global nontermination has no overhead
- for local nontermination the graph can get bigger





Source of resource sections

- either built-in (mutexes, condition variables, thread joining, ...)
- or user-provided (in source code; block of code, function end, ...)



Source of resource sections

- either built-in (mutexes, condition variables, thread joining, ...)
- or user-provided (in source code; block of code, function end, ...)

Summary

- allows detecting bugs not captured by safety (or LTL/CTL*) analysis
- built on explicit-state model checking → finite state space required
- works also on programs which do not terminate



Source of resource sections

- either built-in (mutexes, condition variables, thread joining, ...)
- or user-provided (in source code; block of code, function end, ...)

Summary

- allows detecting bugs not captured by safety (or LTL/CTL*) analysis
- built on explicit-state model checking → finite state space required
- works also on programs which do not terminate

Limitations

- works only with deterministic threads
- does not work with relaxed memory

Published on SEFM 2019 [ŠB19].

```
atomic< bool > flag = false;
void t1() {
    while ( true ) {
        x = read_input();
        if ( x < 0 )
            flag = true;
    }
}
void t2() {
    while ( !flag ) { } /* wait */
}
```

Does *wait* end?



```
atomic< bool > flag = false;
void t1() {
    while ( true ) {
        x = read_input();
        if ( x < 0 )
            flag = true;
    }
}
void t2() {
    while ( !flag ) { } /* wait */
}
```

Does *wait* end? depends on input



```
atomic< bool > flag = false;
void t1() {
    while ( true ) {
        x = read_input();
        if ( x < 0 )
            flag = true;
    }
}
void t2() {
    while ( !flag ) { } /* wait */
}
```

Does *wait* end? depends on input

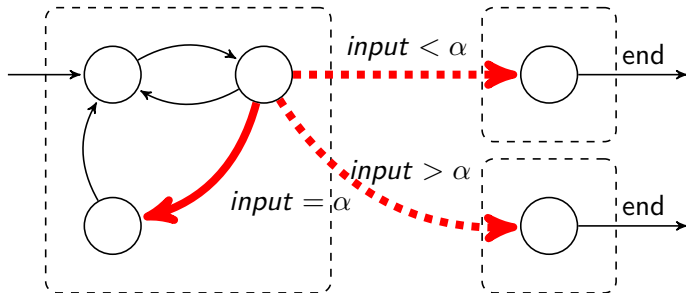
- for user, network or similar inputs we cannot expect that any outcome will eventually happen



```
atomic< bool > flag = false;
void t1() {
    while ( true ) {
        x = read_input();
        if ( x < 0 )
            flag = true;
    }
}
void t2() {
    while ( !flag ) { } /* wait */
}
```

Does *wait* end? depends on input

- for user, network or similar inputs we cannot expect that any outcome will eventually happen
- want to report problems like this



- suppose the choice of red edges depends on user input
- need to disregard outgoing edges bound to the same choice if there is at least one staying in the SCC



- up to this point verification was precise



- up to this point verification was precise

with x86-TSO:

- the x86-TSO memory model is an *over-approximation* of behaviour of the real processor
 - it can delay write arbitrarily

- up to this point verification was precise

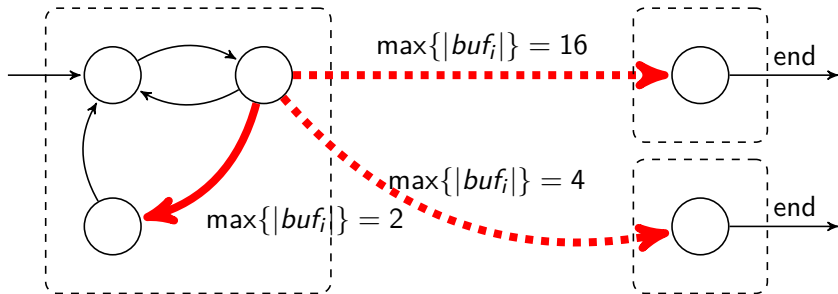
with x86-TSO:

- the x86-TSO memory model is an *over-approximation* of behaviour of the real processor
 - it can delay write arbitrarily
- the implementation is *under-approximation* of x86-TSO
 - as the buffer is bounded
 - we do not know if it is over-approximation or under-approximation of the real hardware

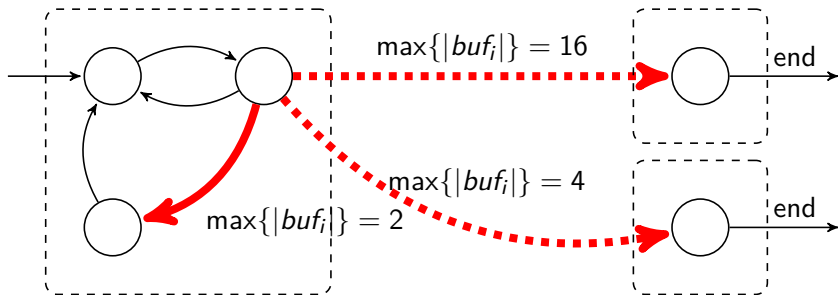
- up to this point verification was precise

with x86-TSO:

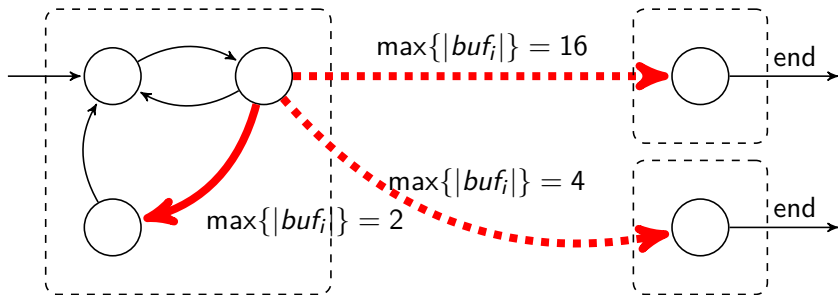
- the x86-TSO memory model is an *over-approximation* of behaviour of the real processor
 - it can delay write arbitrarily
- the implementation is *under-approximation* of x86-TSO
 - as the buffer is bounded
 - we do not know if it is over-approximation or under-approximation of the real hardware
- over-approximation while searching for an end of a resource section is a problem



- maybe report sections as conditionally nonterminating if more relaxed behaviour is required to leave them then stay



- maybe report sections as conditionally nonterminating if more relaxed behaviour is required to leave them then stay
- possibly report only those that can be reached with less relaxed behaviour then is required to reach the point of no return



- maybe report sections as conditionally nonterminating if more relaxed behaviour is required to leave them then stay
- possibly report only those that can be reached with less relaxed behaviour then is required to reach the point of no return
- problem: buffer size is not a good measure of degree of relaxation



Vladimír Štill and Jiří Barnat. “Local Nontermination Detection for Parallel C++ Programs”. In: *Software Engineering and Formal Methods*. <https://divine.fi.muni.cz/2019/Interm>. Cham: Springer International Publishing, 2019, pp. 373–390. DOI: 10.1007/978-3-030-30446-1_20.