Local Nontermination Detection for Parallel C++ Programs

Vladimír Štill Jiří Barnat

ParaDiSe
Parallel & Distributed Systems Laboratory

Masaryk University

Brno, Czech Republic

20th Septempler 2019

Motivation

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

Motivation

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

DEMO: DIVINE

Motivation

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

DEMO: DIVINE

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

Safety Checking Parallel Programs

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

Safety Checking Parallel Programs

- 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

Safety Checking Parallel Programs

- 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)

Safety Checking Parallel Programs

- 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

(Non)Termination Checking

- check that the whole program terminates

(Non)Termination Checking

- check that the whole program terminates

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

Local Nontermination Detection for Parallel Programs

- we aim at nontermination caused by unintended parallel interactions

Local Nontermination Detection for Parallel Programs

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

Local Nontermination Detection for Parallel Programs

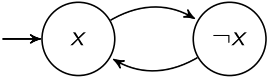
- we aim at nontermination caused by unintended parallel interactions
- not at complex control flow & loops
- should be easy to specify
- should not report nontermination spuriously
- should be useful for analysis of services/servers

Local Nontermination Detection for Parallel Programs

- 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

- build on explicit-state model checking → 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;
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;
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;
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]`$^{\omega}$ (repeats infinitely)

What is Nontermination?

```
atomic< bool > spin_lock;
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]`$^{\omega}$ (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?

$$\texttt{[t0: spin\_lock.exchange(true)} \to \texttt{false]}$$
$$\Big(\texttt{[t1: spin\_lock.exchange(true)} \to \texttt{true]}$$
$$\texttt{[t0: x++]}$$
$$\texttt{[t0: spin\_lock = false]}$$
$$\texttt{[t0: spin\_lock.exchange(true)} \to \texttt{false]}\Big)^{\omega}$$

- 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?

Nontermation

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

What is Nontermination?

Nontermation

- 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?

Nontermation

- 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 Nontermation

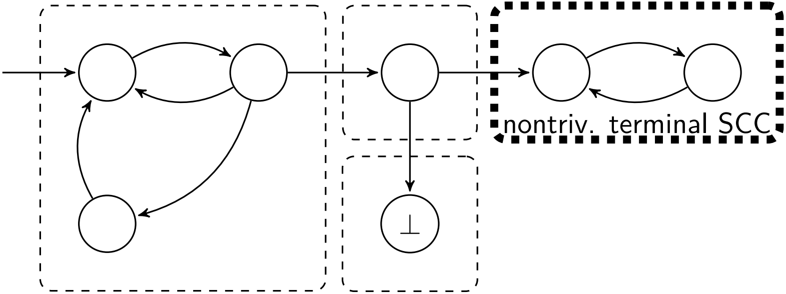
- a resource section does not terminate if the program can reach a point *in the resource section* from which it cannot reach *the corresponding resource section end*

Detecting Nontermination

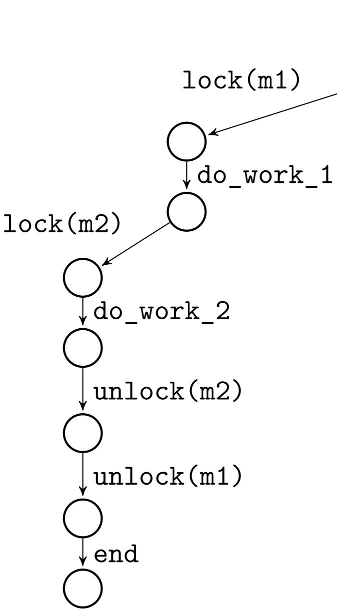
- a program does not terminate if it can reach a point from which it cannot reach its end

Detecting Nontermination

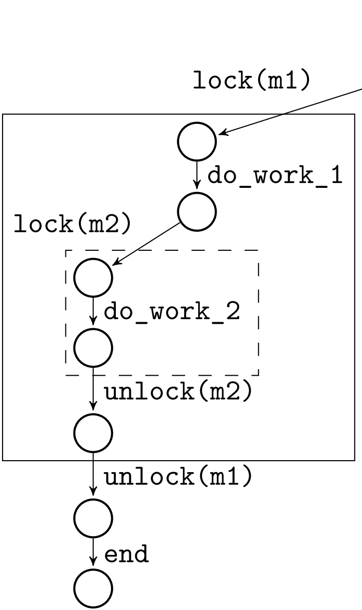
- a program does not terminate if it can reach a point from which it cannot reach its end
- detect nontrivial terminal strongly connected components

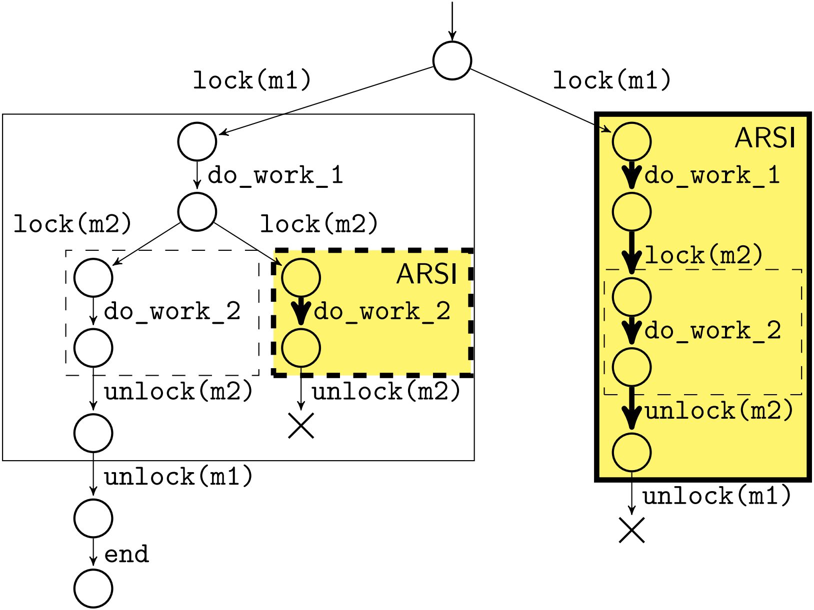


Going **Local**: Active Resource Section Instances



Going **Local**: Active Resource Section Instances



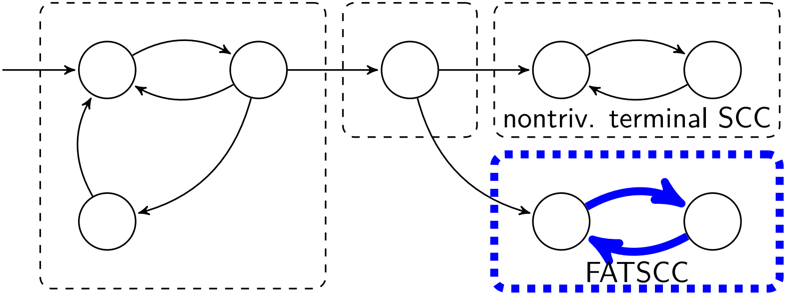

lock(m1)
lock(m1)
ARSI
do_work_1
do_work_1
lock(m2)
lock(m2)
lock(m2)
ARSI
do_work_2
do_work_2
do_work_2
unlock(m2)
unlock(m2)
unlock(m2)
unlock(m1)
unlock(m1)
end

Detecting **Local** Nontermination

- 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

Detecting **Local** Nontermination

- 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**)

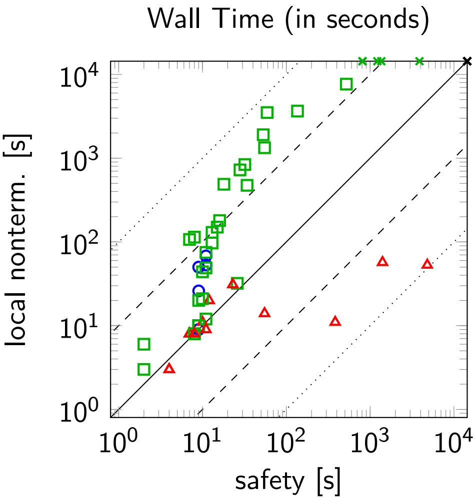


Detection Algorithm

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

Detection Algorithm

- 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 resourcre sections

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

Resource Sections & Conclusions

Source of resourcre sections

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

Conclusion

- we have presented a novel technique which allows detecting bugs not captured by safety (or LTL/CTL*) analysis

Resource Sections & Conclusions

Source of resourcre sections

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

Conclusion

- we have presented a novel technique which allows detecting bugs not captured by safety (or LTL/CTL*) analysis
- built on explicit-state model checking $\rightarrow$ finite state space required

Resource Sections & Conclusions

Source of resourcre sections

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

Conclusion

- we have presented a novel technique which allows detecting bugs not captured by safety (or LTL/CTL*) analysis
- built on explicit-state model checking $\rightarrow$ finite state space required
- works also on programs which do not terminate

Resource Sections & Conclusions

Source of resourcre sections

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

Conclusion

- we have presented a novel technique which 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
- open-source implementation

Resource Sections & Conclusions

Source of resourcre sections

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

Conclusion

- we have presented a novel technique which 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
- open-source implementation
- performance is underwhelming, but it can detect new class of bugs
- `https://divine.fi.muni.cz/2019/lnterm/`