Distributed and Predictable Software Model Checking

Nuno Lopes
INESC-ID / IST - TU Lisbon

Andrey Rybalchenko
TU Munich

Motivation

Motivation

- Software model checking can take weeks to execute

Motivation

- Software model checking can take weeks to execute
- Software changes every day

Motivation

- Software model checking can take weeks to execute
- Software changes every day

- So.. we need faster Model Checkers!

Outline

- Overview of Software Model Checking
- Example of Sequential Algorithm
- Why *Predictable* Model Checking?
- Algorithm
- Evaluation
- Conclusions

Overview of Software Model Checking

- Given a program and a property, we want to verify that the property always holds in the program for all possible inputs

Overview of Software Model Checking

- Given a program and a property, we want to verify that the property always holds in the program for all possible inputs
- e.g., there are no buffer overflows, assert() always holds, etc..

Overview of CEGAR w/ Predicate Abst.

- Compute over-approximation of reachable states (w/ e.g. a BFS)
- Stop when the error state is found

Overview of CEGAR w/ Predicate Abst.

- Compute over-approximation of reachable states (w/ e.g. a BFS)
- Stop when the error state is found

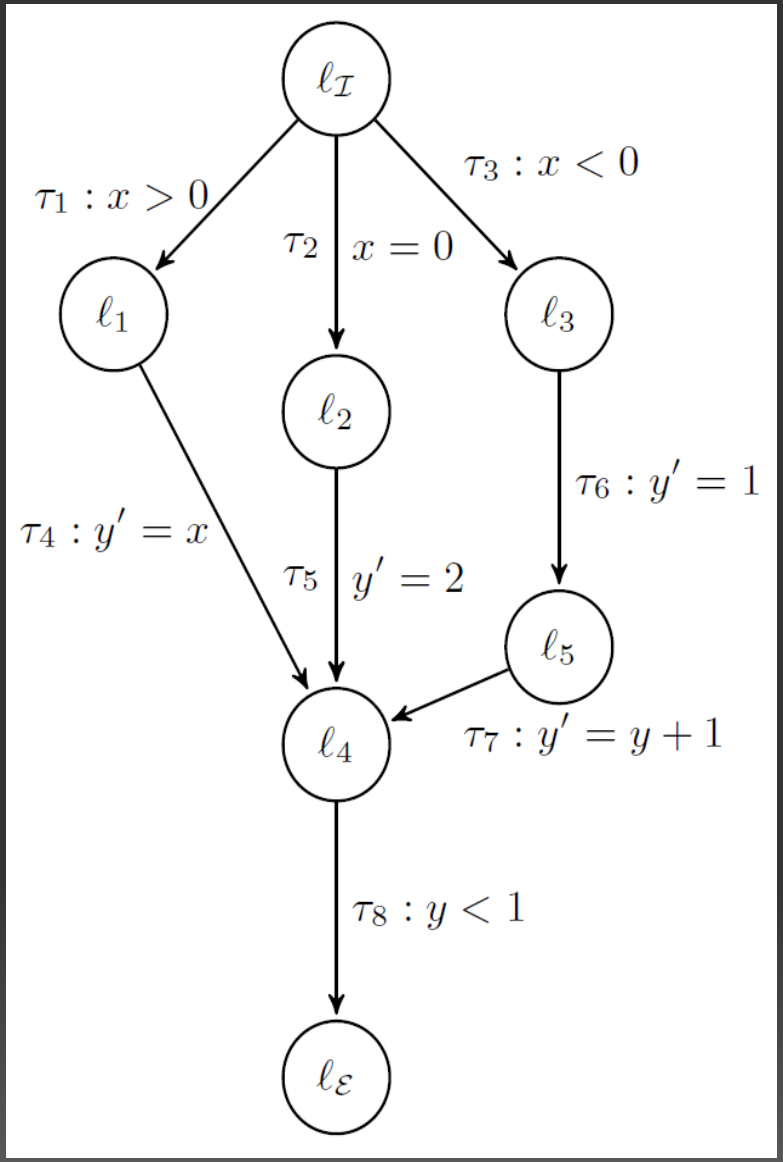
- CEGAR (CounterExample Guided Abstraction Refinement) loop:
  1. check if valid with current abstraction
  2. if not:
     - if counterexample path is feasible, exit
     - otherwise, refine the abstraction
  3. goto 1

Example

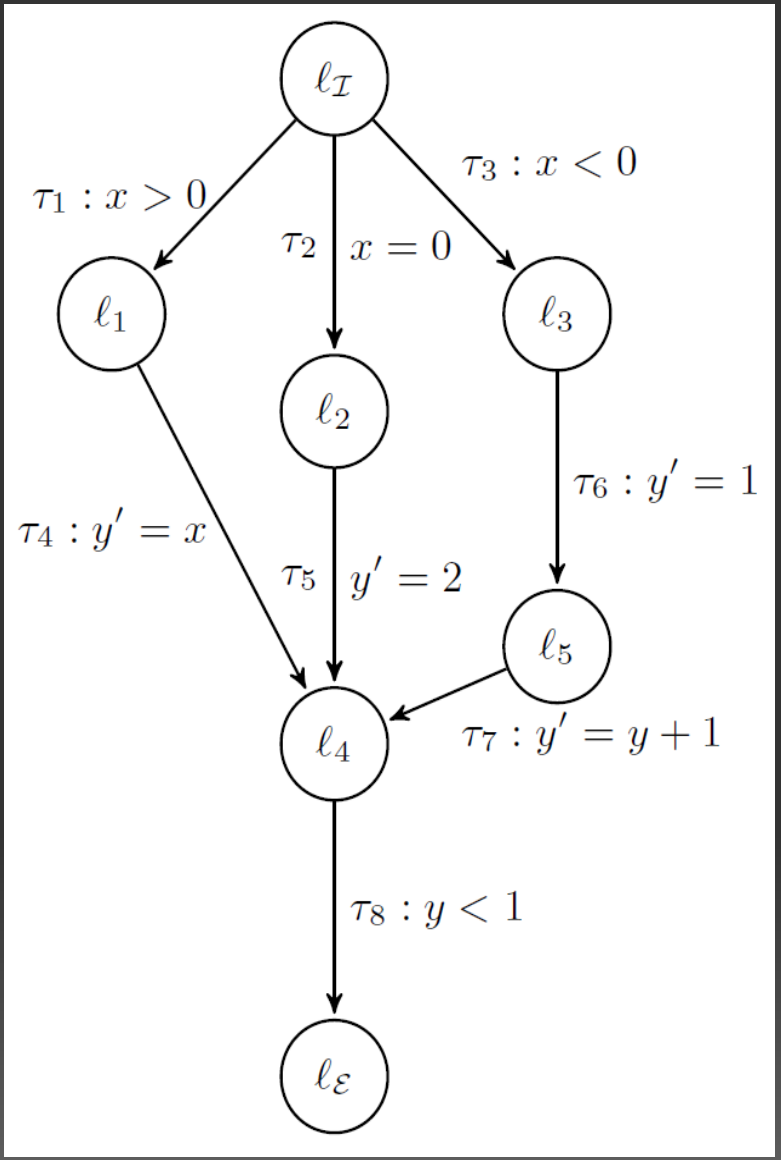
```
if (x > 0) {
    y = x;
} else if (x == 0) {
    y = 2;
} else {
    y = 1;
    y = y + 1;
}

assert( y >= 1 );
```

Example

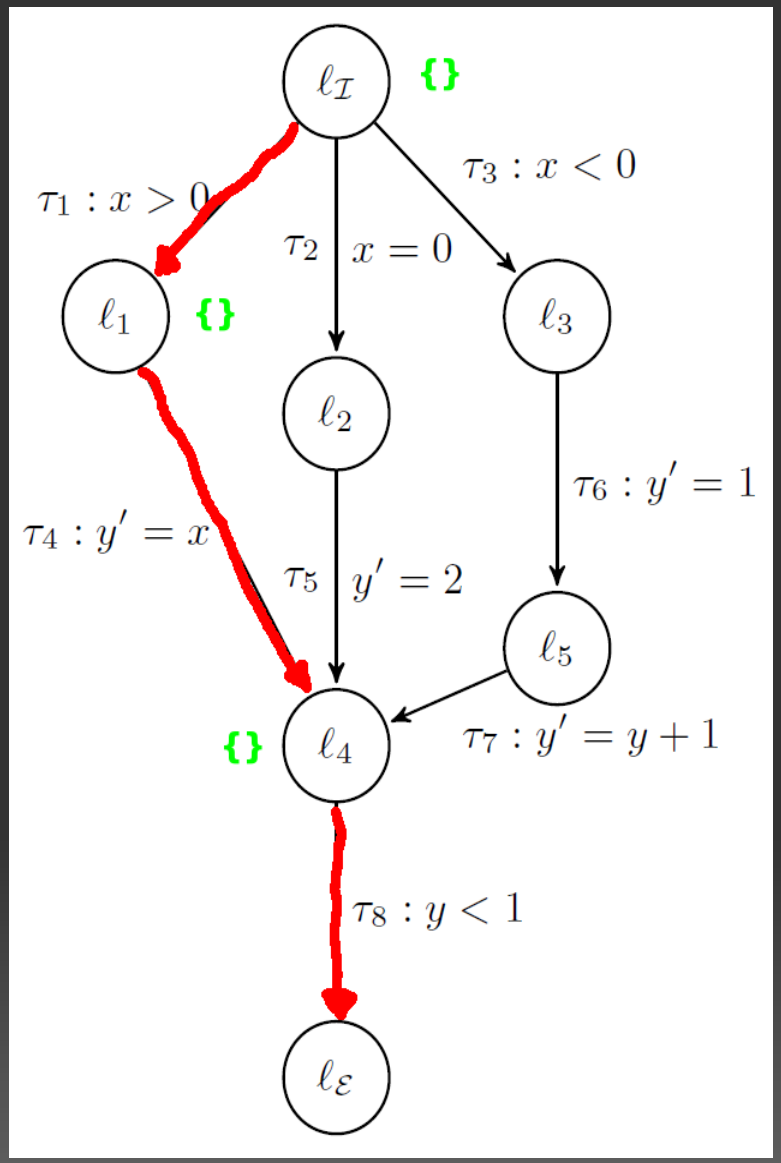
```
if (x > 0) {
    y = x;
} else if (x == 0) {
    y = 2;
} else {
    y = 1;
    y = y + 1;
}

assert( y >= 1 );
```



Example: 1st iteration

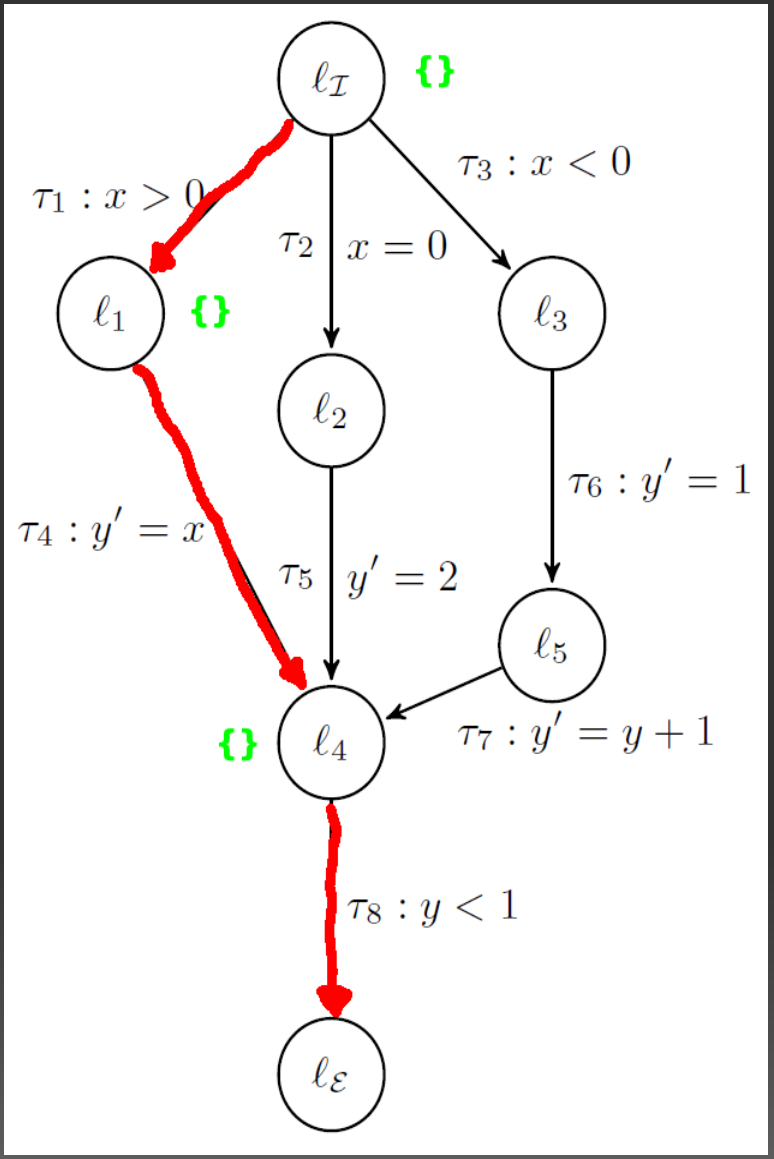


Example: 1st iteration



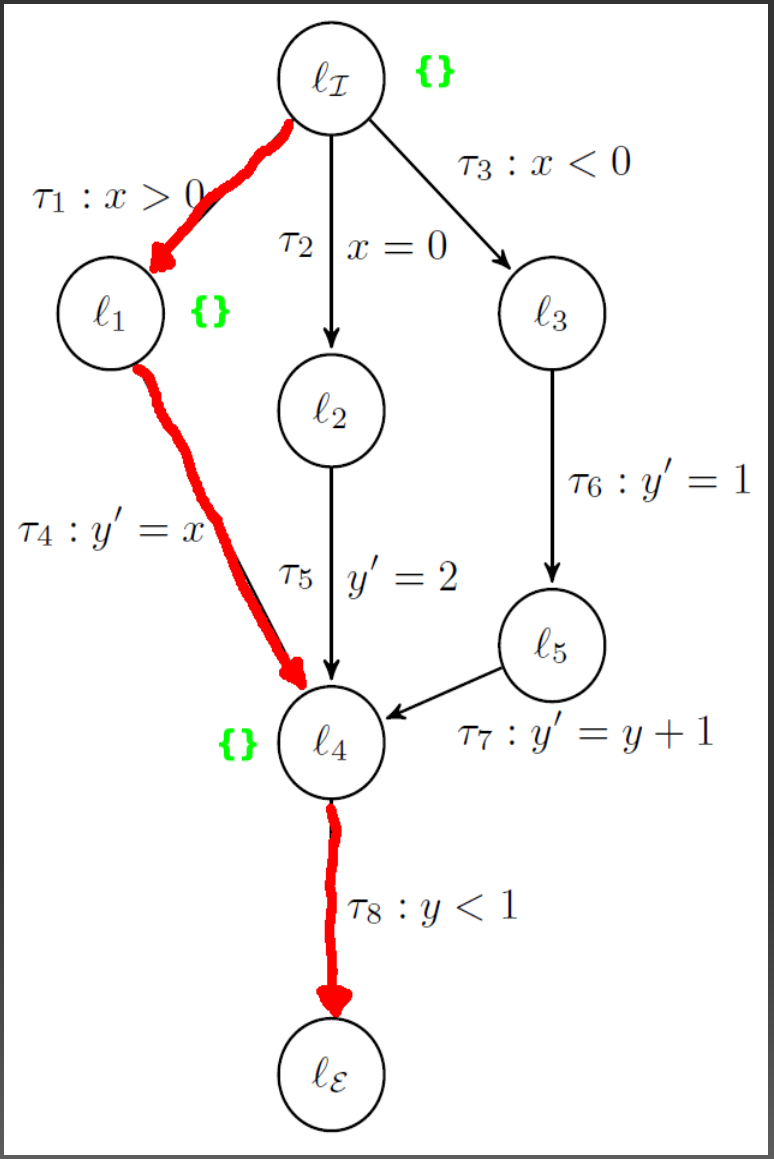
Predicate Abstraction:
P = {}

Example: 1st iteration



error is reachable because:
true ∧ y < 1 is SAT

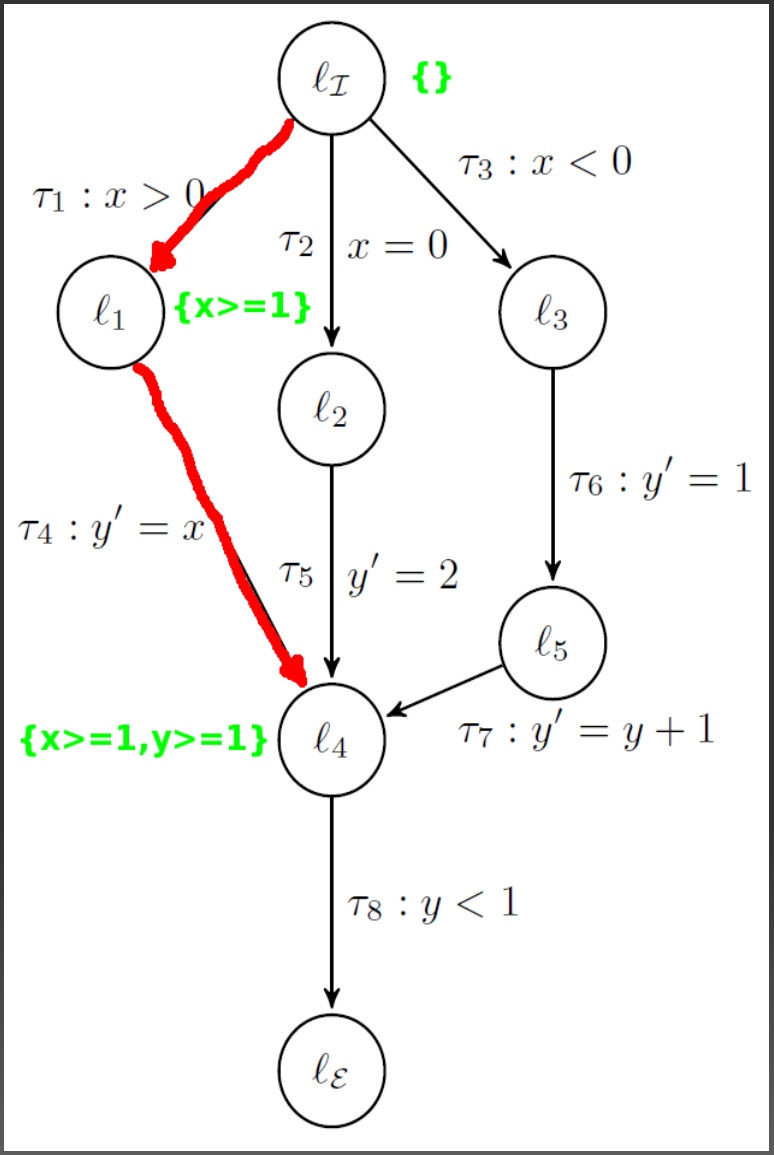
Example: 1st iteration



error is reachable because:
true ∧ y < 1 is SAT

Using interpolation we derive:
P = {x >= 1, y >= 1}

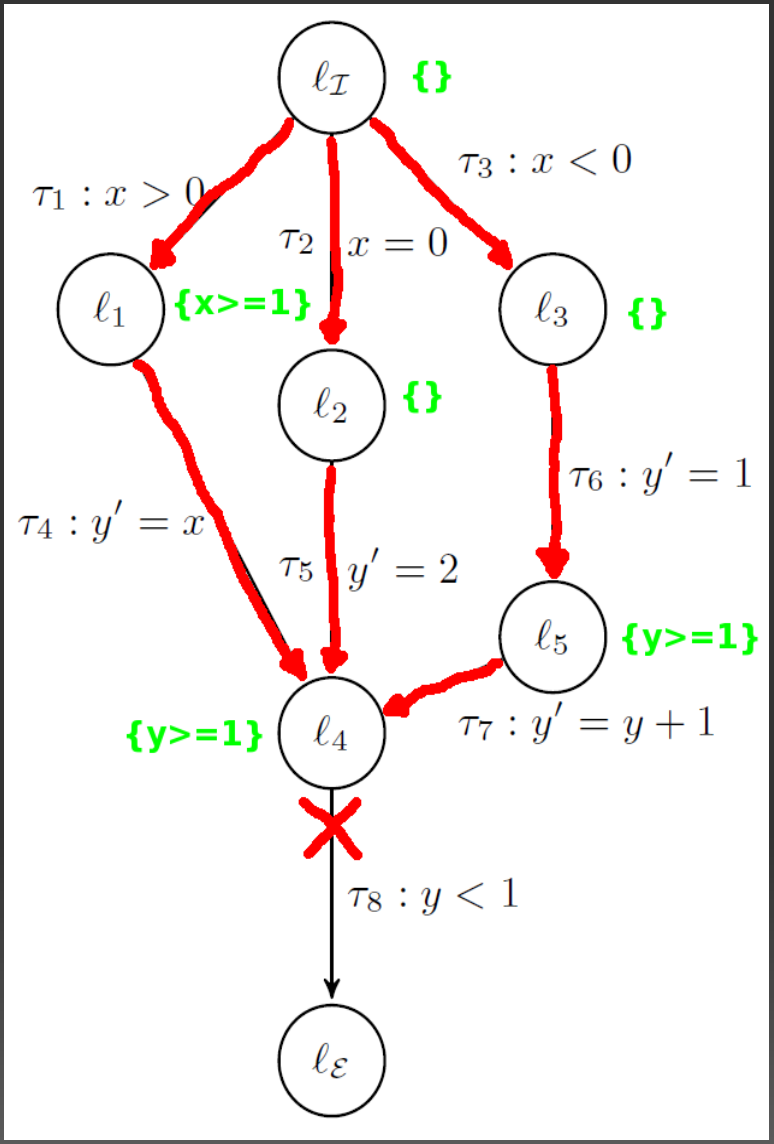
Example: 2nd iteration



P = {x >= 1, y >= 1}

The error state is not reachable from this path anymore because:
x >= 1 ∧ y >= 1 ∧ y < 1 <-> false

Example: 2nd iteration

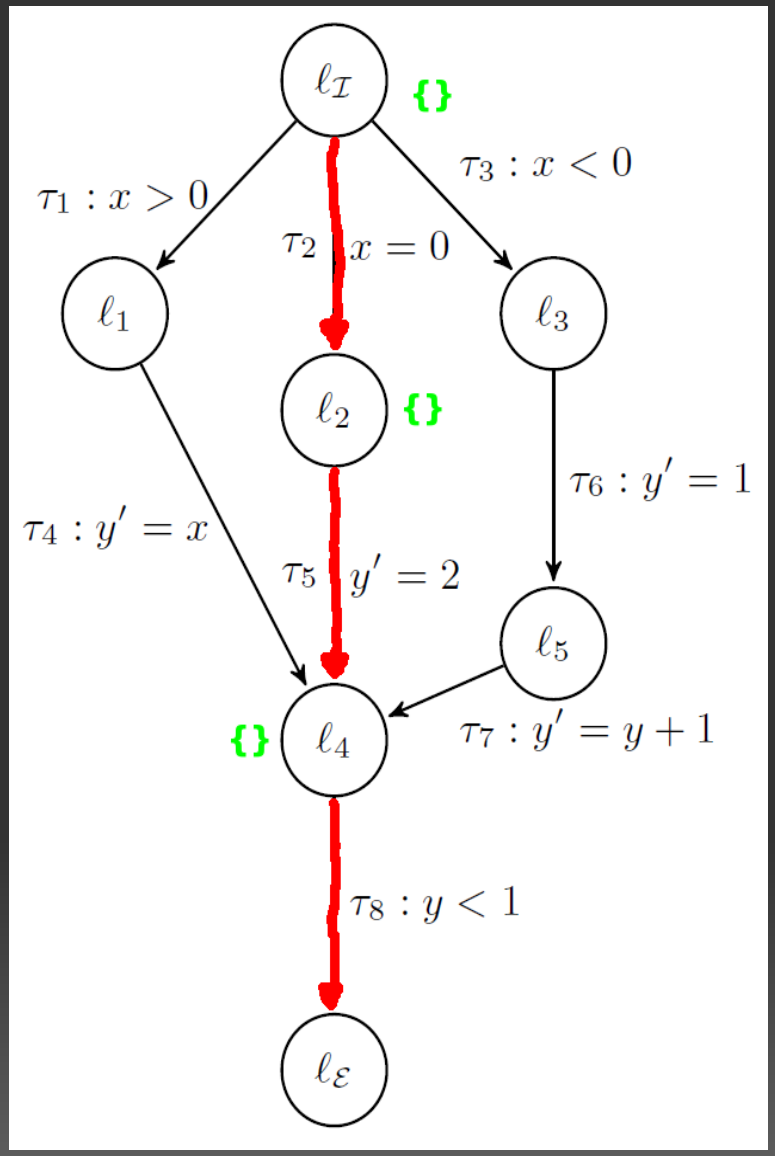


The error state is not reachable from **any** path anymore because:
y >= 1 ∧ y < 1 <-> false

P = {x >= 1, y >= 1}

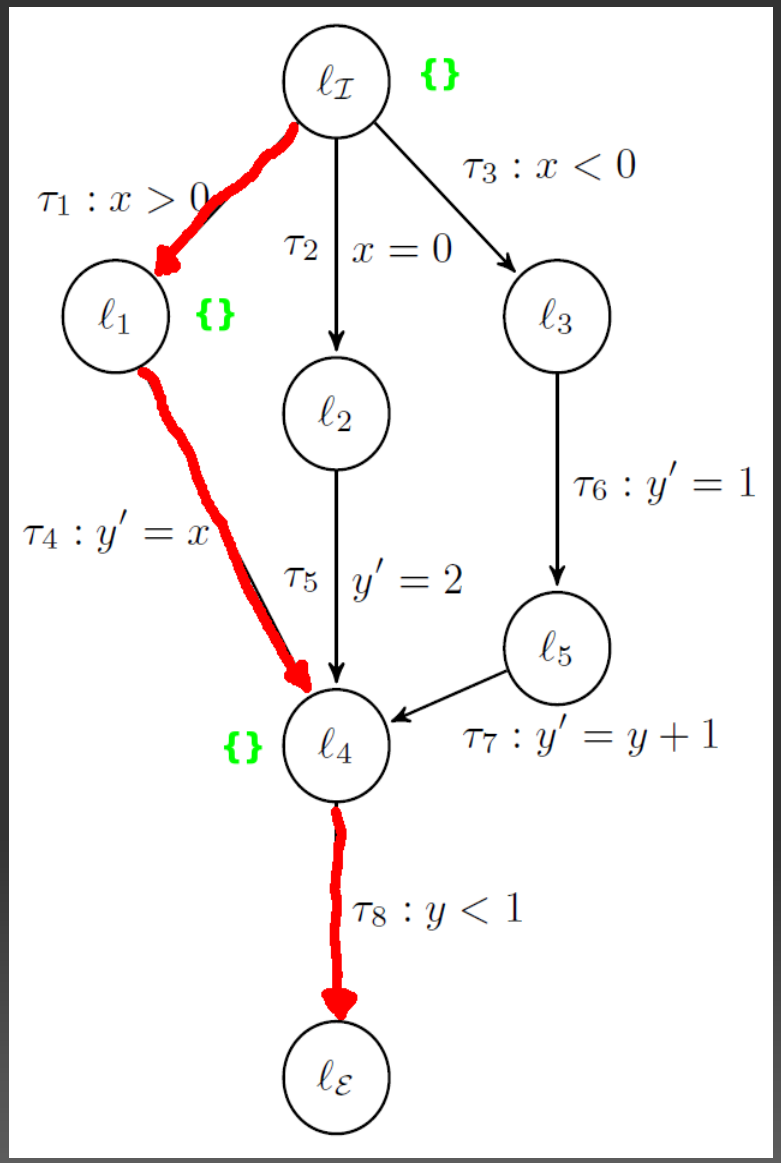
Why *Predictable* Model Checking?

Why *Predictable* Model Checking?



New abstraction:
P = { y >= 2 }

Predictability: 2nd iteration

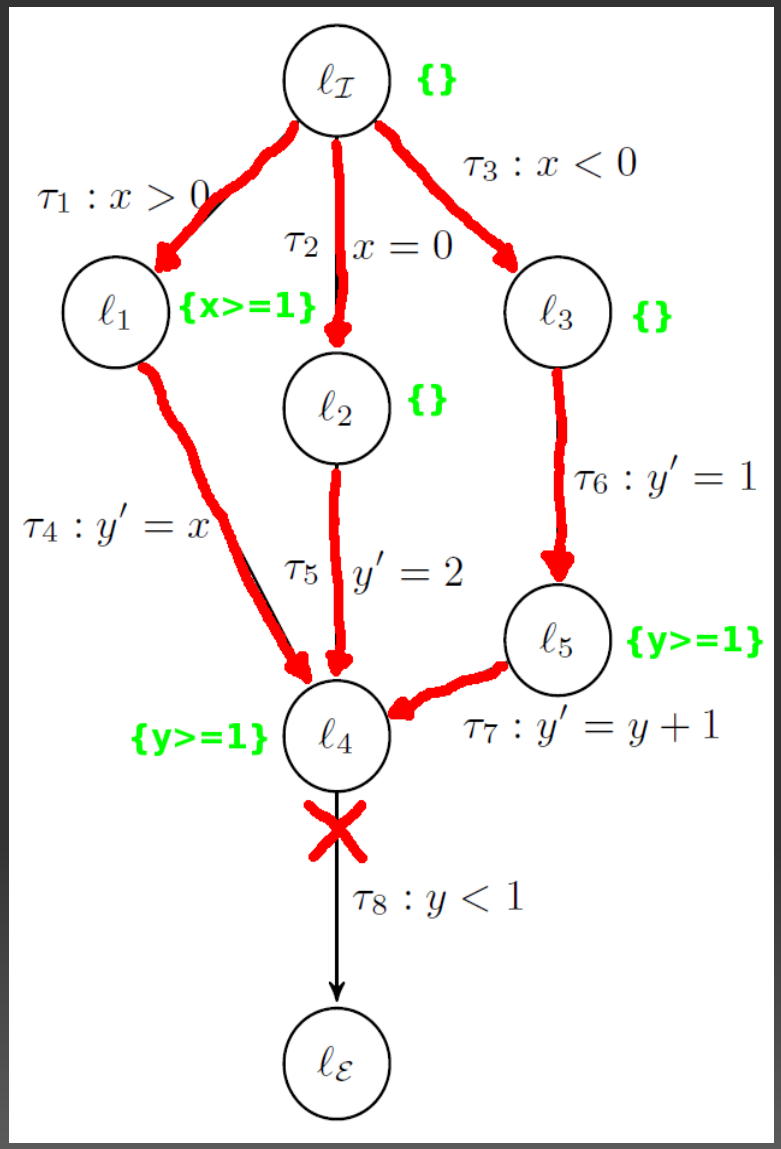


P = { y >= 2 }

error state is still reachable:
true ∧ y < 1 is SAT

Using interpolation we derive:
P' = {x >= 1, y >= 1}

Predictability: 3rd iteration



The error state is not reachable anymore because:
y >= 1 ∧ y < 1 <-> false

P = {y >= 2, x >= 1, y >= 1}

Why *Predictable* Model Checking?

In this simple example it is possible to do:

- 1 refinement (left first)
- 2 refinements (middle first, left second)
- 3 refinements (middle first, right second, left third)

Why *Predictable* Model Checking?

In this simple example it is possible to do:

- 1 refinement (left first)
- 2 refinements (middle first, left second)
- 3 refinements (middle first, right second, left third)

Running time varies accordingly:

- Best and worst executions can have 30x of difference
- Can be up to 2 times as slow as the sequential version

Solutions

Solutions

- Need to resolve the non-deterministic choice of counterexamples

Solutions

- Need to resolve the non-deterministic choice of counterexamples
- Synchronization is an option, but it's not desirable

Solutions

- Need to resolve the non-deterministic choice of counterexamples
- Synchronization is an option, but it's not desirable
- Need a way to reduce synchronization

Our solution

Our solution

- Compute the full tree until a certain depth

Our solution

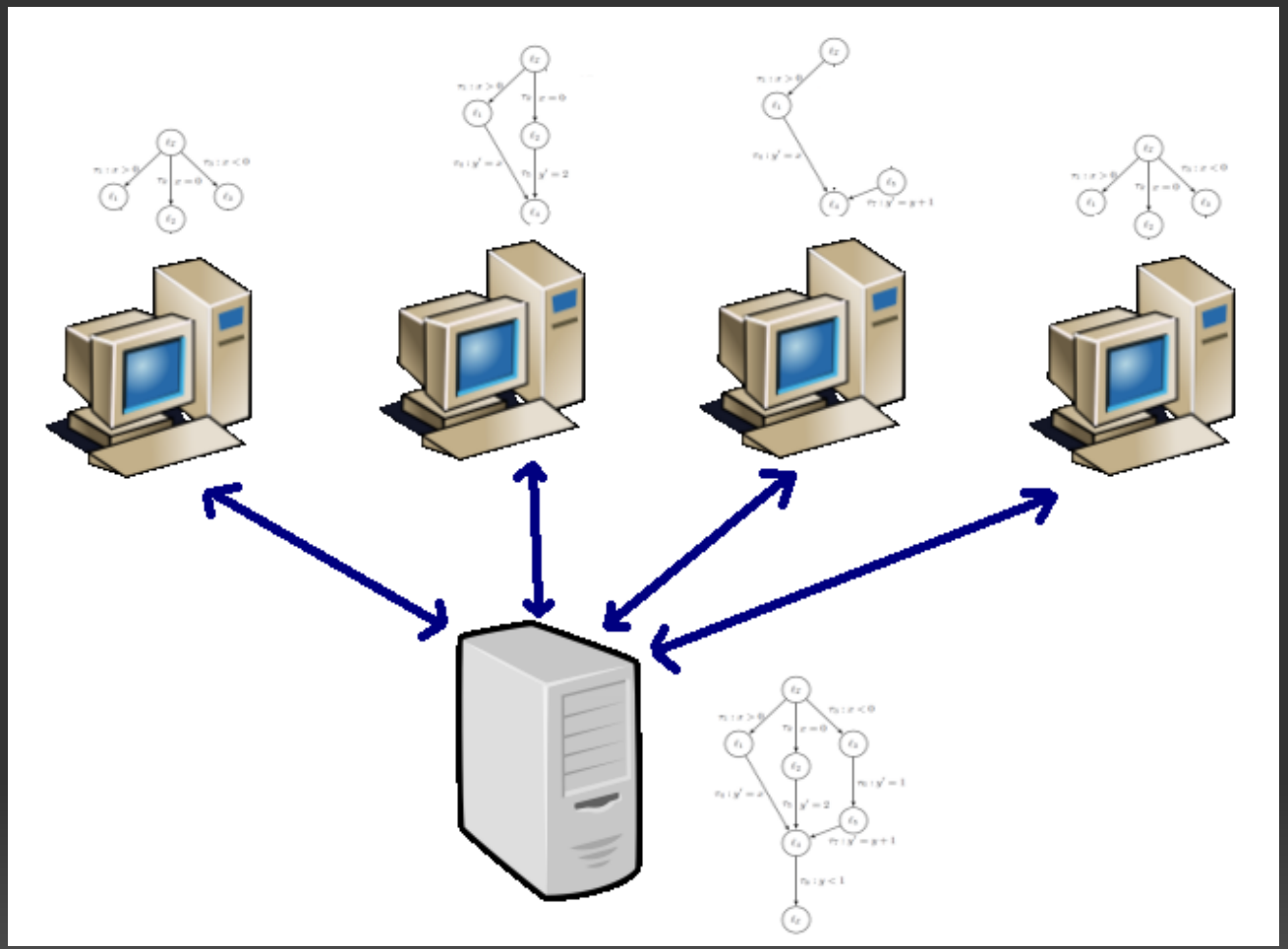
- Compute the full tree until a certain depth
- Refine a shortest counterexample (picked deterministically)

Our solution

- Compute the full tree until a certain depth
- Refine a shortest counterexample (picked deterministically)
- The overhead for computing the full tree is acceptable

Architecture

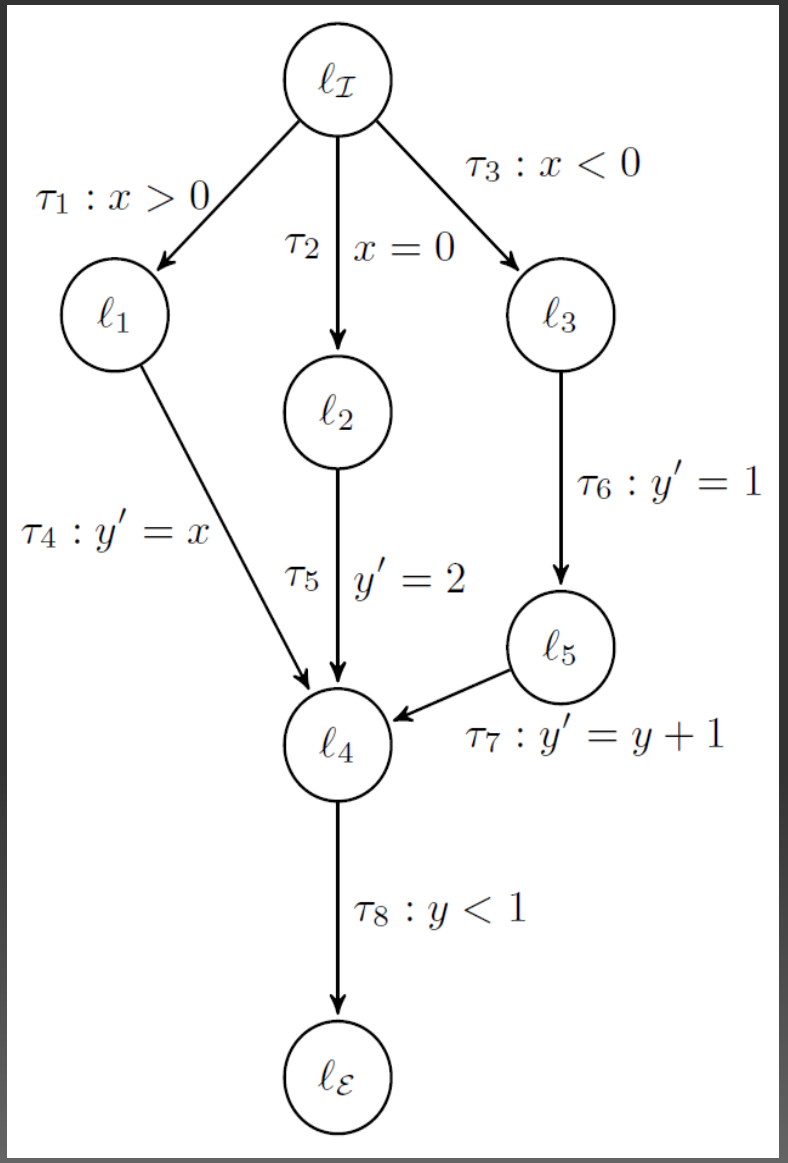
Architecture



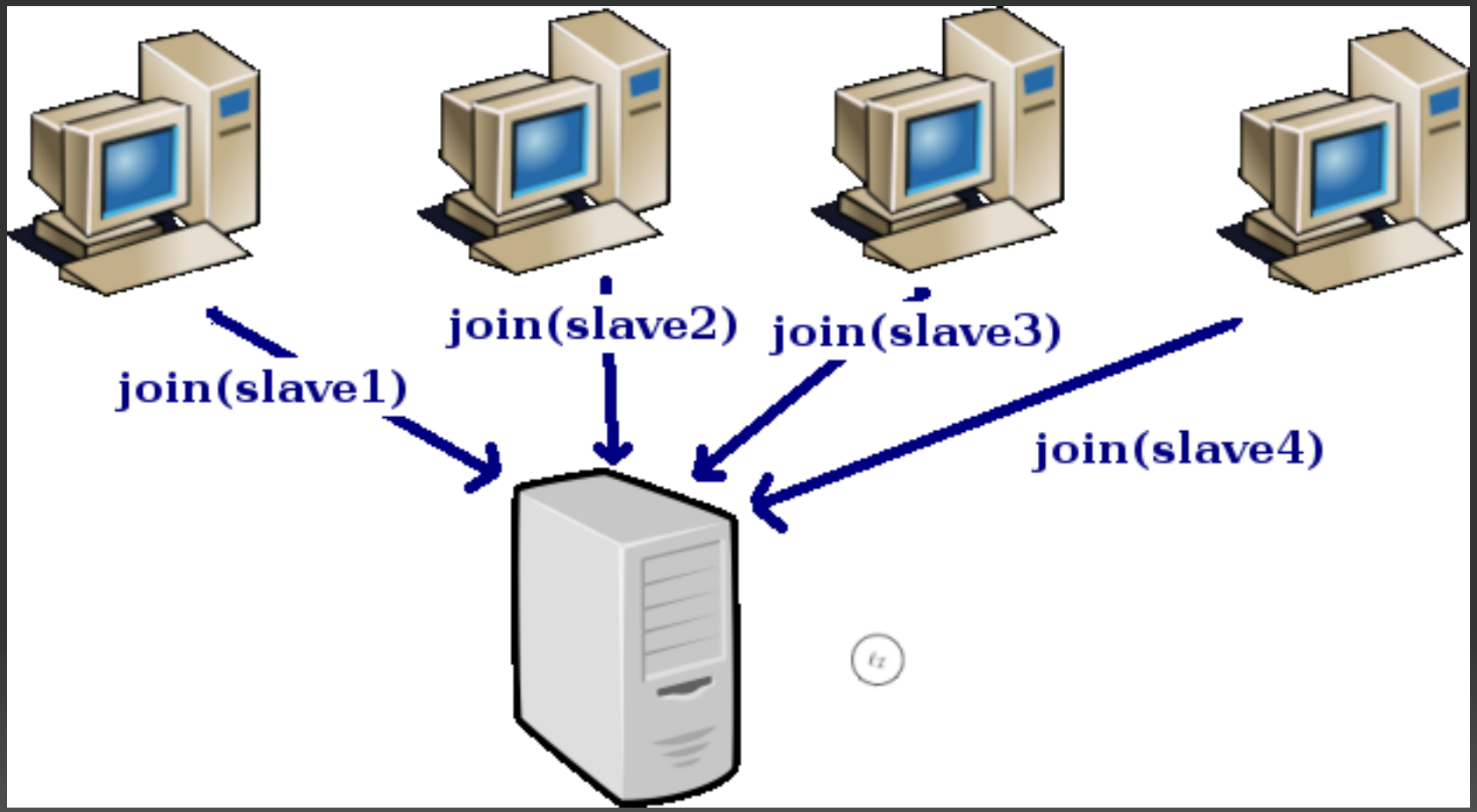
- Master-Slave
- Full tree in master
- Partial trees in slaves (cache)
- no communication between slaves
- work piece = state expansion

Example Distributed

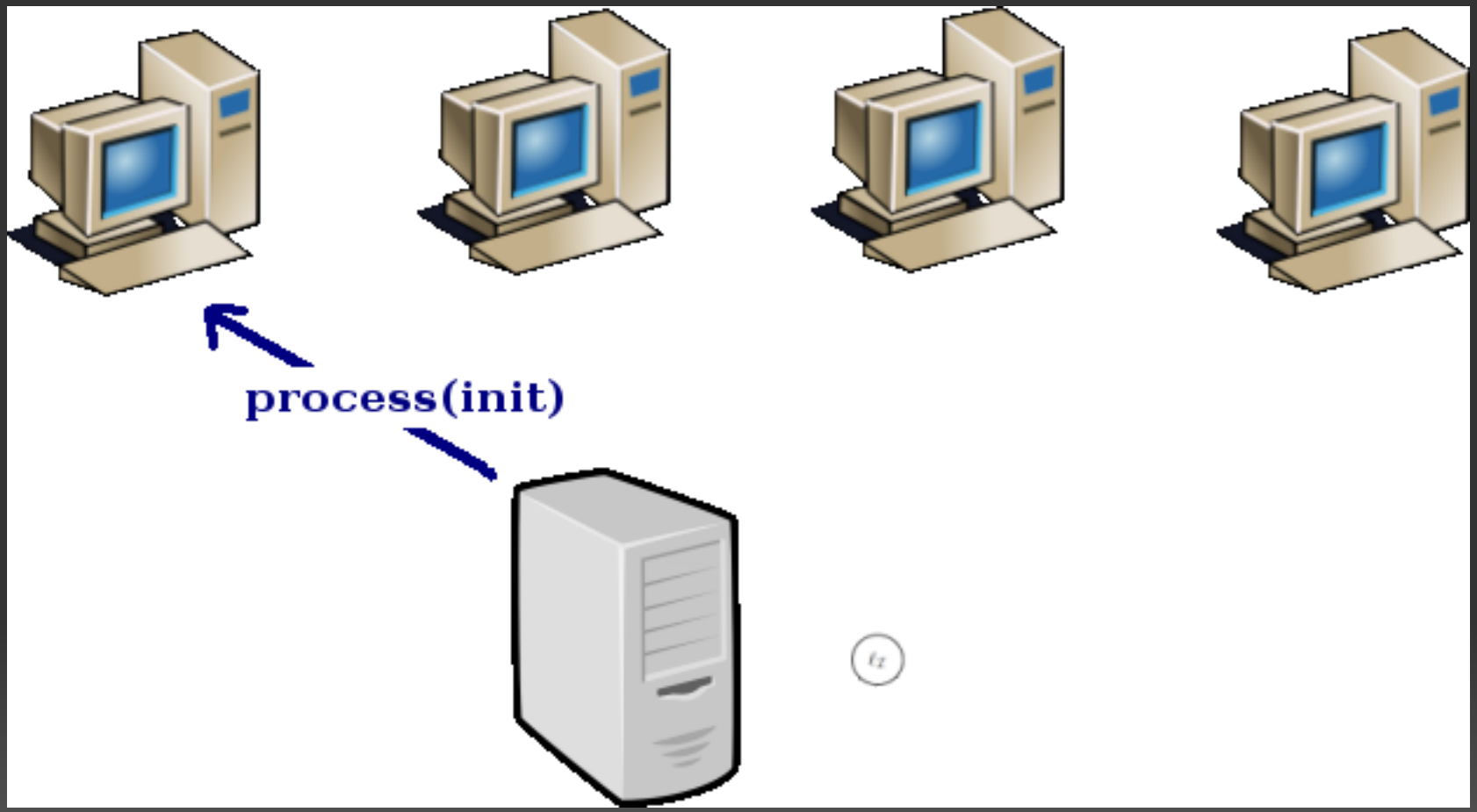
Example Distributed



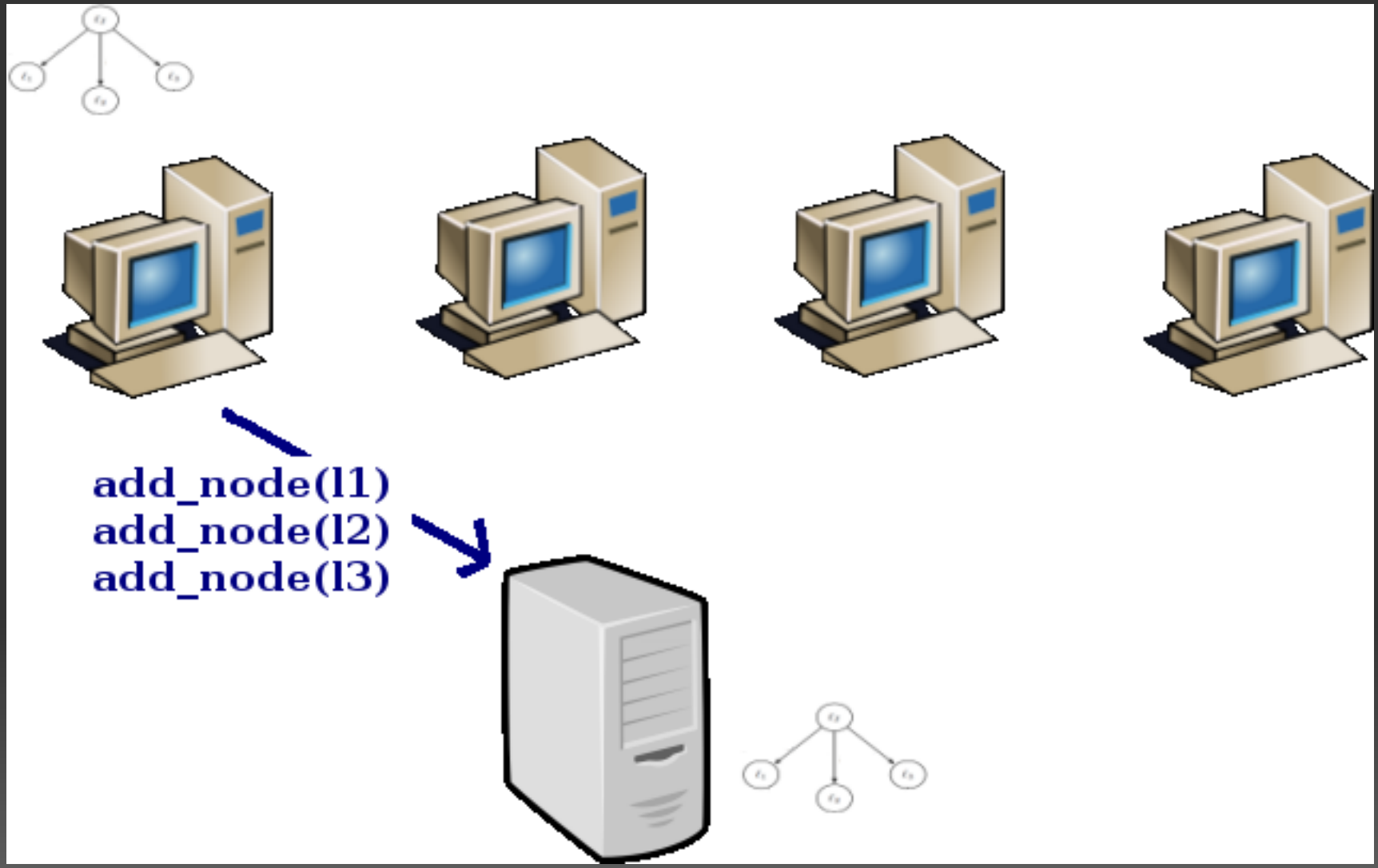
Example Distributed - #1



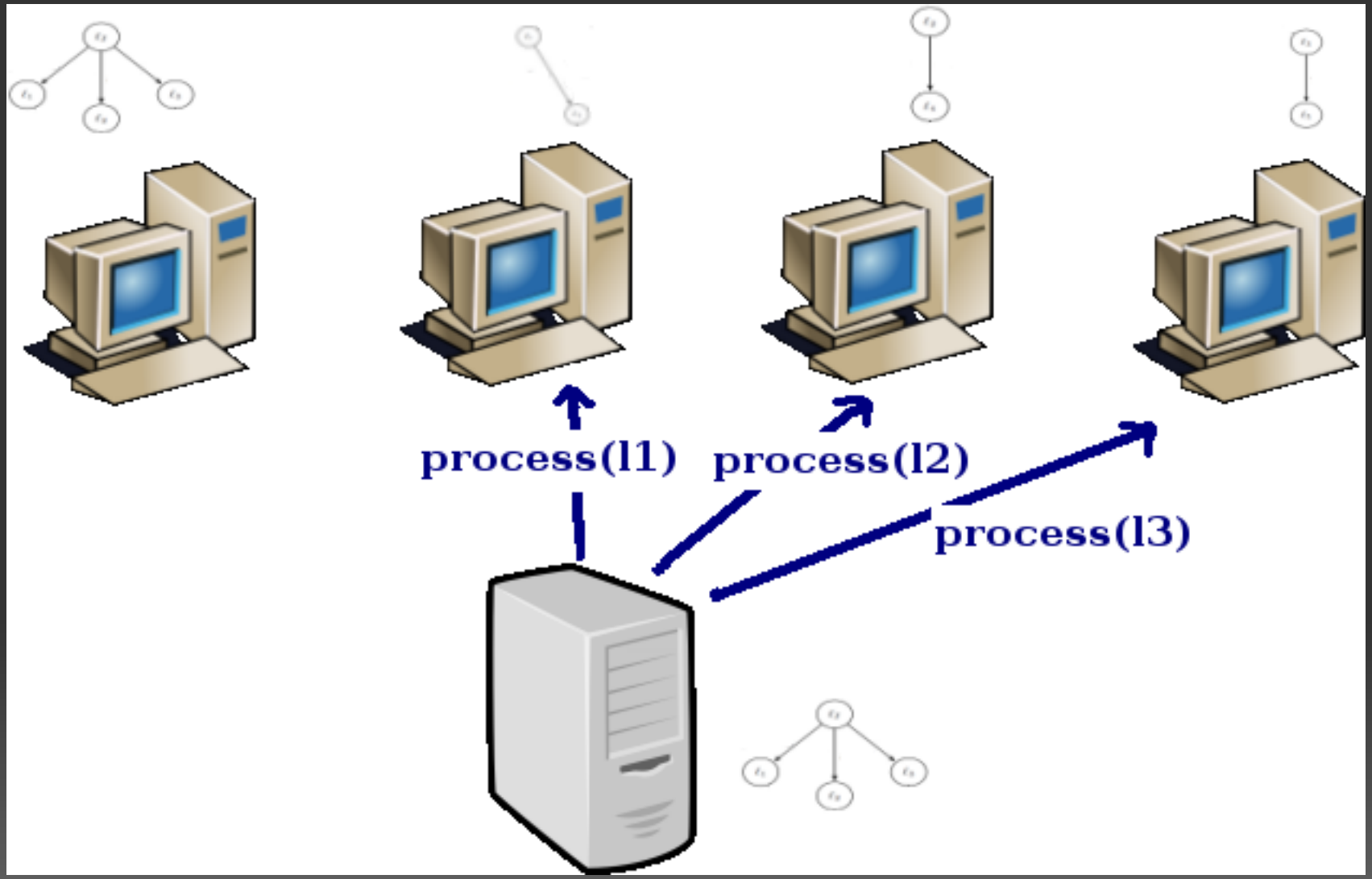
Example Distributed - #2



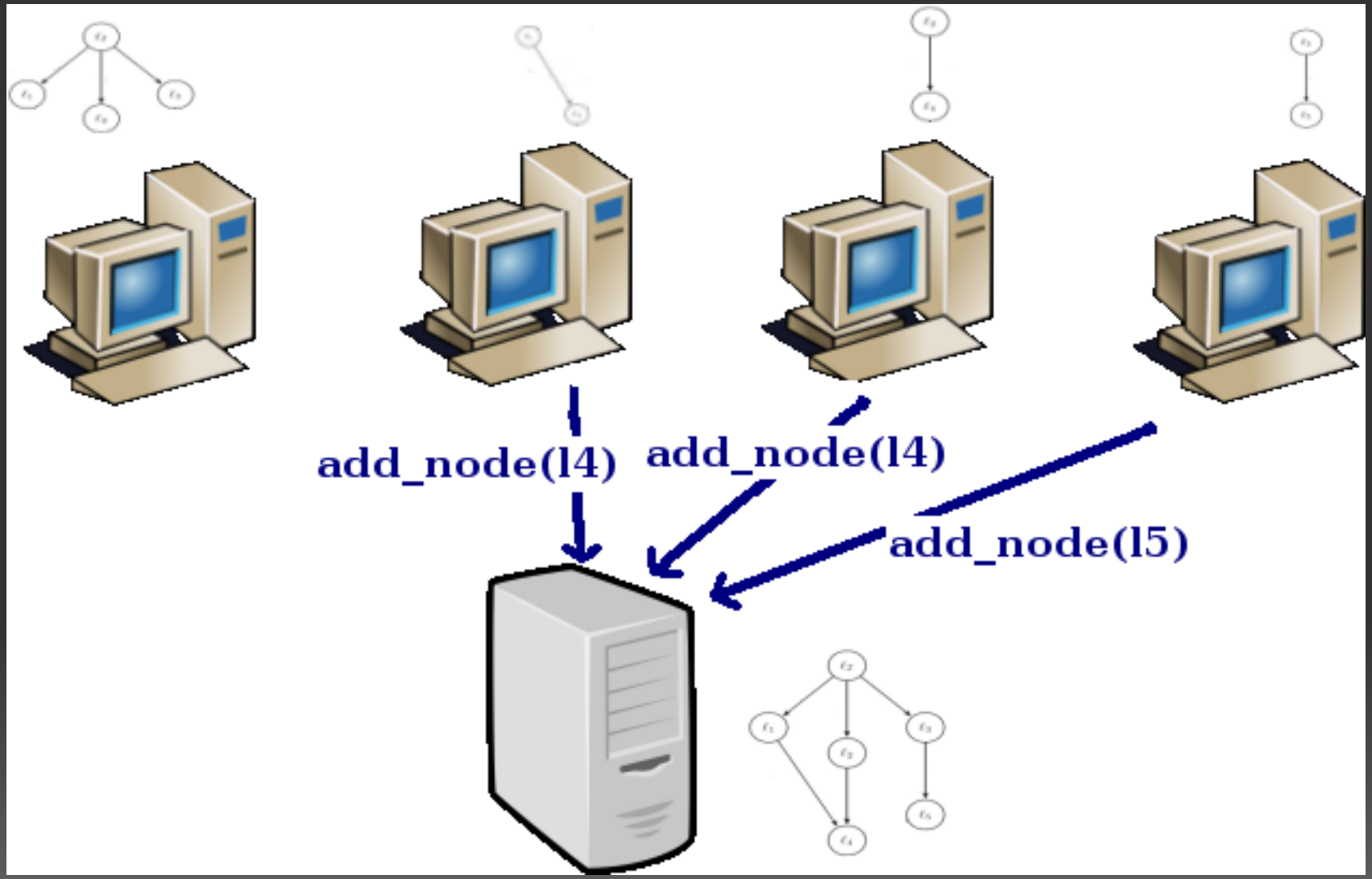
Example Distributed - #3



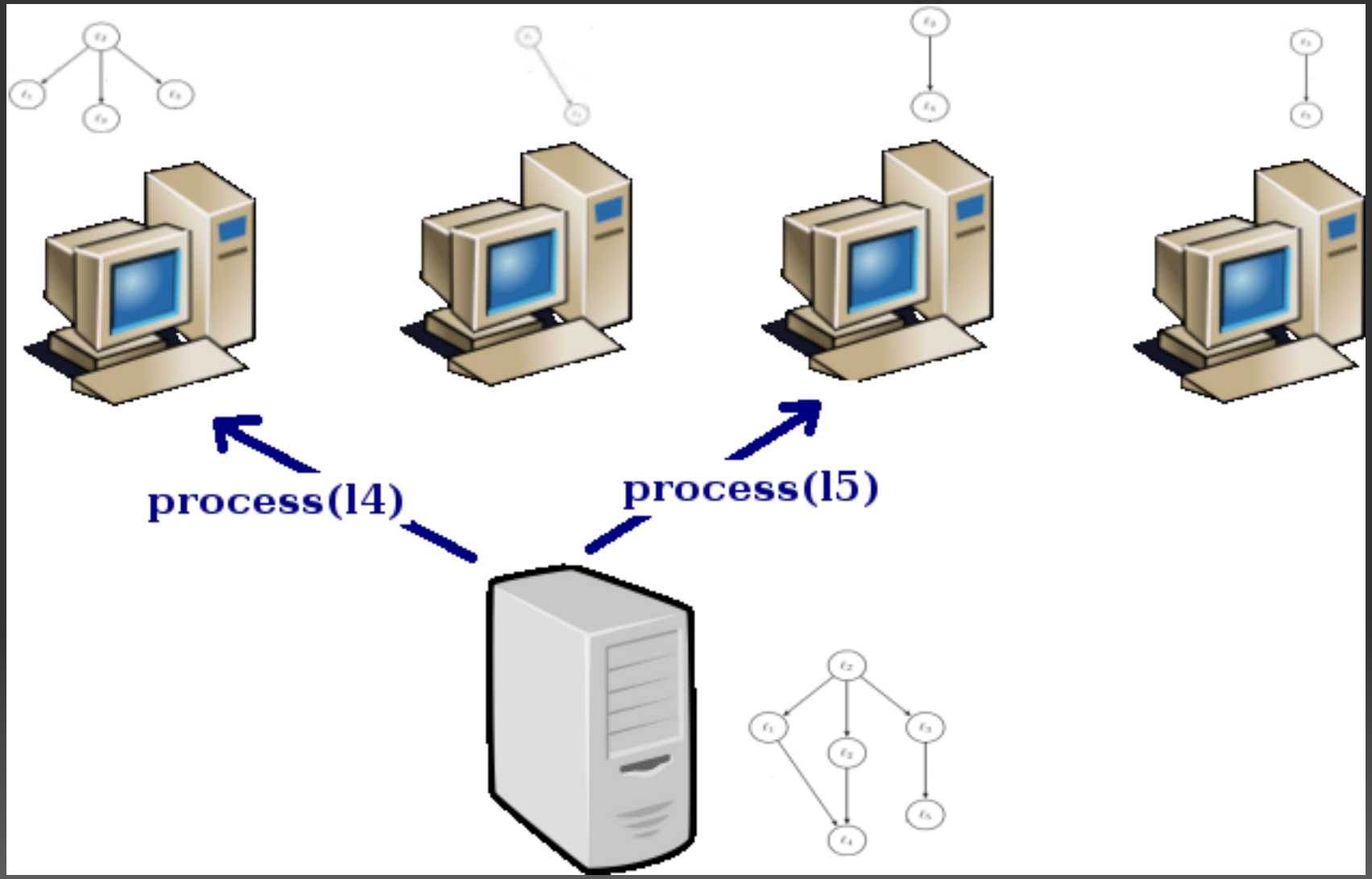
Example Distributed - #4



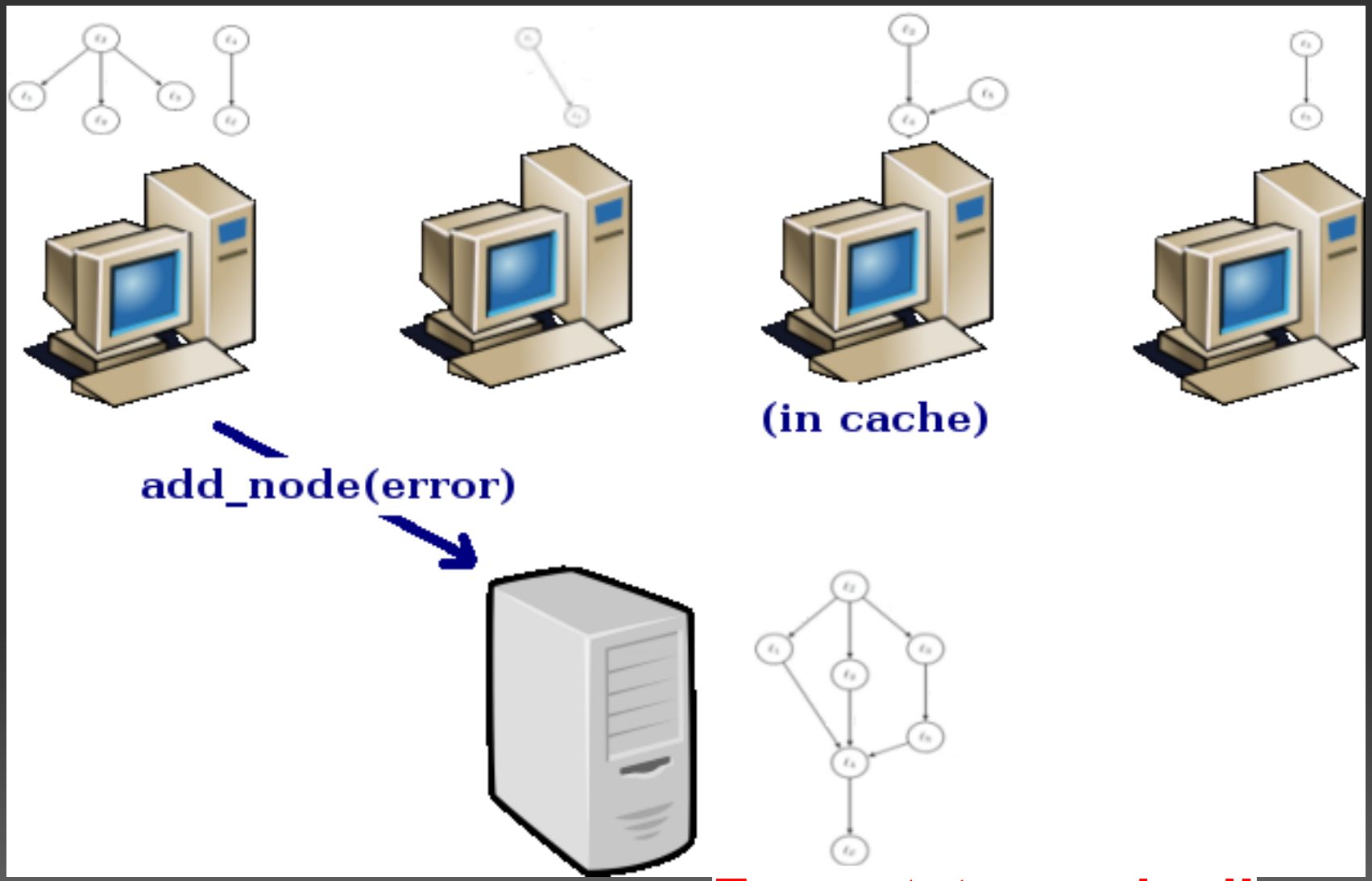
Example Distributed - #5



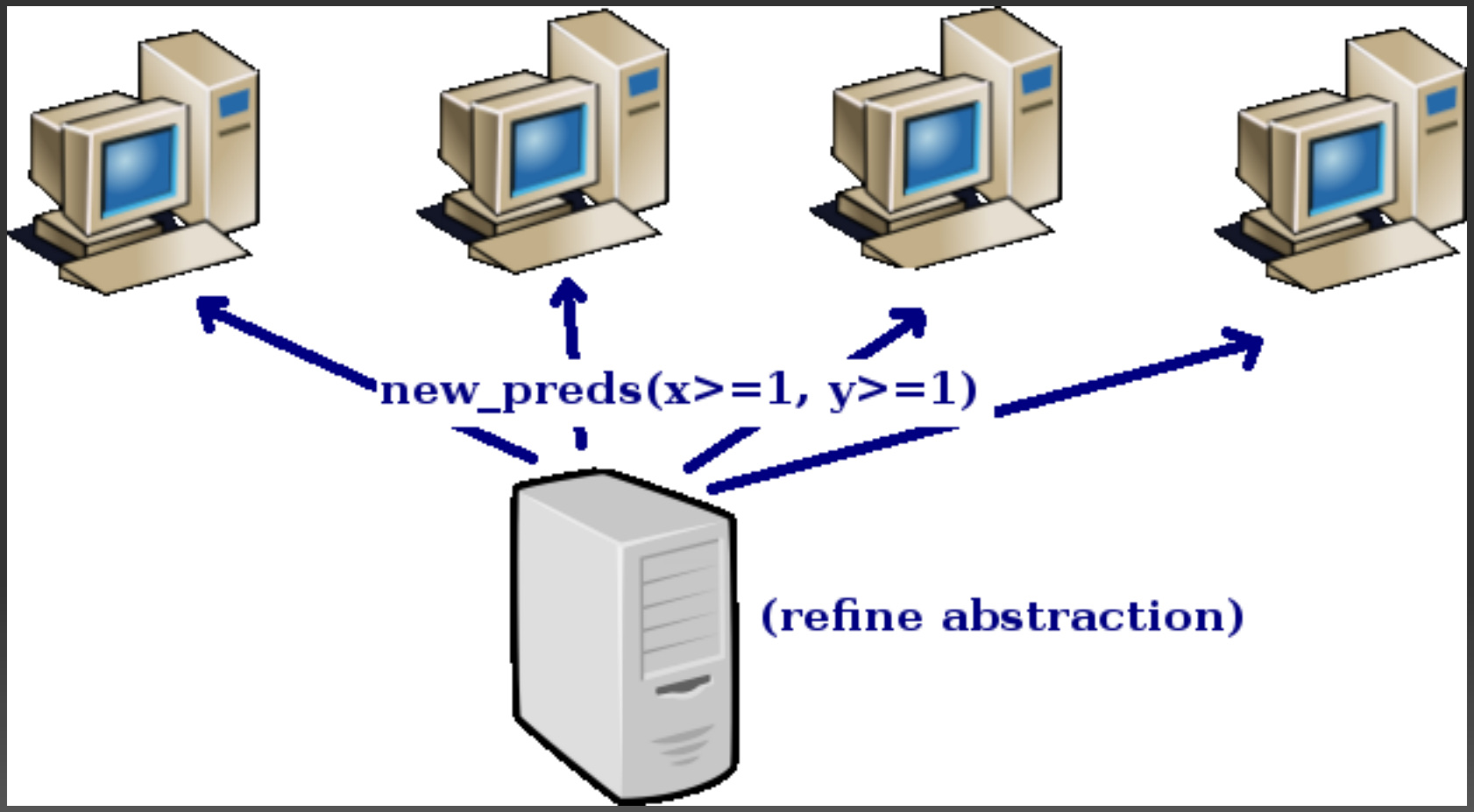
Example Distributed - #6



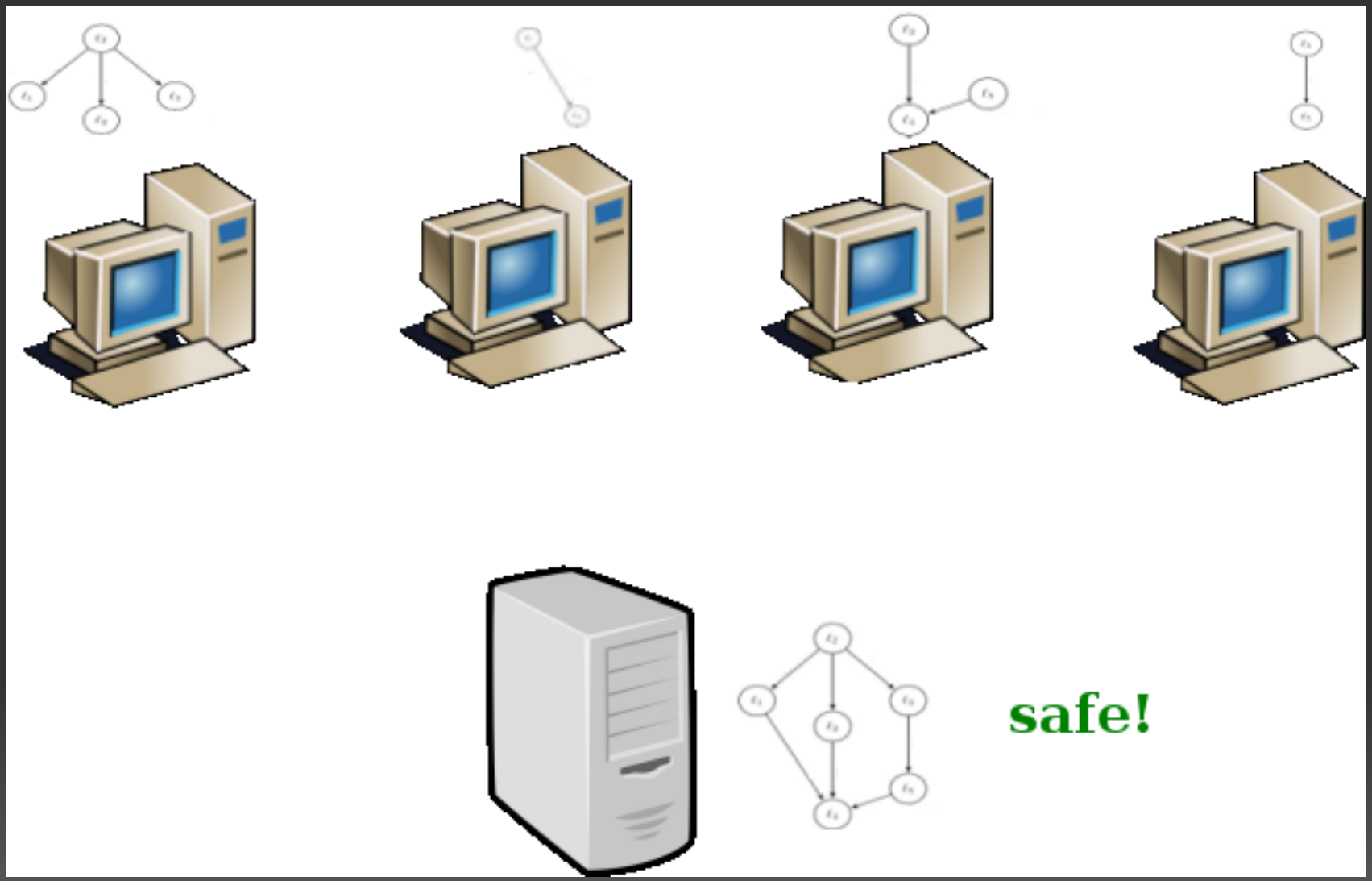
Example Distributed - #7



Example Distributed - #8



Example Distributed - #9



Algorithm

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state
2. if there exists such a path:

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state
2. if there exists such a path:
    1. return if some counterexample is feasible

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state
2. if there exists such a path:
    1. return if some counterexample is feasible
    2. otherwise refine one of those (chosen deterministically)

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state
2. if there exists such a path:
    1. return if some counterexample is feasible
    2. otherwise refine one of those (chosen deterministically)
3. broadcast the new set of predicates to all slaves

Algorithm

Distributed CEGAR:

1. Ask slaves to find all shortest paths leading to an error state
2. if there exists such a path:
    1. return if some counterexample is feasible
    2. otherwise refine one of those (chosen deterministically)
3. broadcast the new set of predicates to all slaves
4. goto 1.

Algorithm: Summary

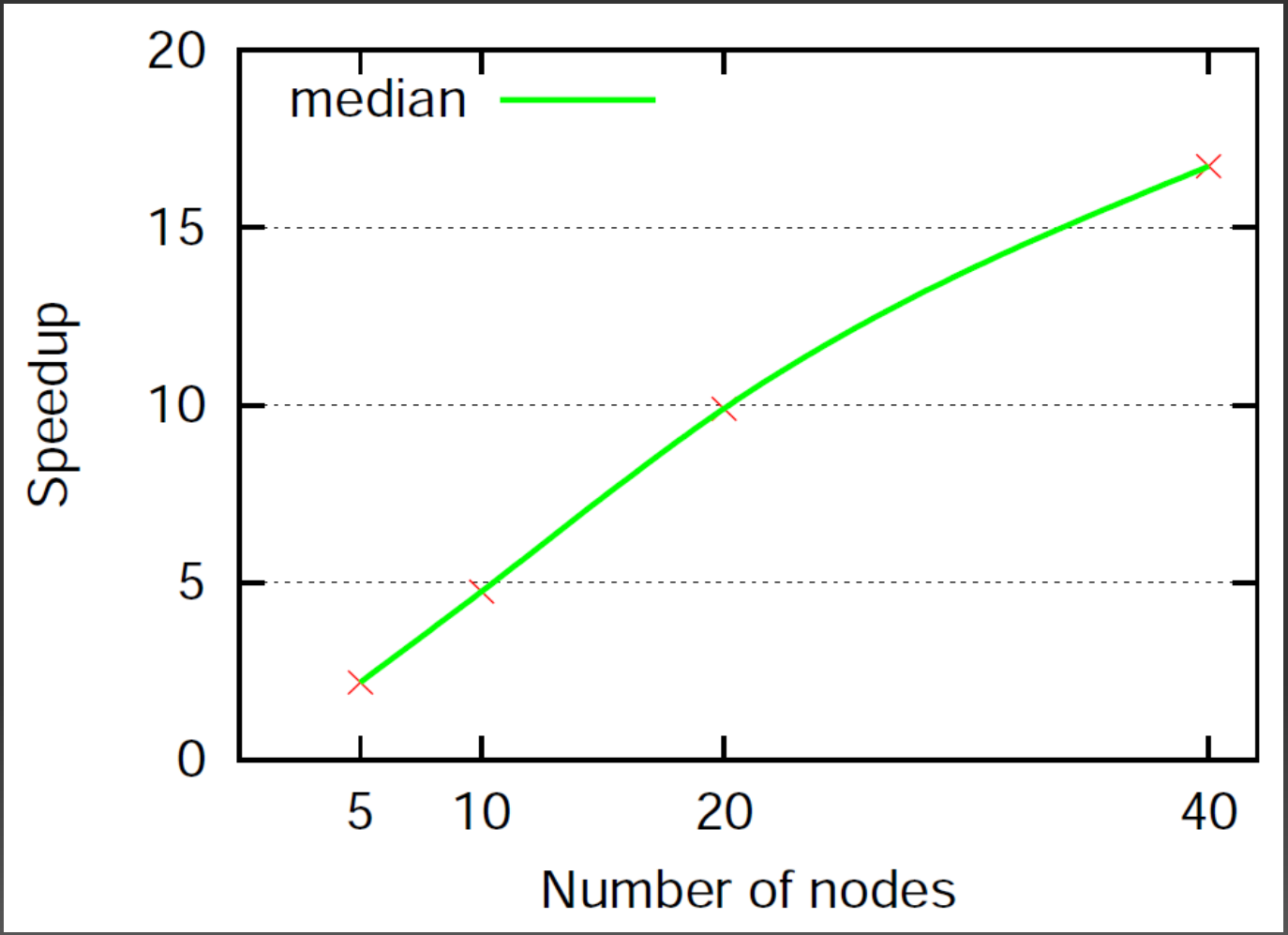
- Runs a BFS-style search over the graph
- Computes the full tree until a certain depth
- Always refines a shortest counterexample
- Speculative execution; some work may be discarded

Evaluation

Evaluation

- Extension of ARMC
- Benchmarks from the transportation domain (AVACS)
- Sequential execution ranging from hours to days

Evaluation



Conclusions

- Presented first distributed software model checking algorithm using message passing
- Linear scalability