Weak Memory Models as LLVM-to-LLVM Transformations

Vladimír Štill Petr Ročkai Jiří Barnat

Masaryk University

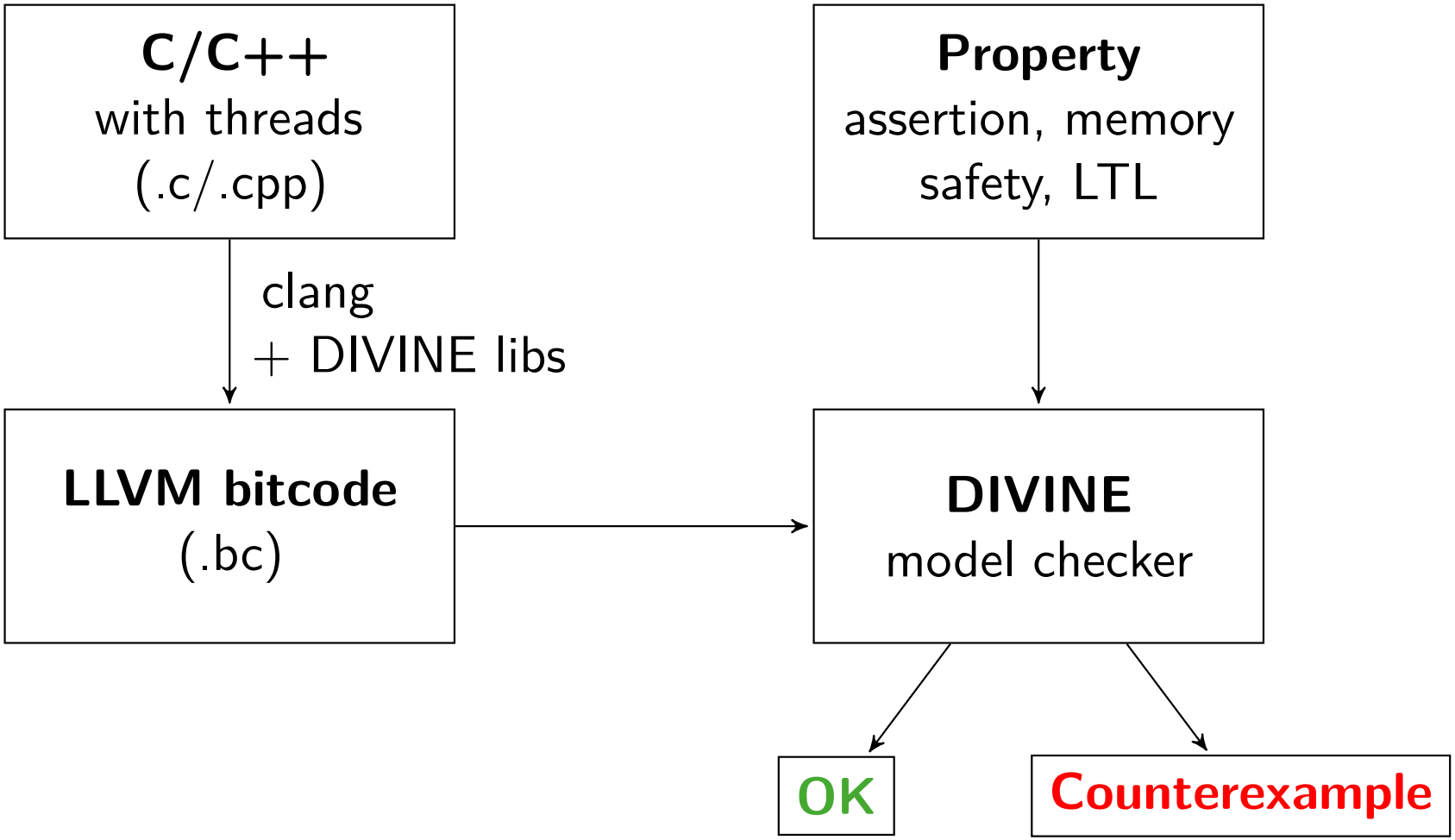
Brno, Czech Republic

24th October 2015

DIVINE model checker

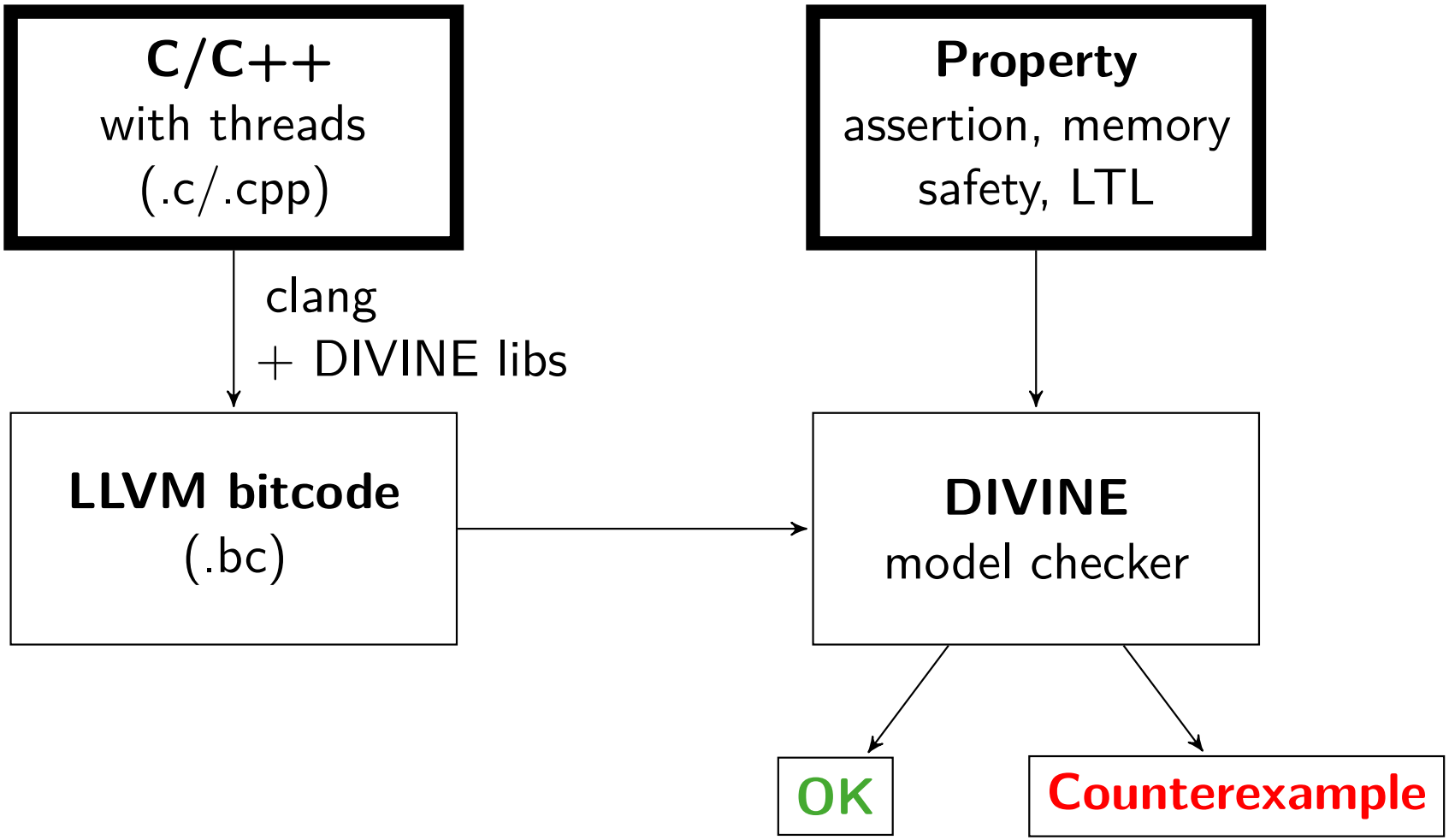
- automated detection of bugs in parallel C++ programs
- verification of assertion/memory safety, LTL specification
- using LLVM bitcode
  - kind of an assembler produced by compilers, such as `clang`
- supports large portions of C++ standard library
- verification of largely unmodified code

Introduction



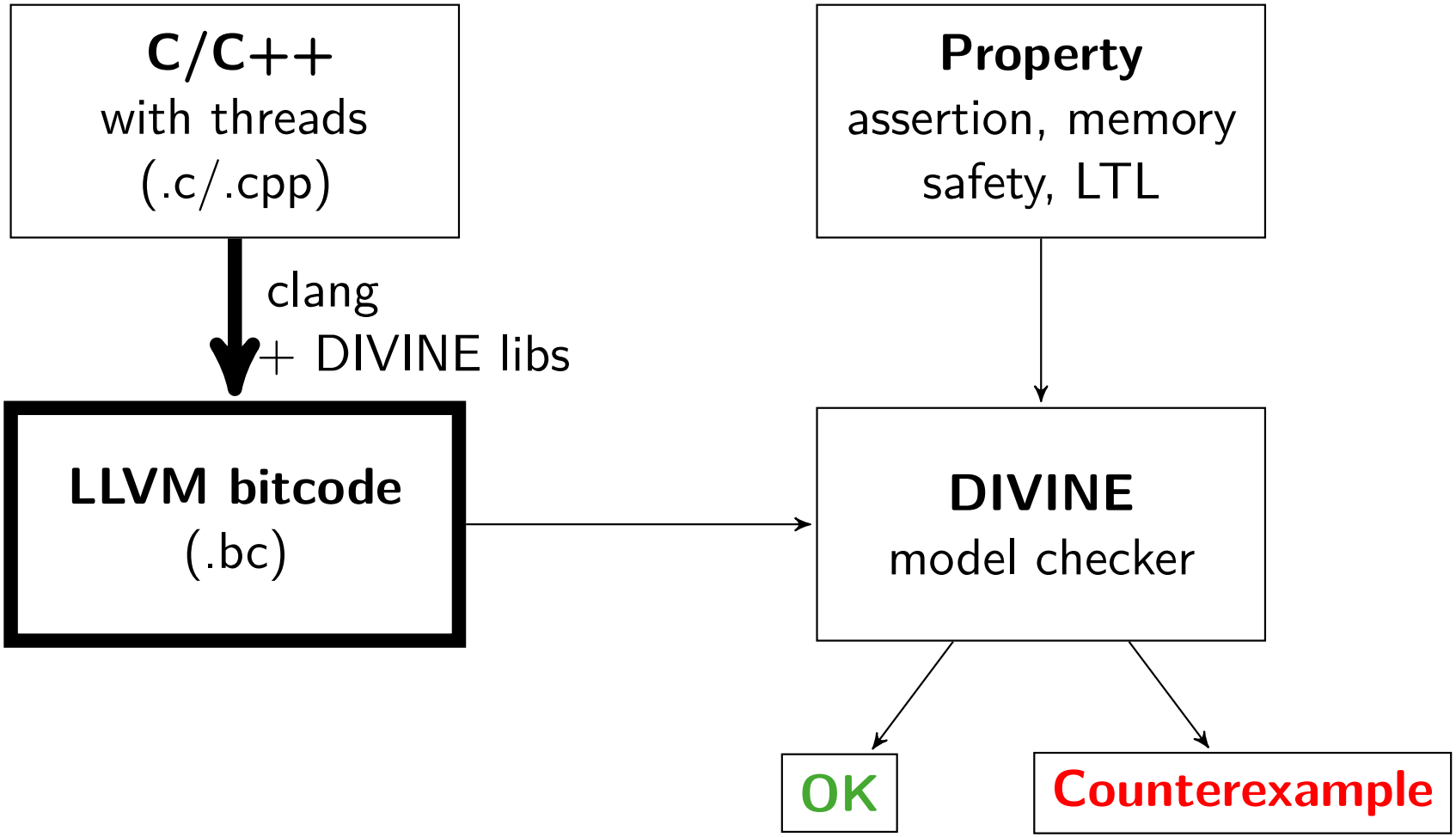
model checking programs with DIVINE

Introduction



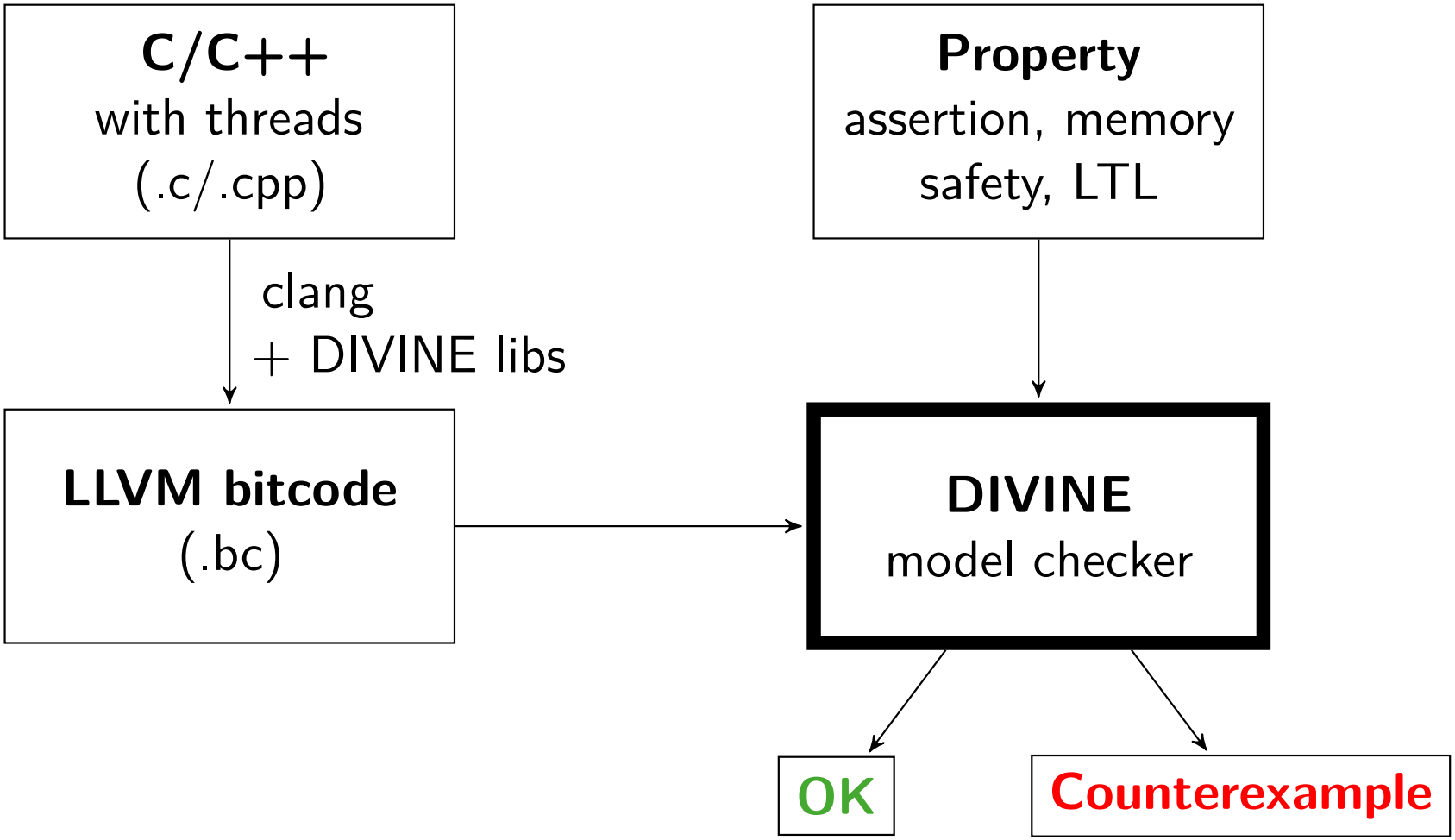
programmer gives inputs: source code and specification

Introduction



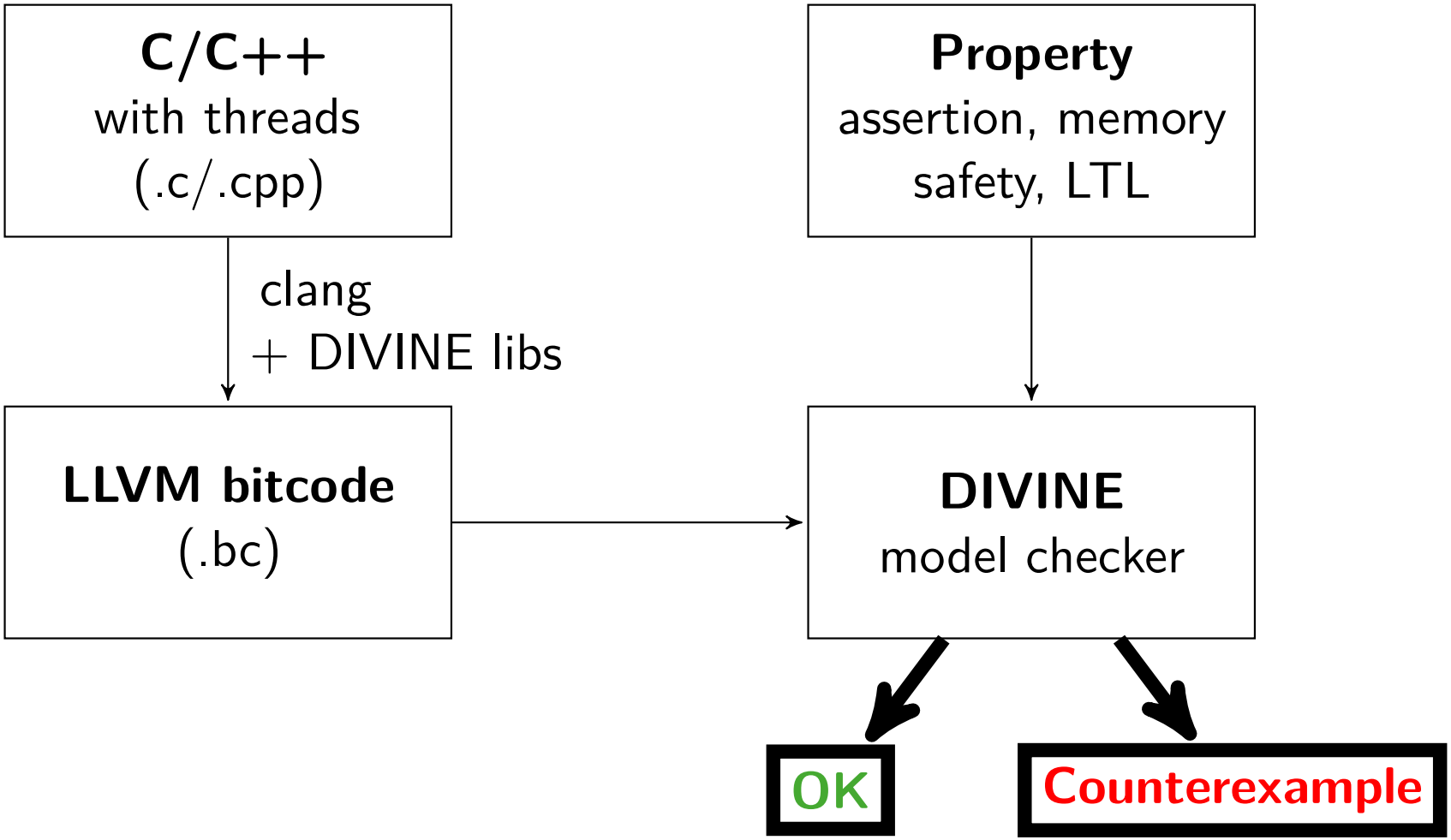
the program is compiled into LLVM bitcode

Introduction



DIVINE explores all relevant interleavings

Introduction



verification results

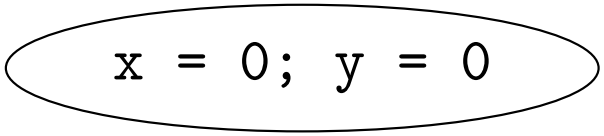
Explicit-State Model Checking

explores all relevant outcomes of a program

Explicit-State Model Checking

explores all relevant outcomes of a program

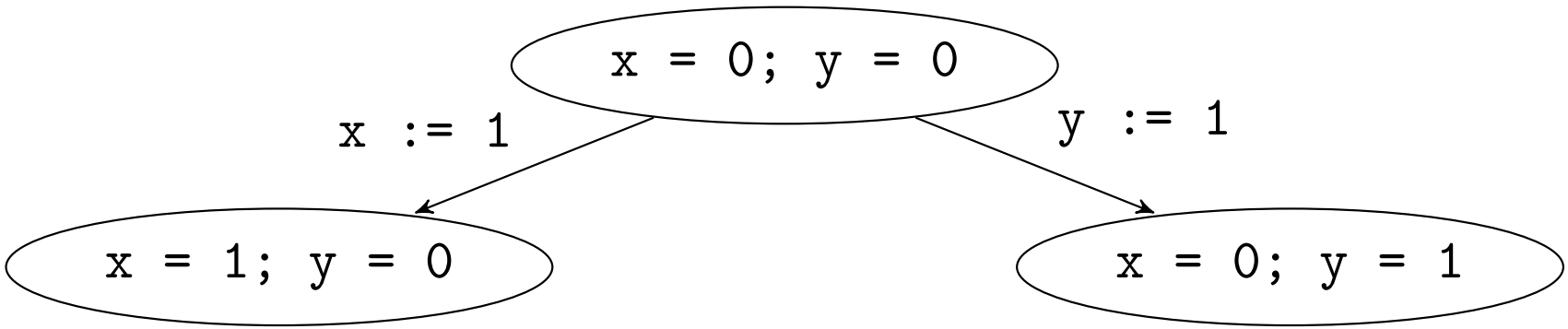
- starts from an initial state



Explicit-State Model Checking

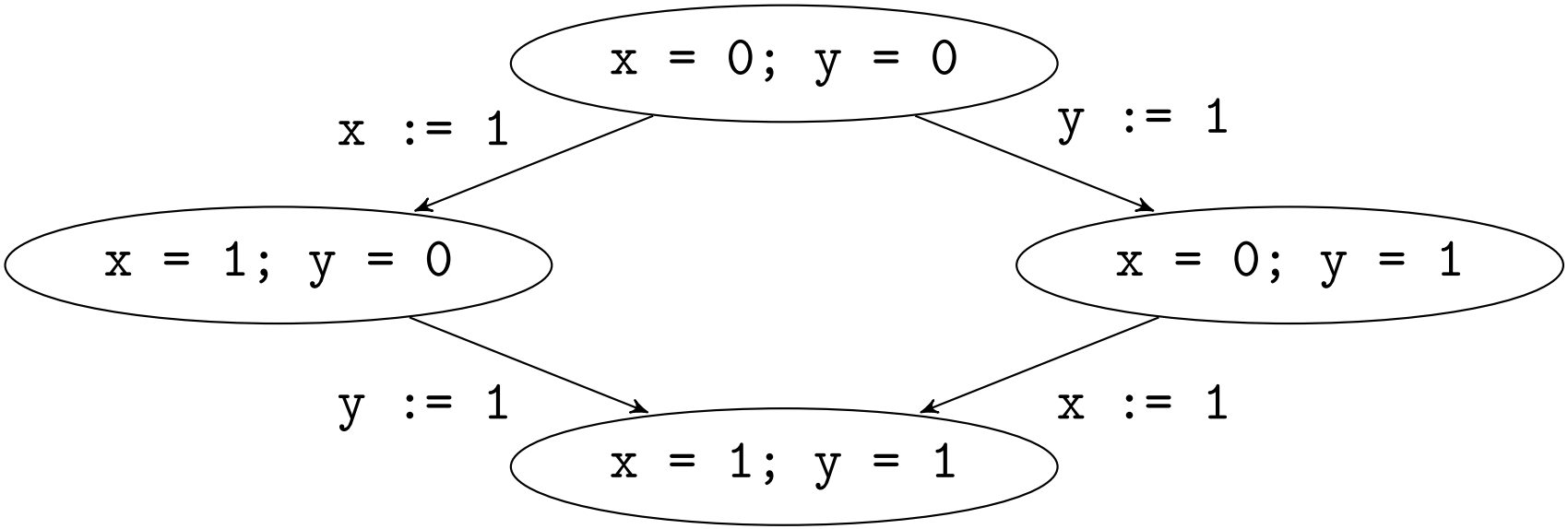
explores all relevant outcomes of a program

- starts from an initial state
- looks at possible actions that can be taken in each state



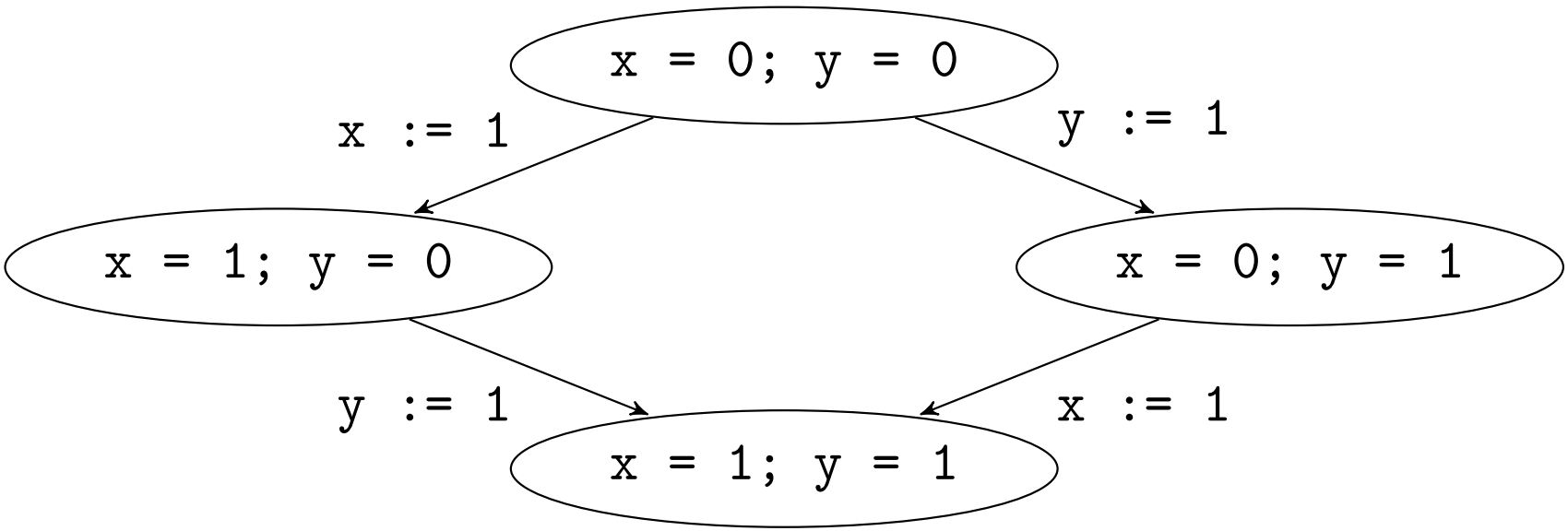
explores all relevant outcomes of a program

- starts from an initial state
- looks at possible actions that can be taken in each state



explores all relevant outcomes of a program

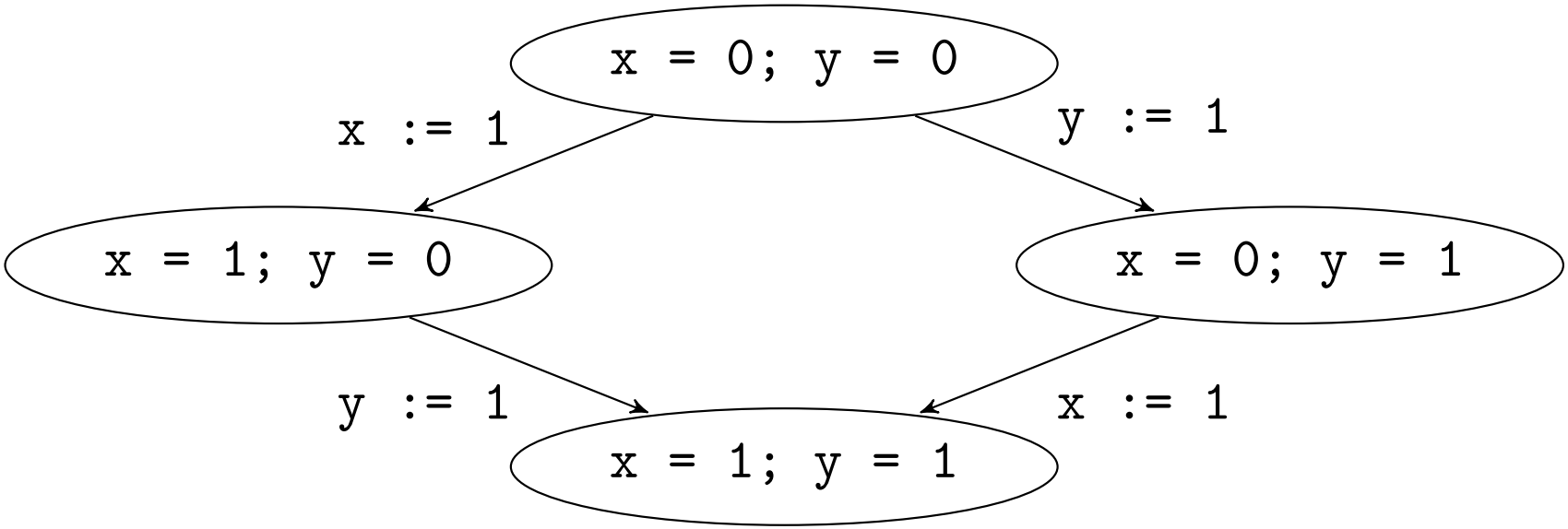
- starts from an initial state
- looks at possible actions that can be taken in each state



- builds state space

explores all relevant outcomes of a program

- starts from an initial state
- looks at possible actions that can be taken in each state



- builds state space
- graph exploration

Demo

Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - optimizations on the CPU, cache hierarchy

Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - optimizations on the CPU, cache hierarchy

- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce

Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - optimizations on the CPU, cache hierarchy
- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce
- newer revisions of C/C++ have support for specifying atomic variables with memory access ordering
  - same for Java, LLVM bitcode,...

Memory Models

- the order of reads and writes in the code does not need to match the order of their execution
  - compiler optimizations
  - optimizations on the CPU, cache hierarchy
- it is hard to reason about memory models
- parallelism is hard even under **Sequential Consistency**
  - reads and writes are immediate and cannot be reordered
  - not realistic, expensive to enforce
- newer revisions of C/C++ have support for specifying atomic variables with memory access ordering
  - same for Java, LLVM bitcode,...
- verifiers often assume sequential consistency
  - so does DIVINE

C++11 semantics of concurrent access to shared memory

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

C++11 semantics of concurrent access to shared memory

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- no guarantees
- `thread2` can read `y` only once
- undefined behaviour

C++11 semantics of concurrent access to shared memory

```
volatile int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- no atomicity guarantees
- variables must be read from memory on every access
- undefined behaviour

C++11 semantics of concurrent access to shared memory

```
std::atomic< int > x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- atomicity guaranteed
- variables must be atomically read from memory on every access

C++11 semantics of concurrent access to shared memory

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- can the output be `y = 2` and `x = 0`?

C++11 semantics of concurrent access to shared memory

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- can the output be `y = 2` and `x = 0`?
- yes, both the CPU and the compiler can reorder instructions

C++11 semantics of concurrent access to shared memory

```
volatile int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- can the output be `y = 2` and `x = 0`?
- yes, the CPU can reorder instructions

C++11 semantics of concurrent access to shared memory

```
std::atomic< int > x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

- can the output be `y = 2` and `x = 0`?
- no, the order of atomic loads and stores is guaranteed to match the order in the source code

Weak Memory Models

many different models on different architectures

- details are often not public
- details can vary between CPUs of the same architecture

Weak Memory Models

many different models on different architectures

- details are often not public
- details can vary between CPUs of the same architecture

theoretical memory models

- Total Store Order (TSO)
  - similar to the memory model used by x86_64
  - the order of execution of stores is guaranteed to match their order in machine code
    - compiler might still reorder stores
  - independent loads can be reordered

Weak Memory Models

many different models on different architectures

- details are often not public
- details can vary between CPUs of the same architecture

theoretical memory models

- Total Store Order (TSO)
  - similar to the memory model used by x86_64
  - the order of execution of stores is guaranteed to match their order in machine code
    - compiler might still reorder stores
  - independent loads can be reordered
- Partial Store Order (PSO)
  - weaker than TSO
  - independent stores can be reordered too
  - enables more optimizations

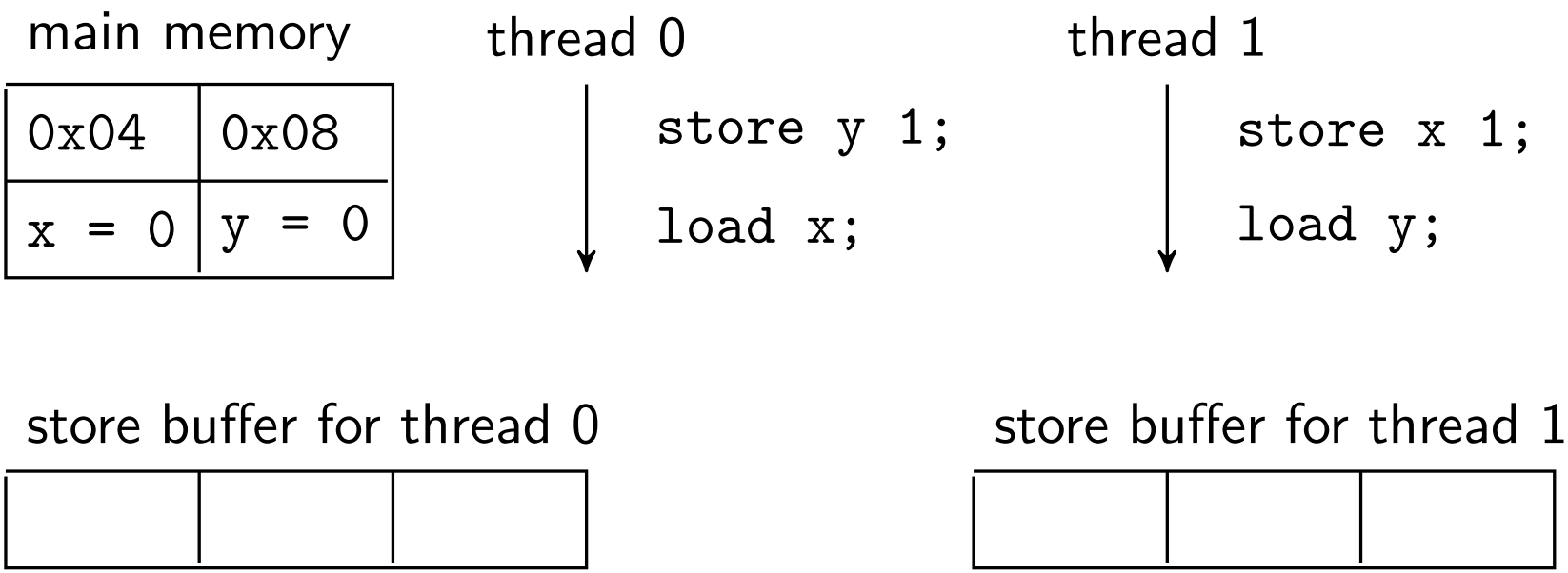
Weak Memory Models

```
int x = 0, y = 0;
```

```
1  void thread0() {
2      y = 1;
3      cout << "x = " << x;
4    }
```

```
1  void thread1() {
2      x = 1;
3      cout << "y = " << y;
4  }
```

Total Store Order can be simulated using store buffers:



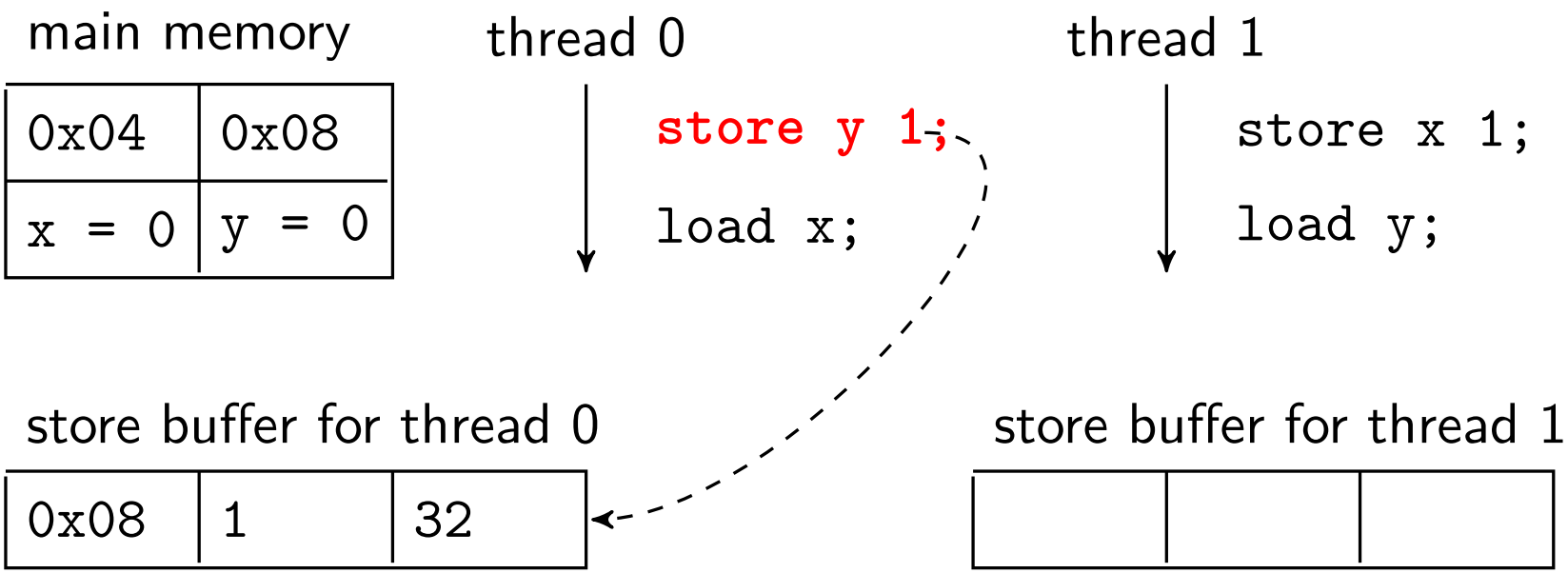
Weak Memory Models

```
int x = 0, y = 0;
```

```
1  void thread0() {
2      y = 1;
3      cout << "x = " << x;
4    }
```

```
1  void thread1() {
2      x = 1;
3      cout << "y = " << y;
4  }
```

Total Store Order can be simulated using store buffers:



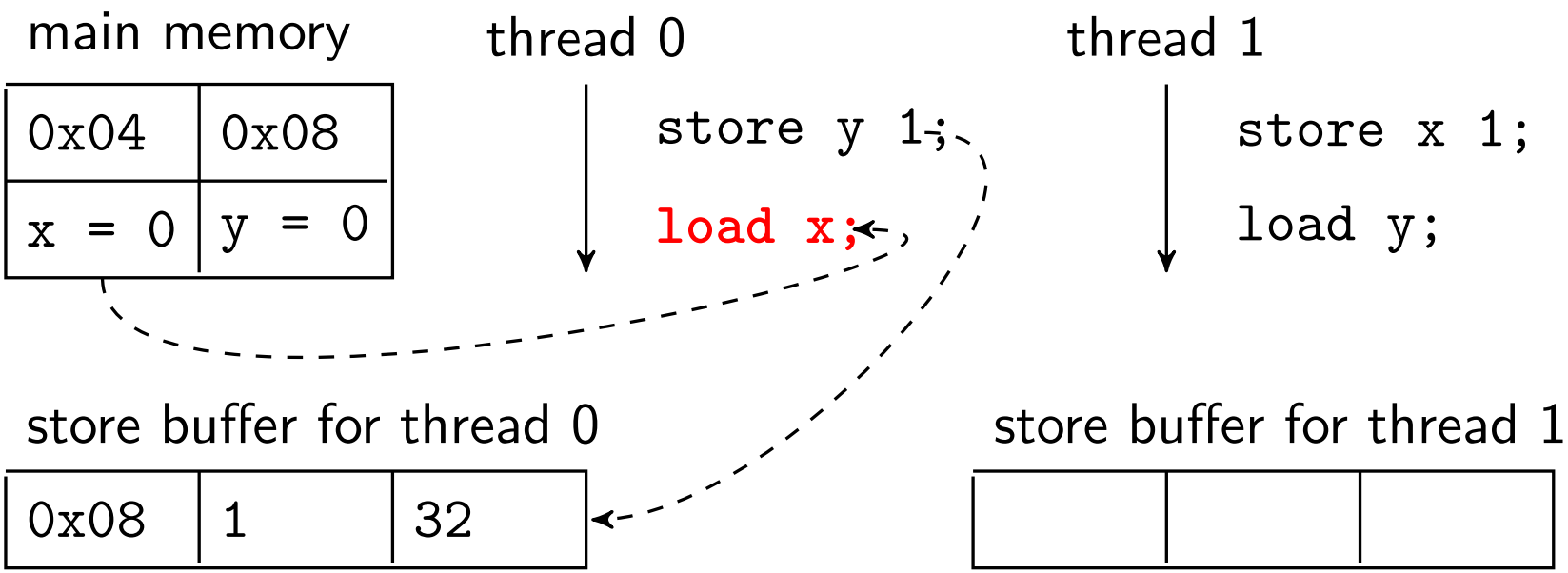
Weak Memory Models

```
int x = 0, y = 0;
```

```
1  void thread0() {
2      y = 1;
3      cout << "x = " << x;
4    }
```

```
1  void thread1() {
2      x = 1;
3      cout << "y = " << y;
4  }
```

Total Store Order can be simulated using store buffers:



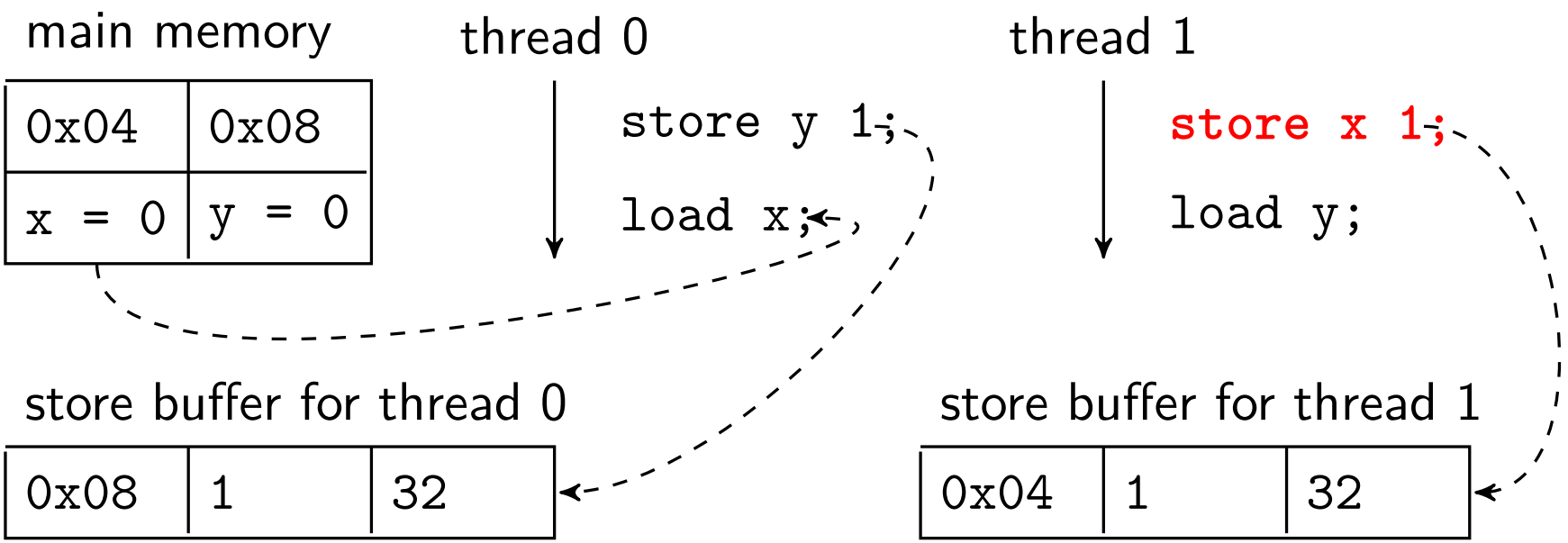
Weak Memory Models

```
int x = 0, y = 0;
```

```
1  void thread0() {
2      y = 1;
3      cout << "x = " << x;
4    }
```

```
1  void thread1() {
2      x = 1;
3      cout << "y = " << y;
4  }
```

Total Store Order can be simulated using store buffers:



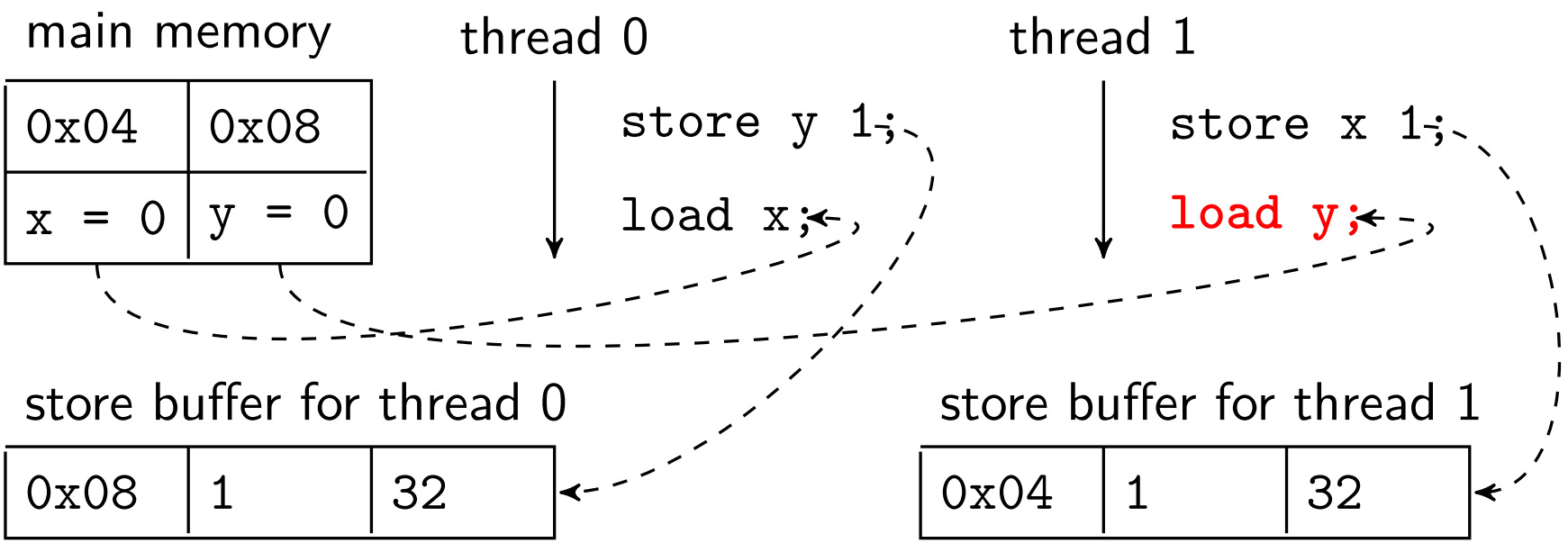
Weak Memory Models

```
int x = 0, y = 0;
```

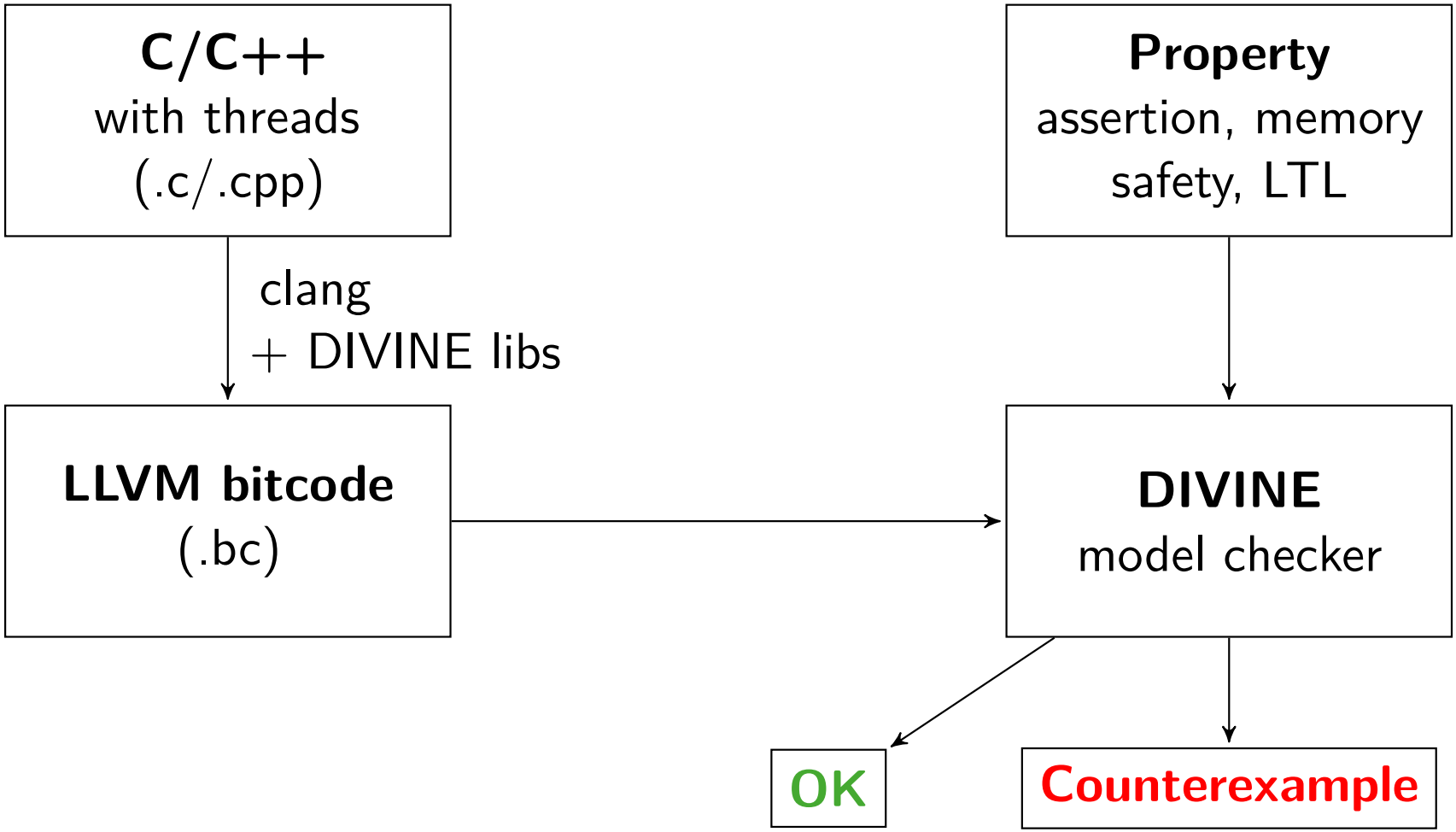
```
1  void thread0() {
2      y = 1;
3      cout << "x = " << x;
4    }
```

```
1  void thread1() {
2      x = 1;
3      cout << "y = " << y;
4  }
```

Total Store Order can be simulated using store buffers:

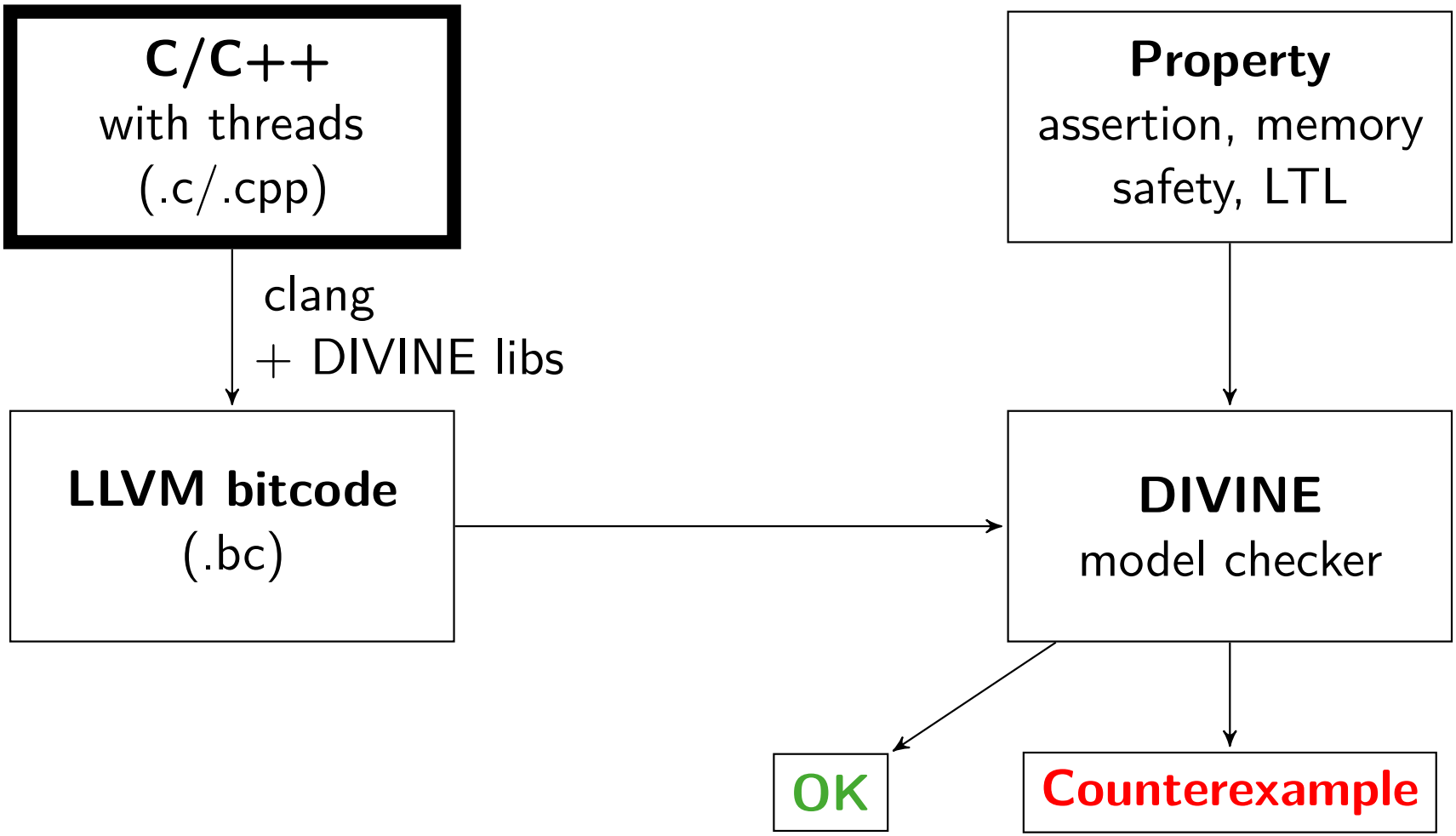


Weak Memory Models as Transformation



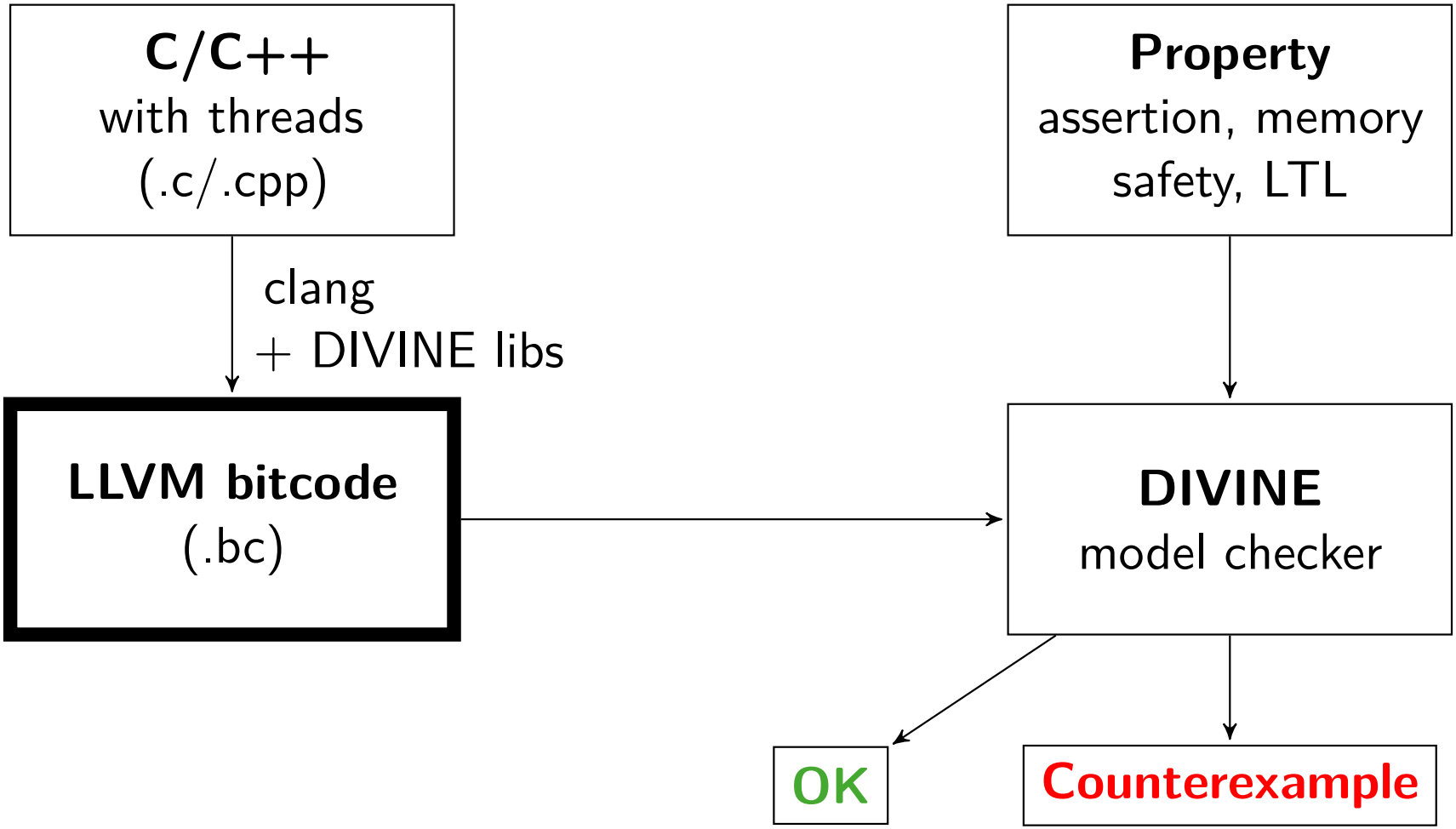
model checking programs with DIVINE

Weak Memory Models as Transformation



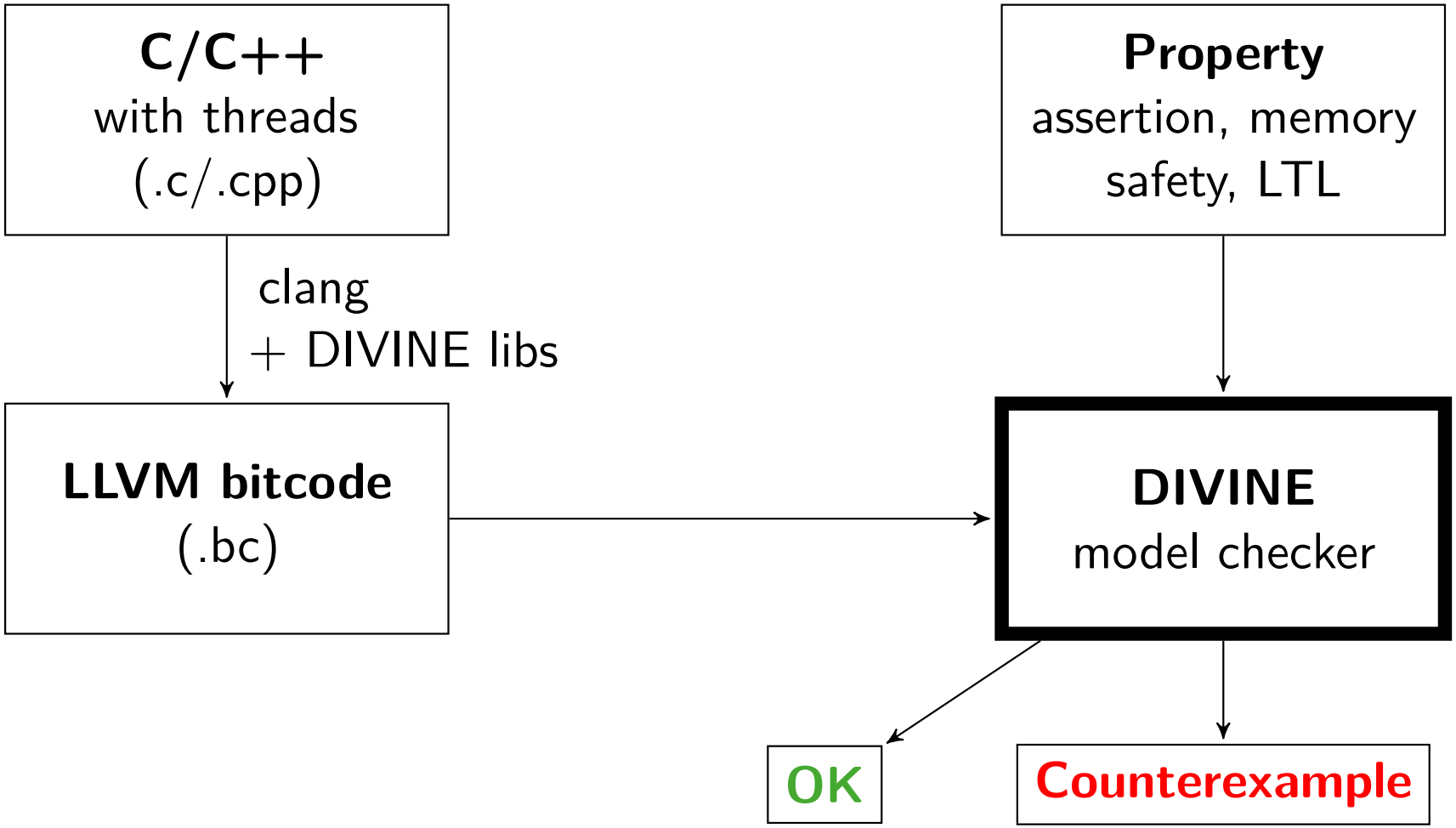
input program does not specify the memory model exactly

Weak Memory Models as Transformation



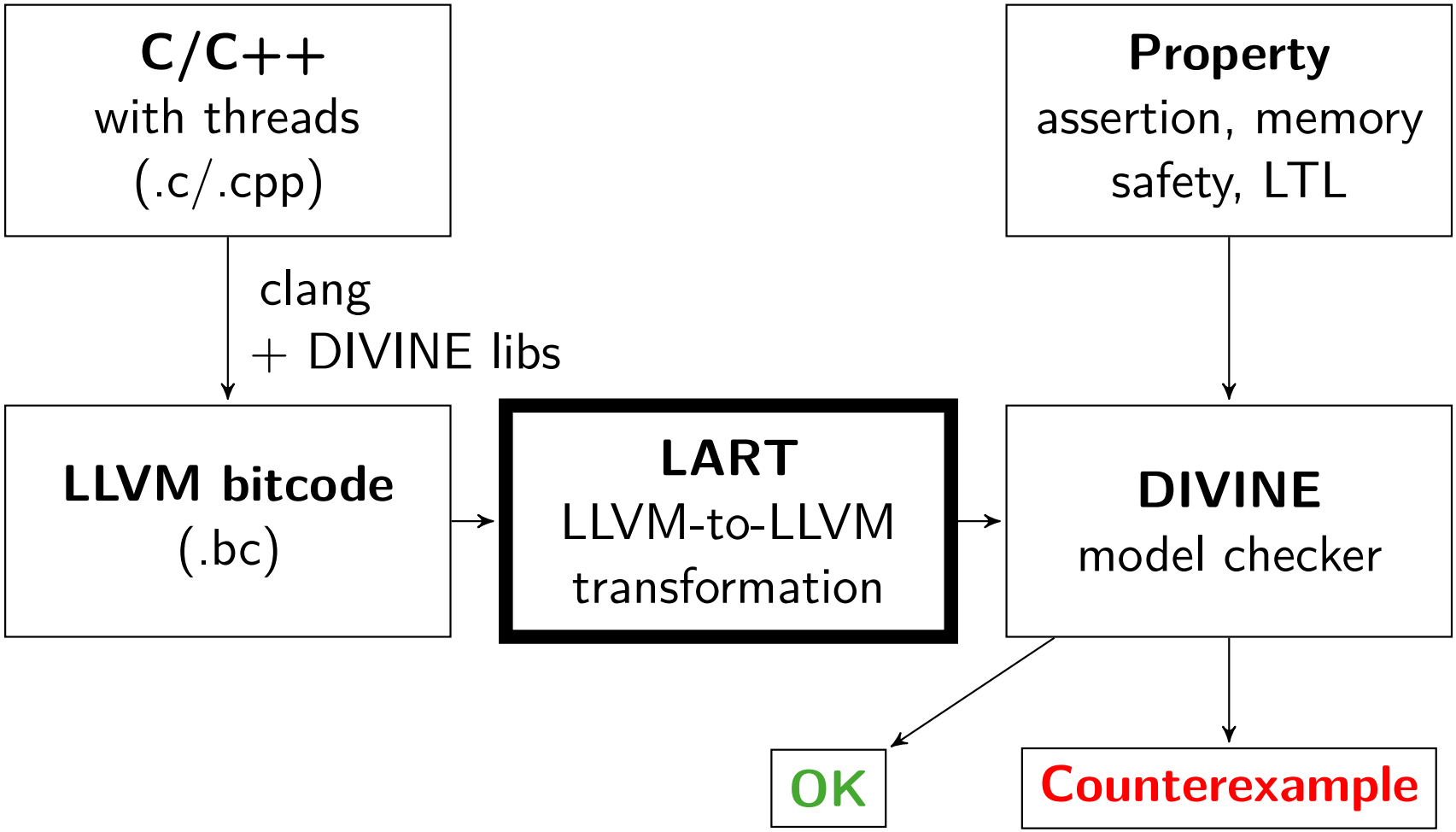
LLVM roughly copies the C++11 memory model

Weak Memory Models as Transformation



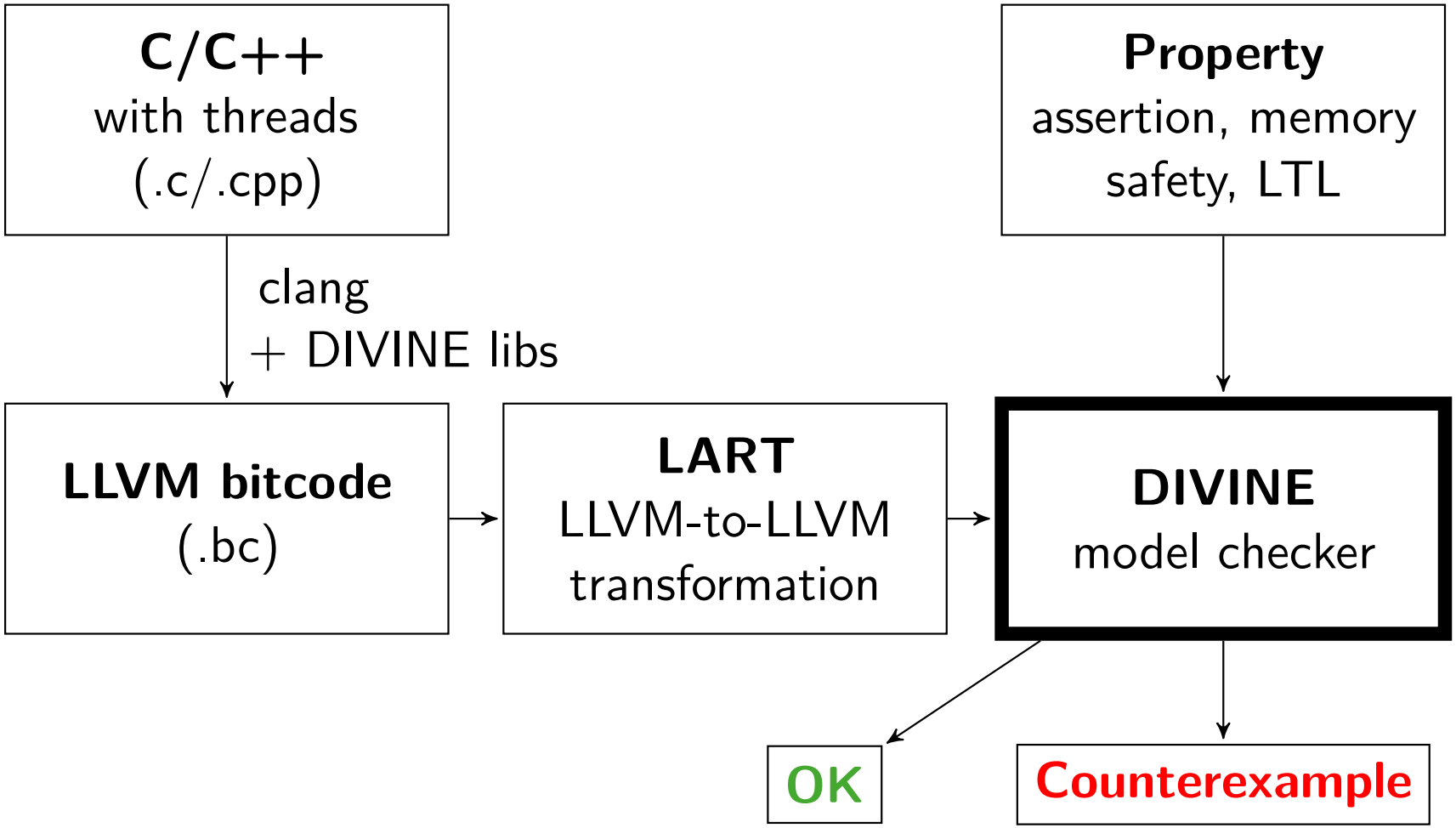
DIVINE assumes sequential consistency

Weak Memory Models as Transformation



LART instruments LLVM bitcode based on a relaxed memory model

Weak Memory Models as Transformation



the model is now verified assuming the relaxed memory model

TSO Under-Approximation

- TSO can be simulated using an unbounded store buffer
- this can easily make the state space infinite

TSO Under-Approximation

- TSO can be simulated using an unbounded store buffer
- this can easily make the state space infinite

- can be under-approximated using a bounded store buffer
  - if a bug is found, it can occur on TSO hardware
  - if bug is not found, there is no guarantee

TSO Under-Approximation

- TSO can be simulated using an unbounded store buffer
- this can easily make the state space infinite

- can be under-approximated using a bounded store buffer
  - if a bug is found, it can occur on TSO hardware
  - if bug is not found, there is no guarantee

  - integrates well with explicit-state model checker like DIVINE

Why LLVM-to-LLVM Transformation?

- memory model support could be integrated in the verifier
  - complicates the verifier
  - impractical if more memory models are to be supported

Why LLVM-to-LLVM Transformation?

- memory model support could be integrated in the verifier
  - complicates the verifier
  - impractical if more memory models are to be supported
- can be implemented in the program to be verified
  - manual implementation is tedious
  - automatic transformation is hard to do (especially for C++)

Why LLVM-to-LLVM Transformation?

- memory model support could be integrated in the verifier
  - complicates the verifier
  - impractical if more memory models are to be supported
- can be implemented in the program to be verified
  - manual implementation is tedious
  - automatic transformation is hard to do (especially for C++)
- transformation of LLVM
  - LLVM is an assembly-like language
  - abstracts away from machine registers, caches, ...
  - significantly simpler than high-level programming languages
  - API for transformations

Why LLVM-to-LLVM Transformation?

- memory model support could be integrated in the verifier
  - complicates the verifier
  - impractical if more memory models are to be supported
- can be implemented in the program to be verified
  - manual implementation is tedious
  - automatic transformation is hard to do (especially for C++)
- transformation of LLVM
  - LLVM is an assembly-like language
  - abstracts away from machine registers, caches, ...
  - significantly simpler than high-level programming languages
  - API for transformations
  - an ecosystem of code generators for many languages
  - analysis tools other than DIVINE can use this transformation

The Transformation

memory-manipulating instructions need to be replaced to enable TSO simulation

- add a store buffer for each thread
- `store` instructions save data to the store buffer
- `load` instructions first look up data in the local store buffer, then in memory
- `fence` (memory barrier) instructions flush the store buffer
- atomic instructions and memory-manipulating functions such as `memcpy` needs to be handled
- buffers are flushed nondeterministically

Infinite Delay Problem

$\varphi = FG(\neg w_0 \wedge \neg w_1)$

```
bool x = false, y = false;
```

```
1  void thread0() {
2    y = true;
3    while (!x) { AP(w0); }
4    for (;;) { /*work*/ }
5  }
```

```
1  void thread1() {
2    x = true;
3    while (!y) { AP(w1); }
4    for (;;) { /*work*/ }
5  }
```

- φ does not hold on a run where `flush` is delayed infinitely

Infinite Delay Problem

$\varphi = FG(\neg w_0 \wedge \neg w_1)$

```
bool x = false, y = false;
```

```
1  void thread0() {
2    y = true;
3    while (!x) { AP(w0); }
4    for (;;) { /*work*/ }
5  }
```

```
1  void thread1() {
2    x = true;
3    while (!y) { AP(w1); }
4    for (;;) { /*work*/ }
5  }
```

- φ does not hold on a run where `flush` is delayed infinitely

Solution

- for each program thread, we add a dedicated flushing thread
- we use weak fairness to avoid infinite delays

Writes To Invalidated Memory

store buffers associated with invalid memory locations can be flushed

- writes to freed dynamic memory
- writes to stack frames of finished functions

Writes To Invalidated Memory

store buffers associated with invalid memory locations can be flushed

- writes to freed dynamic memory
- writes to stack frames of finished functions

- for the first case, we can replace `free` so that it evicts freed memory addresses from store buffers

Writes To Invalidated Memory

store buffers associated with invalid memory locations can be flushed

- writes to freed dynamic memory
- writes to stack frames of finished functions

- for the first case, we can replace `free` so that it evicts freed memory addresses from store buffers

- for stack memory, cleanup needs to be added at the end of every function which uses stack memory
  - exception handlers need to be added

Writes To Invalidated Memory

store buffers associated with invalid memory locations can be flushed

- writes to freed dynamic memory
- writes to stack frames of finished functions

- for the first case, we can replace `free` so that it evicts freed memory addresses from store buffers

- for stack memory, cleanup needs to be added at the end of every function which uses stack memory
  - exception handlers need to be added

- currently not implemented
  - DIVINE cannot verify memory safety with TSO currently

Demo

State Space Reductions

- DIVINE employs $\tau+$ and heap symmetry reductions
  - $\tau+$ hides actions which are not observable, such as loads and stores to thread-local memory
  - this enables verification of real-world code

State Space Reductions

- DIVINE employs $\tau+$ and heap symmetry reductions
  - $\tau+$ hides actions which are not observable, such as loads and stores to thread-local memory
  - this enables verification of real-world code
- with TSO simulation, every load and store is visible
  - this makes reduction inefficient

State Space Reductions

- DIVINE employs $\tau+$ and heap symmetry reductions
  - $\tau+$ hides actions which are not observable, such as loads and stores to thread-local memory
  - this enables verification of real-world code
- with TSO simulation, every load and store is visible
  - this makes reduction inefficient
- thread-local memory does not need to be stored in a store buffer

Experimental Results

model	buffer size	assertion found	# of states	reduced # of states	memory [GB]	time [s]
`simple_sc`	N/A	no	205	N/A	0.16	1
`simple_mtso`	1	**yes**	6.89 k	N/A	0.17	3
`simple_stso`	1	**yes**	10.7 k	10.7 k	0.17	6
`simple_tso`	1	**yes**	24.7 M	537.2 k	3.18	20318
`peterson_sc`	N/A	no	1.68 k	N/A	0.16	1
`peterson_tso`	0	no	55.9 k	N/A	0.17	38
`peterson_tso`	2	**yes**	2.86 M	95.7 k	0.79	990
`peterson_tso`	3	**yes**	4.70 M	129.9 k	1.21	1610
`fifo_sc`	0	no	6951	N/A	0.73	20
`fifo_tso`	1	no	–	44 M	–	–

Conclusion

- fully automatic instrumentation of LLVM bitcode with Total Store Order approximation
- enables TSO verification in DIVINE or other verifiers which assume Sequential Consistency
- verification of assertion safety and LTL properties
- we were able to verify some interesting properties with this transformation, despite the state space growth

Conclusion

- fully automatic instrumentation of LLVM bitcode with Total Store Order approximation
- enables TSO verification in DIVINE or other verifiers which assume Sequential Consistency
- verification of assertion safety and LTL properties
- we were able to verify some interesting properties with this transformation, despite the state space growth

Future Work

- enable memory safety verification with TSO
- implementation of other weak memory models, such as Partial Store Order
- better state space reductions

Conclusion

- fully automatic instrumentation of LLVM bitcode with Total Store Order approximation
- enables TSO verification in DIVINE or other verifiers which assume Sequential Consistency
- verification of assertion safety and LTL properties
- we were able to verify some interesting properties with this transformation, despite the state space growth

Future Work

- enable memory safety verification with TSO
- implementation of other weak memory models, such as Partial Store Order
- better state space reductions

Thank You!