DIVINE: Verification of Real-World Software

Vladimír Štill

Masaryk University

Brno, Czech Republic

April 20, 2017

The Problem Introduction

- we are concerned with helping developers create correct programs

The Problem Introduction

- we are concerned with helping developers create correct programs
- using formal methods

The Problem Introduction

- we are concerned with helping developers create correct programs
- using formal methods
- these methods have to be actually usable
  - minimal investment from the developers' perspective
  - gives useful answers most of the time

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...
  - well used by developers
  - cannot prove absence of errors
  - does not work well with nondeterministic programs

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...
  - well used by developers
  - cannot prove absence of errors
  - does not work well with nondeterministic programs
    - threading

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...
  - well used by developers
  - cannot prove absence of errors
  - does not work well with nondeterministic programs
    - threading
    - nondeterministic error injection

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...
  - well used by developers
  - cannot prove absence of errors
  - does not work well with nondeterministic programs
    - threading
    - nondeterministic error injection
  - requires good tests

Ways Towards Program Correctness

many already widely used methods

- testing
  - unit tests, integration tests, scenarios,...
  - well used by developers
  - cannot prove absence of errors
  - does not work well with nondeterministic programs
    - threading
    - nondeterministic error injection
  - requires good tests
- static analysis
  - good for finding common problems directly from the source (without execution)
  - compiler warnings, linters, ...

Ways Towards Program Correctness

formal methods

- symbolic execution
  - often used for test case generation to increase coverage
  - requires programs which terminate for all inputs to work well
  - mostly used for sequential programs with arbitrary input data
  - detects violations of given property, strength depends on the property
  - path explosion

Ways Towards Program Correctness

formal methods

- symbolic execution
  - often used for test case generation to increase coverage
  - requires programs which terminate for all inputs to work well
  - mostly used for sequential programs with arbitrary input data
  - detects violations of given property, strength depends on the property
  - path explosion
- explicit-state model checking
  - mostly used for parallel or otherwise nondeterministic programs without nondeterministic input data
  - can deal with non-terminating programs as long as they have finitely many states
  - detects violations of given property, strength depends on the property
  - state-space explosion

Ways Towards Program Correctness

formal methods

- symbolic execution
  - often used for test case generation to increase coverage
  - requires programs which terminate for all inputs to work well
  - mostly used for sequential programs with arbitrary input data
  - detects violations of given property, strength depends on the property
  - path explosion
- explicit-state model checking
  - mostly used for parallel or otherwise nondeterministic programs without nondeterministic input data
  - can deal with non-terminating programs as long as they have finitely many states
  - detects violations of given property, strength depends on the property
  - state-space explosion
- theorem proving
  - requires substantial human input
  - the user must be skilled both in programming and in mathematical reasoning

Ways Towards Program Correctness

formal methods

- symbolic execution
  - often used for test case generation to increase coverage
  - requires programs which terminate for all inputs to work well
  - mostly used for sequential programs with arbitrary input data
  - detects violations of given property, strength depends on the property
  - path explosion
- explicit-state model checking
  - mostly used for parallel or otherwise nondeterministic programs without nondeterministic input data
  - can deal with non-terminating programs as long as they have finitely many states
  - detects violations of given property, strength depends on the property
  - state-space explosion
- theorem proving
  - requires substantial human input
  - the user must be skilled both in programming and in mathematical reasoning

Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program

Introducing Explicit-State Model Checking

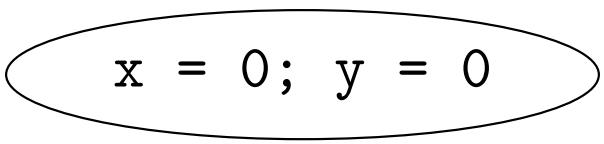
- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds

Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*

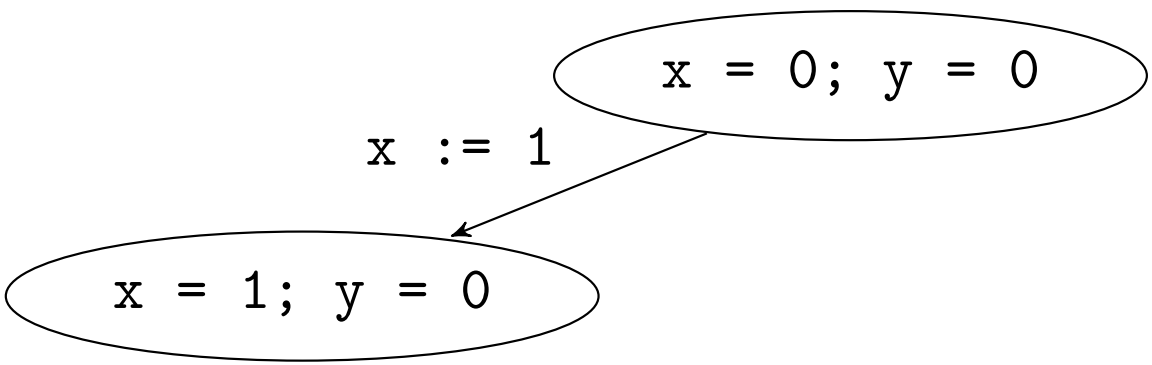
Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*



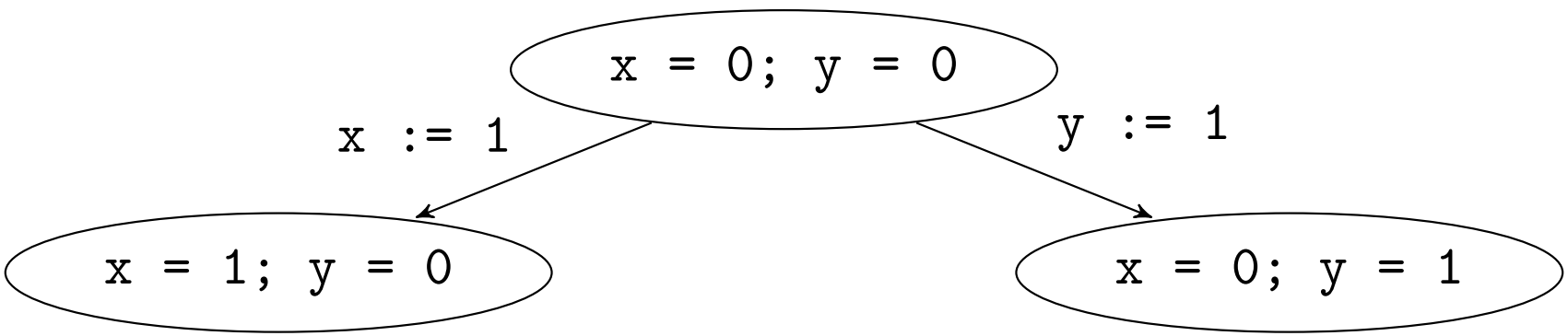
Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*



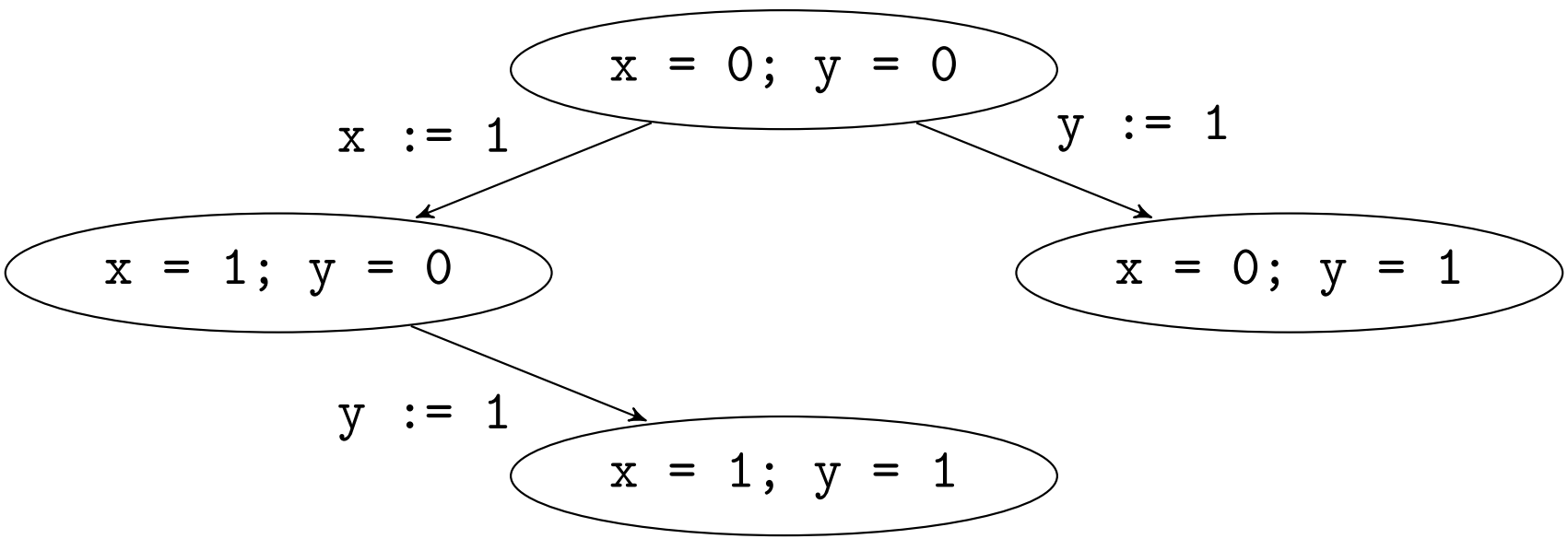
Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*



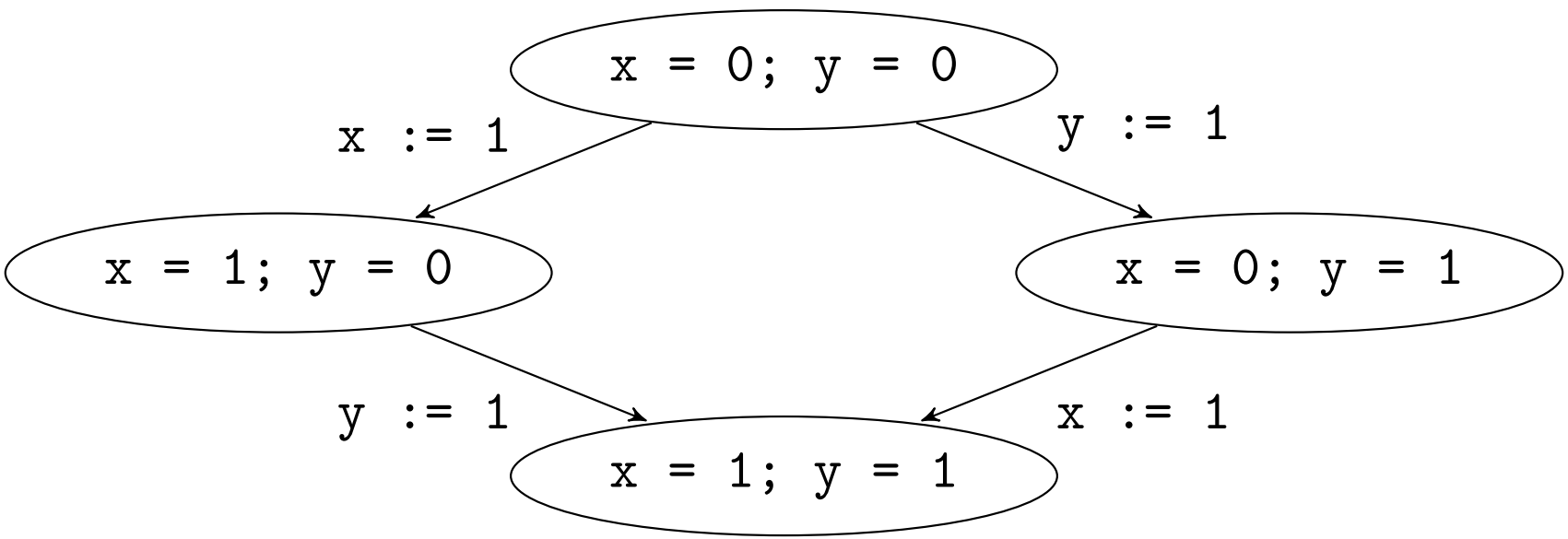
Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*



Introducing Explicit-State Model Checking

- verifies given property by systematically exploring all possible runs of the program
  - or a sufficient subset of runs needed to prove the property holds
- does this by exploring all possible states in which the program can be and by creating a *state space graph*



Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++

Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++
- aims to be useful for developers
  - testing nondeterministic programs
  - strict memory checking
  - support for complex properties, including temporal properties of infinite runs

Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++
- aims to be useful for developers
  - testing nondeterministic programs
  - strict memory checking
  - support for complex properties, including temporal properties of infinite runs
- has to solve many problems to be usable
  - comprehension of the input language and its common libraries

Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++
- aims to be useful for developers
  - testing nondeterministic programs
  - strict memory checking
  - support for complex properties, including temporal properties of infinite runs
- has to solve many problems to be usable
  - comprehension of the input language and its common libraries
  - efficient storage of program state

Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++
- aims to be useful for developers
  - testing nondeterministic programs
  - strict memory checking
  - support for complex properties, including temporal properties of infinite runs
- has to solve many problems to be usable
  - comprehension of the input language and its common libraries
  - efficient storage of program state
  - efficient exploration of important parts of the state space

Introducing DIVINE

- an explicit-state model checker
- a platform for verification of software written in C and C++
- aims to be useful for developers
  - testing nondeterministic programs
  - strict memory checking
  - support for complex properties, including temporal properties of infinite runs
- has to solve many problems to be usable
  - comprehension of the input language and its common libraries
  - efficient storage of program state
  - efficient exploration of important parts of the state space
  - general usability without expertise in program analysis

Language Support in DIVINE

- DIVINE executes LLVM intermediate language (LLVM IR)
- C and C++ can be translated to LLVM IR using clang compiler

Language Support in DIVINE

- DIVINE executes LLVM intermediate language (LLVM IR)
- C and C++ can be translated to LLVM IR using clang compiler
- all parts of the program, including the libraries it uses, need to be translated to LLVM IR
  - DIVINE handles the translation of user's program and links it to pre-translated libraries
  - C and C++ standard libraries + `pthreads` are supported

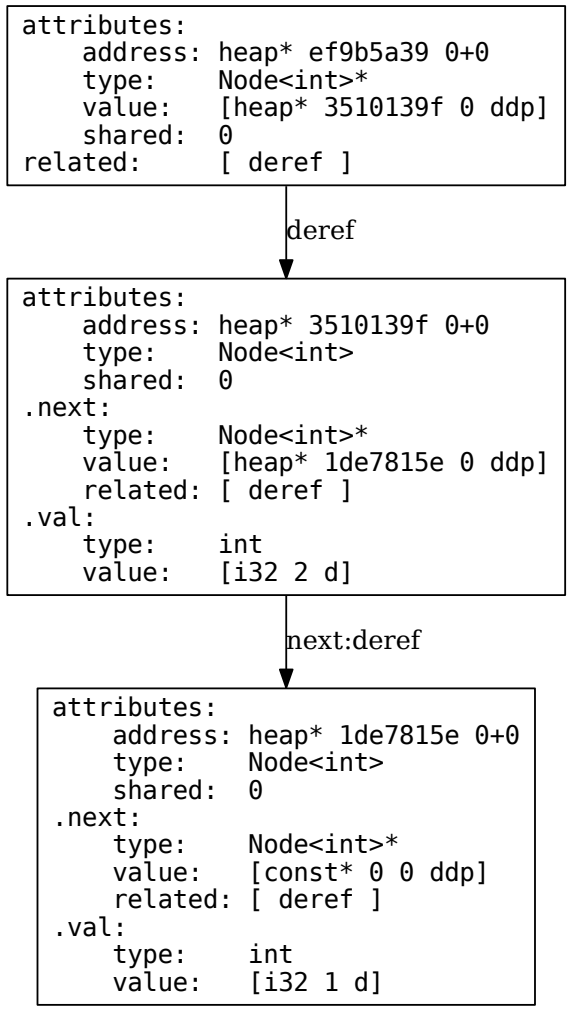
State Space Storage

- DIVINE needs to store the state space
- states are essential snapshots of the memory of the program
  - memory is organised as a graph

State Space Storage

- DIVINE needs to store the state space
- states are essential snapshots of the memory of the program
  - memory is organised as a graph

```
template< typename T >
struct Node {
    Node(T val, Node *n = nullptr) :
        val(val), next(n)
    {}
    Node *next = nullptr;
    T val;
};
int main() {
    Node<int> *n1 = new Node<int>(1);
    Node<int> *n2 = new Node<int>(2, n1);
}
```



Efficiency of State Space Storage

- currently blocks of memory are stored deduplicated
- but minor change in a large block requires it to be stored whole again

Efficiency of State Space Storage

- currently blocks of memory are stored deduplicated
- but minor change in a large block requires it to be stored whole again
- there are compression methods available, the interesting part is using them in both memory and time efficient way

Efficiency of State Space Storage

- currently blocks of memory are stored deduplicated
- but minor change in a large block requires it to be stored whole again
- there are compression methods available, the interesting part is using them in both memory and time efficient way

- there is also possibility of optimizing the program in a way so that the state representation is smaller
  - detecting unchanging global variables
  - slicing-out provably unused variables
  - relaxing static single assignment form of LLVM in the internal representation
  - ...

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races
- not all interleavings are visibly different
  - not all actions of a thread are visible to other threads
  - local actions can stay hidden

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races
- not all interleavings are visibly different
  - not all actions of a thread are visible to other threads
  - local actions can stay hidden
  - the interesting part is detecting *local actions*

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races
- not all interleavings are visibly different
  - not all actions of a thread are visible to other threads
  - local actions can stay hidden
  - the interesting part is detecting *local actions*
- local actions can be anything
  - that does not interact with memory

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races
- not all interleavings are visibly different
  - not all actions of a thread are visible to other threads
  - local actions can stay hidden
  - the interesting part is detecting *local actions*
- local actions can be anything
  - that does not interact with memory
  - that does not interact with *globally visible* memory

Efficient Exploration of State Space

- in the case of parallelism, we need to explore all the different interleavings to detect errors caused by data races
- not all interleavings are visibly different
  - not all actions of a thread are visible to other threads
  - local actions can stay hidden
  - the interesting part is detecting *local actions*
- local actions can be anything
  - that does not interact with memory
  - that does not interact with *globally visible* memory
  - that interacts with globally visible memory *that is nevertheless only accessed by one thread*

Efficient Exploration of State Space II

- even with hiding of local actions, DIVINE explores state space eagerly
  - many interleavings can have the same results because threads do not actually interact

Efficient Exploration of State Space II

- even with hiding of local actions, DIVINE explores state space eagerly
  - many interleavings can have the same results because threads do not actually interact
- it should be possible to combine *(dynamic) partial order reduction* techniques with DIVINE
  - these techniques allow exploration of only a subset of state's successors if this subset is sufficient

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime
  - if interprocedural analysis is needed

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime
  - if interprocedural analysis is needed
  - if it is necessary to reorder or replace instructions

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime
  - if interprocedural analysis is needed
  - if it is necessary to reorder or replace instructions
  - to simplify the executor

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime
  - if interprocedural analysis is needed
  - if it is necessary to reorder or replace instructions
  - to simplify the executor
- any optimizations done by DIVINE have to be safe – must not change the verification outcome

Integration With Static Analysis and Optimizations

- in some cases it is better to aid reduction by static analysis and optimizations then calculate everything at runtime
  - can save work at runtime
  - if interprocedural analysis is needed
  - if it is necessary to reorder or replace instructions
  - to simplify the executor
- any optimizations done by DIVINE have to be safe – must not change the verification outcome
  - compiler optimizations are safe only for sequential programs
  - we need optimizations safe for all programs