# IA159 Formal Verification Methods

## Introduction

Jan Strejček

Department of Computer Science
Faculty of Informatics
Masaryk University

# Agenda

Agenda

- basic information about the course
- quick overview
- motivation

# What does “Formal Verification Methods” mean?

**Formal methods** are a collection of notations and techniques for describing and analyzing systems. Methods are **formal** in the sense that they are based on some mathematical theories, such as logic, automata or graph theory. [Pel01]

**Verification** is the process of applying a manual or an automatic technique that is supposed to establish whether the code either satisfies a given property or behaves in accordance with some higher-level description of it. [Pel01]

# What does “Formal Verification Methods” mean?

In the context of this course, formal verificatin methods are techniques (usually based on mathematical theories) for analysing systems with the aim to improve their quality and reliability.

# Focus of the course

- The course is focused on theoretical and algorithmic bases of verification methods.
- The software engineering aspects of verification methods are beyond the scope of this course.

# Literature

- There is no single reading material covering the course.
- Two main sources:
  - *D. A. Peled: Software Reliability Methods, Springer, 2001.*
  - *E. M. Clarke, O. Grumberg, and D. A. Peled: Model Checking, MIT, 1999.*
- Other sources (mainly recent journal or conference papers) will be referred.

# Connections to other courses

The course assumes familiarity with the following notions:

- IB005 Formal Languages and Automata I (aka FJA I)
  - pushdown automata
- IA006 Selected topics on automata theory (aka FJA II)
  - infinite words, Büchi automata, bisimulation equivalence
- IA040 Modal and Temporal Logics for Processes
  - temporal logics, mainly LTL
- IV113 Introduction to Validation and Verification
  - automata based LTL model checking

# Connections to other courses

Other relevant courses:

- MA015 Graph Algorithms
- IV010 Communication and Parallelism
- IB002 Design of Algorithms I
- IV022 Design and Verification of Algorithms
- PA008 Compiler Construction

# Connections to other courses

Courses following (is some sense) our course:

- IV101 Seminar on verification
- IV115 Parallel and Distributed Laboratory Seminar
- IV074 Laboratory for Parallel and Distributed Systems
- IA072 Seminar on Concurrency

# Examination

- There will be an oral exam at the end.
- No intrasemestral tests, no written exams, no homeworks.

# Overview of verification methods

# Basic verification methods

- testing
- deductive verification (with use of theorem provers)
- equivalence checking
- reachability and model checking
- static analysis and abstract interpretation
- symbolic execution

Other related techniques

- abstraction
- slicing
- SAT/SMT solving

# Abstraction

- reduces the size of systems to be analyzed
- can transform an infinite-state system into a finite one
- the set of system behaviours is usually increased (source of false alarms)

# Slicing

- reduces the size of systems on the level of source code
- the reduced system preserves values of given variables at given control locations
- M. Weiser: *Program Slicing*, IEEE Transactions on Software Engineering 10(4), 1984.

# Slicing: example

```
 1: char *copy(char *dst, char *src, int n, int *L) {
 2:     int i, len;
 3:     len = 0;
 4:     if (src != NULL && dst != NULL) {
 5:         len = n;
 6:         lock(L);
 7:     }
 8:     i = 0;
 9:     while (i < len) {
10:         dst[i] = src[i];
11:         i++;
12:     }
13:     if (len > 0) {
14:         unlock(L);
15:     }
16:     return dst;
17: }
```

Assume that we are interested only in values of lock `L` at the end of line 16.

# Slicing: example

```
 1: char *copy(char *dst, char *src, int n, int *L) {
 2:     int    len;
 3:     len = 0;
 4:     if (src != NULL && dst != NULL) {
 5:         len = n;
 6:         lock(L);
 7:     }
 8:
 9:
10:
11:
12:
13:     if (len > 0) {
14:         unlock(L);
15:     }
16:     return    ;
17: }
```

Assume that we are interested only in values of lock `L` at the end of line 16.

# SAT/SMT solving

SAT problem is to decide satisfiability of propositional logic formulae.

Satisfiability Modulo Theories (SMT) problem is to decide satisfiability of first-order logic formulea with respect to a given theory (e.g. theory of integers with addition and substraction).

- crucial for symbolic execution, abstraction, deductive verification
- A. R. Bradley amd Z. Manna: *The Calculus of Computation: Decision Procedures with Applications to Verification*, Springer, 2007.

# Testing

- simple, feasible, very good cost/performance ratio
- very effective in early stages of debugging process
- applicable directly to real systems
- cannot guarantee that there are no errors
- in practice: standard technique for enhancing the quality of systems, wide tool support

# Deductive verification

Deductive verification is a method for proving that, for any input values satisfying a given initial condition, a given program terminates and resulting variable values satisfy a given final assertion.

If initial condition $x2 > 0$ holds, then the execution of

```
y1=0;
y2=0;
while (y2 < x2) {
   y1 = y1 + x1;
   y2++;
}
```

always terminates and the resulting variable values satisfy final assertion $y1 = x1 * x2$ .

# Deductive verification

- applicable to models of real systems
- needs a huge effort of an expert on both deductive verification and systems under verification
- can guarantee that (a model of) a real system satisfies a given property
- in practice: used rarely (e.g. partial correctness of FPU in AMD processors)

# Equivalence checking

Equivalence checking decides whether two given systems are equivalent with respect to a given equivalence.

- applicable to models of real systems
- needs a detailed formal specification of a system under verification (or another “second system”)
- there are no algorithms for reasonable equivalences and infinite-state systems
- in practice: some specific applications (e.g. equivalence of different levels of hardware design)

# Reachability and model checking

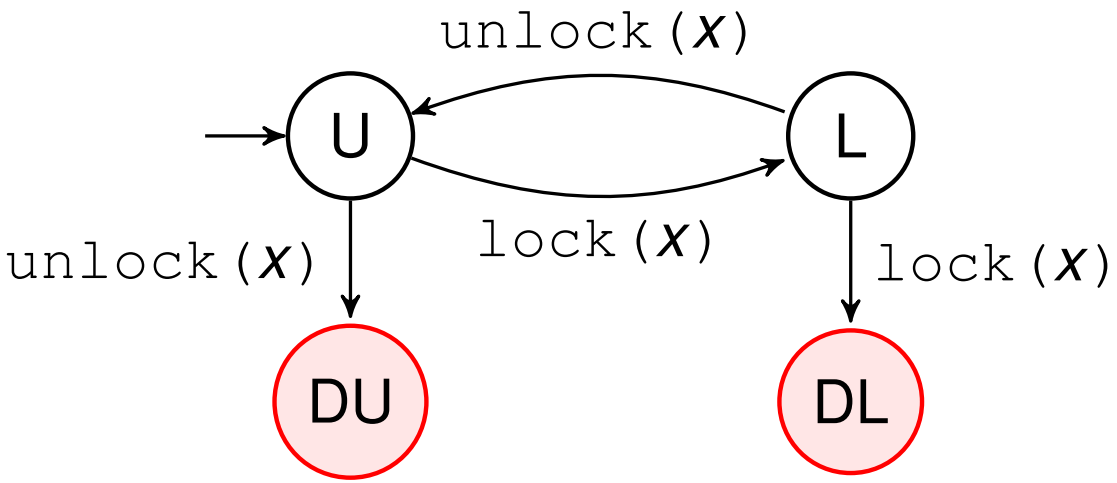
Reachability decides whether any execution of a given program can reach a given state. Model checking deficed whether each execution of a given program satisfies a given property (which is typically described by a temporal logic formula).

- applicable to (usually finite-state) models of real systems
- needs formal description of the property to be checked
- fully automatic, but feasible only for relatively small finite-state systems
- succesfull verification of real systems may require provision of a suitable abstraction
- in practice: a standard technique for verification of simple hardware designs, used also for verification of small systems (e.g. communication protocols)

# Static analysis and abstract interpretation

Static analysis and abstract interpretation is typically used to overapproximate or underapproximate a set of reachable states of selected program variables in each program location. The analyzed code is not executed.

Consider the following states of a lock $x$:




U = unlocked
L = locked

error states: DU = double unlock
DL = double lock

# Static analysis and abstract interpretation

```
 1: char *copy(char *dst, char *src, int n, int *L) {
 2:     int i, len;                                    // {U}
 3:     len = 0;                                       // {U}
 4:     if (src != NULL && dst != NULL) {              // {U}
 5:         len = n;                                   // {U}
 6:         lock(L);                                   // {L}
 7:     }                                              // {U,L}
 8:     i = 0;                                         // {U,L}
 9:     while (i < len) {                              // {U,L}
10:         dst[i] = src[i];                           // {U,L}
11:         i++;                                       // {U,L}
12:     }                                              // {U,L}
13:     if (len > 0) {                                 // {U,L}
14:         unlock(L);                                 // {DU,U}
15:     }                                              // {U,L}
16:     return dst;                                    // {U}
17: }
```

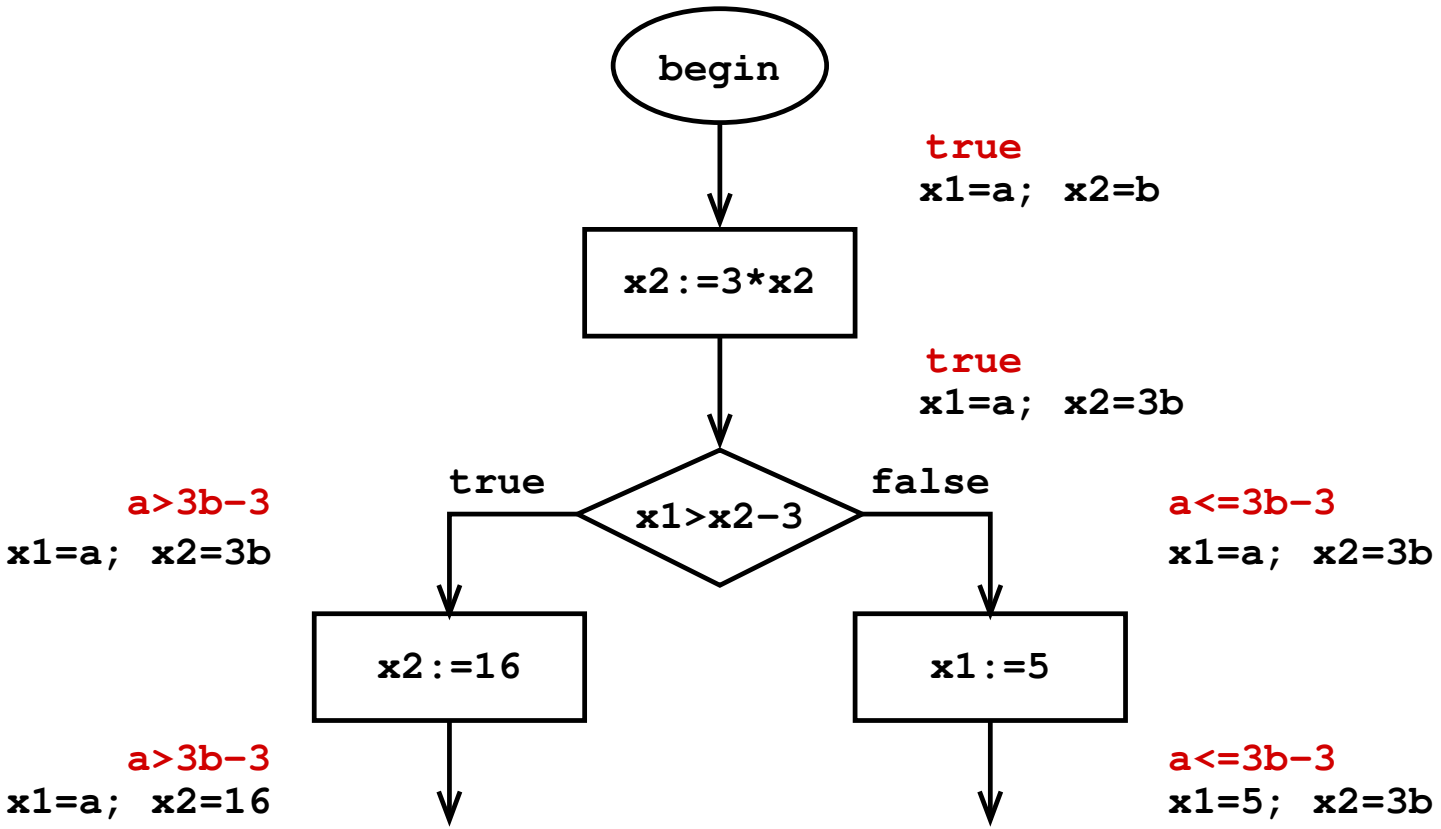
The indicated double unlock error is a false positive.

# Static analysis and abstract interpretation

- applicable directly to source code of real systems (or directly to executables)
- feasible
- can verify only a specific class of properties (including many interesting properties)
- may produce false alarms
- fully automatic
- in practice: some static analysis is performed by almost every compiler, there are very efficient tools (e.g. Coverity, CodeSonar, Stanse) able to work with big pieces of real software (e.g. Linux kernel)

# Symbolic execution

Symbolic execution executes the code on abstract symbols instead of input values.



# Symbolic execution

- can be seen as exhaustive testing
- applicable directly to source code of real systems (or directly to executables)
- fully automatic
- do not report false alarms
- feasible, but the computation usually did not finish due to large or even infinite number of execution paths
- in practice: several successful applications, but computational cost of pure symbolic execution is too high

# Combined methods

- popular combinations:
  - abstraction + model checking
  - model checking + counter-example guided abstraction refinement (CEGAR)
  - abstract interpretation + CEGAR
  - testing + model checking
  - testing + symbolic execution
- the aim is to develop methods which are automatic (as much as possible) and applicable directly to sources or binaries of real systems
- may be incomplete and/or produce some false alarms
- in practice: already has some specific applications in verification (e.g. verification of Windows drivers by Static Driver Verifier) and many applications in test-generation and bug-finding (e.g. SAGE, PEX, Coverity)
- combination of basic techniques is definitely the most promising approach

# Actual topics

- deductive software verification: verification of flowcharts, axiomatic program verification
- theorem prover ACL2 (with a demo)
- model checking of infinite-state systems (an overview)
- LTL model checking of pushdown systems
- abstraction and CEGAR
- symbolic execution (and whitebox fuzz testing)
- abstract interpretation

# Motivation

- Formal verification is used in Microsoft, Intel, AMD,...
- Formal verification is usually a supplementary method, the main methods are testing or simulation.
- In development of execution cluster of Core i7, formal verification has been used as a primary validation vehicle (simulation has been dropped)
- only 3 bugs escaped to silicon (2 other bugs were detected during the pre-silicon stage by full chip testing)
- this number is usually about 40
- the previous minimum is 11
- More information in Kaivola et al: *Replacing Testing with Formal Verification in Intel Core i7 Processor execution Engine Validation*, CAV 2009, LNCS 5643, Springer, 2009.

# Coming next week

## Deductive software verification

- prehistory of formal verification: 40+ years old technique!
- Does my program terminate? For all inputs?
- If yes, does it do what it is supposed to do?
- Can it be proven automatically?