IA159 Formal Verification Methods

Introduction

Jan Strejček

Faculty of Informatics
Masaryk University

Agenda

Agenda

- basic information about the course
- quick overview of formal methods
- selected topics

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 verification methods are techniques (usually based on mathematical theories) for analysing systems with the aim to improve their quality and reliability.

What does “Formal Verification Methods” mean?

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

In other words, methods that can find a bug or prove its absence.

Focus of the course

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

Literature

- Books (cover only some topics of the course):
  - *D. A. Peled: Software Reliability Methods, Springer, 2001.*
  - *E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and R. Bloem: Model Checking, Second Edition, MIT, 2018.*
  - *Ch. Baier and J.-P. Katoen: Principles of Model Checking, MIT, 2008.*
  - *E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem: Handbook of Model Checking, Springer, 2018.*
  - *D. S. Scott: The Seventeen Provers of the World, Springer, 2006.*
  - …
- Other sources (mainly journal or conference papers) will be referred and available in Study materials in IS.

Connections to other courses

Mandatory prerequisites

- IA169 System Verification and Assurance or
- IV113 Introduction to Validation and Verification († 2018)

Connections to other courses

Mandatory prerequisites

- IA169 System Verification and Assurance or
- IV113 Introduction to Validation and Verification († 2018)

Other relevant courses

- IA006 Selected Topics on Automata Theory (aka FJA II)
- IA040 Modal and Temporal Logics for Processes
- IV022 Design and Verification of Algorithms
- IV101 Seminar on Verification († 2015)

Examination

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

Overview of verification methods

Basic verification methods

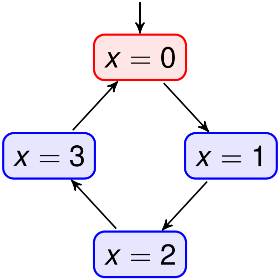
- testing
- deductive verification (with use of theorem provers)
- equivalence checking
- reachability analysis and model checking
- abstract interpretation and other static analyses
- symbolic execution

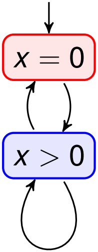
Other related techniques

- abstraction
- slicing
- SAT/SMT solving
- Craig interpolation

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 source code level
- 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 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.

SAT/SMT solving

SAT problem is to decide satisfiability of a given propositional logic formula.

Satisfiability Modulo Theories (SMT) problem is to decide satisfiability of a given first-order logic formula 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 and Z. Manna: *The Calculus of Computation: Decision Procedures with Applications to Verification*, Springer, 2007.

Craig interpolation

- if $\varphi \implies \psi$ then there exists an interpolant ρ such that $\varphi \implies \rho \implies \psi$ and ρ uses only propositional variables occurring in both φ and ψ
- ρ overapproximates φ and it is usually smaller than φ
- crucial for PDR/IC3, Ultimate Automizer, and many methods/tools using abstraction refinement
- W. Craig: *Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory*, The Journal of Symbolic Logic 22(3), 1957.

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 the 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

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 the 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 or small parts 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)
- tools: Coq, ACL2, Dafny, …

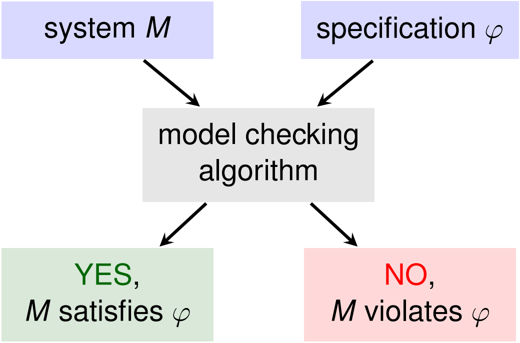
Equivalence checking

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

- applicable mainly 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 analysis and model checking

Reachability analysis decides whether any run of a given system can reach a given state. Model checking decides whether each run of a given system satisfies a given specification property (which is typically described by a temporal logic formula).



system M

specification φ

model checking algorithm

YES, M satisfies φ

NO, M violates φ

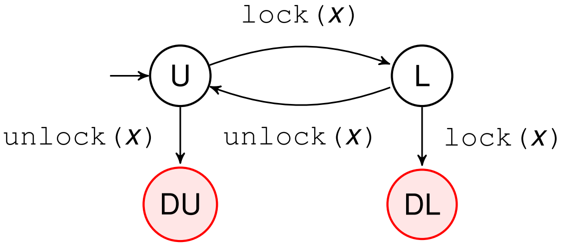
Reachability analysis and model checking

- needs formal description of the property to be checked
- fully automatic, but feasible mainly 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)
- tools: DIVINE, SPIN, NuSMV, . . .

Abstract interpretation and other static analyses

Abstract interpretation and other static analyses are typically used to overapproximate or underapproximate a set of reachable values 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

Abstract interpretation and other static analyses

```
char *copy(char *dst, char *src, int n, int *L) {
    int i, len;                                  U
    len = 0;                                     U
    if (src != NULL && dst != NULL) {            U
        len = n;                                 U
        lock(L);                                 L
    }                                           U,L
    i = 0;                                      U,L
    while (i < len) {                           U,L
        dst[i] = src[i];                        U,L
        i++;                                    U,L
    }                                           U,L
    if (len > 0) {                              U,L
        unlock(L);                             DU,U
    }                                           U,L
    return dst;                                 U,L
}
```

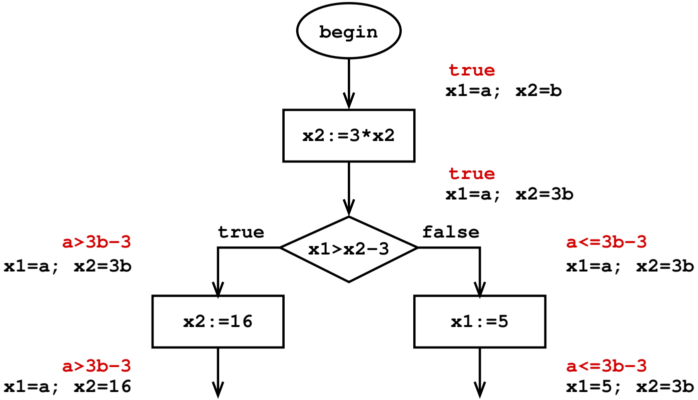
The indicated double unlock error is a false positive.

Abstract interpretation and other static analyses

- 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 many efficient tools able to work with big pieces of real software (e.g. Linux kernel)
- tools: Coverity, CodeSonar, . . .

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
- does 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
- tools: Klee, …

Combined methods

- popular combinations:
  - model checking + abstraction + counter-example guided abstraction refinement (CEGAR)
  - abstract interpretation + CEGAR
  - testing + model checking
  - testing + symbolic execution + Craig interpolation
- 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, CPAchecker, Ultimate Automizer) and many applications in test-generation and bug-finding (e.g. SAGE, PEX, CBMC)
- the most promising approaches usually combine several basic techniques

Verification of infinite-state systems

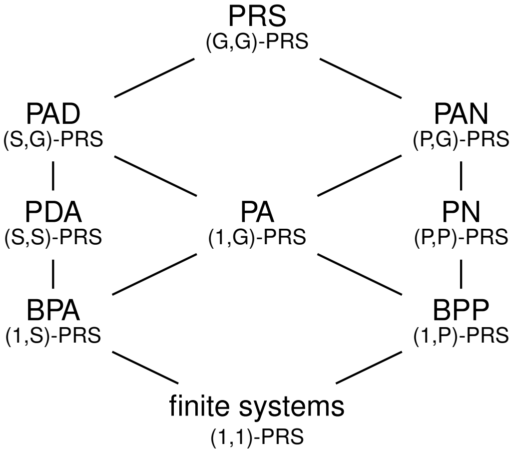
Finite vs. infinite-state systems

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

- verification of algorithm vs. verification of programs
- all verification problems are decidable for finite systems
- for infinite-state systems, decidability depends on the problem and type of the system
- explicit and symbolic (BDD-based) model checking applicable only to finite systems

PRS-hierarchy of infinite-state systems

The hierarchy compares expressive power of many classes of infinite-state systems including BPA, BPP, PA, Petri nets (PN), and pushdown processes (PDA). systems.



PRS
(G,G)-PRS

PAD
(S,G)-PRS

PAN
(P,G)-PRS

PDA
(S,S)-PRS

PA
(1,G)-PRS

PN
(P,P)-PRS

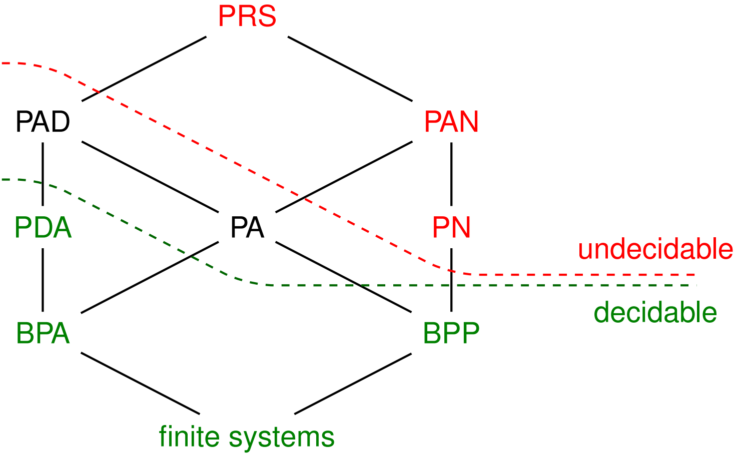
BPA
(1,S)-PRS

BPP
(1,P)-PRS

finite systems
(1,1)-PRS

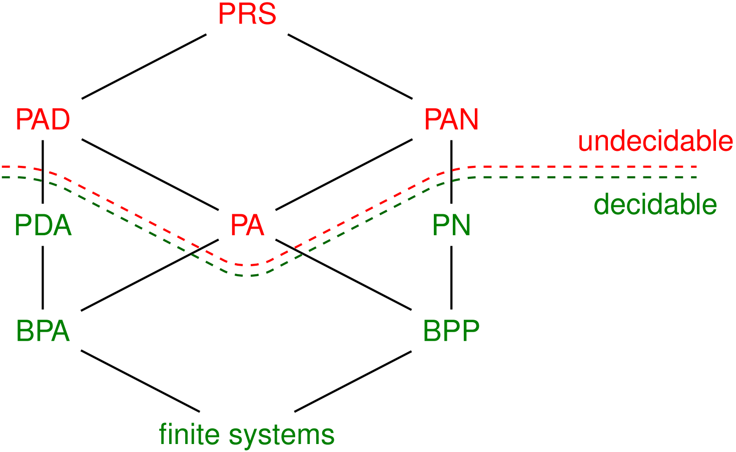
Decidability of equivalence checking

The decidability boundary of strong bisimulation in the PRS-hierarchy.



Decidability of model checking

The decidability boundary of the action-based LTL model checking in the PRS-hierarchy.



Actual topics of the course

- deductive verification
    - theorem prover ACL2 + Demo

- reachability analysis & verification of infinite-state systems
    - reachability analysis of pushdown systems
    - ~~LTL model checking of pushdown systems~~

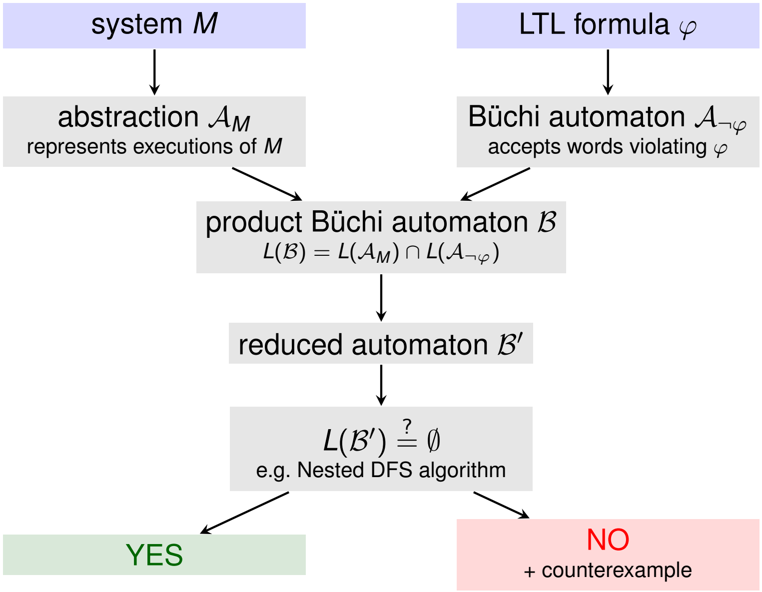
- LTL model checking
    - ~~translation of LTL to Büchi automata (via alternating aut.)~~
    - partial order reduction
    - abstraction and CEGAR

- static analysis
    - abstract interpretation
    - shape analysis (abs. int. of dynamic memory operations)

- Ultimate Automizer: verification via automata, symbolic execution, and interpolation

- property-directed reachability (PDR/IC3)

Automata-based LTL model checking of finite systems



system M

LTL formula φ

abstraction $\mathcal{A}_M$

represents executions of M

Büchi automaton $\mathcal{A}_{\neg\varphi}$

accepts words violating φ

product Büchi automaton $\mathcal{B}$

$L(\mathcal{B}) = L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\varphi})$

reduced automaton $\mathcal{B}'$

$L(\mathcal{B}') \stackrel{?}{=} \emptyset$

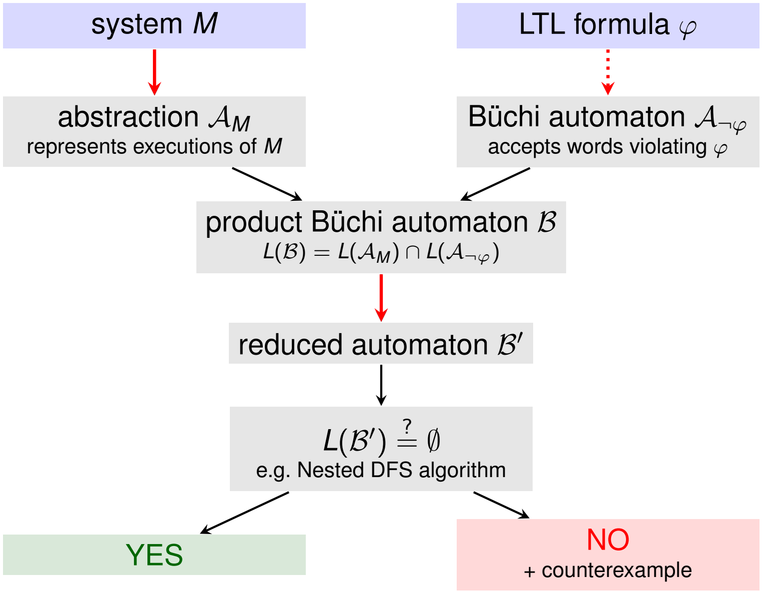
e.g. Nested DFS algorithm

YES

NO

+ counterexample

Automata-based LTL model checking of finite systems



system M

LTL formula φ

abstraction $\mathcal{A}_M$
represents executions of M

Büchi automaton $\mathcal{A}_{\neg\varphi}$
accepts words violating φ

product Büchi automaton $\mathcal{B}$
$L(\mathcal{B}) = L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\varphi})$

reduced automaton $\mathcal{B}'$

$L(\mathcal{B}') \overset{?}{=} \emptyset$
e.g. Nested DFS algorithm

YES

NO
+ counterexample

An extra piece of motivation

- Formal verification is used in Microsoft, Intel, facebook, ...
- 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

Theorem prover ACL2

`http://www.cs.utexas.edu/users/moore/acl2/`

- How it works?
- What is it good for?
- Including a live show!