Symbolic Execution

Ahmed Rezine

IDA, Linköpings Universitet

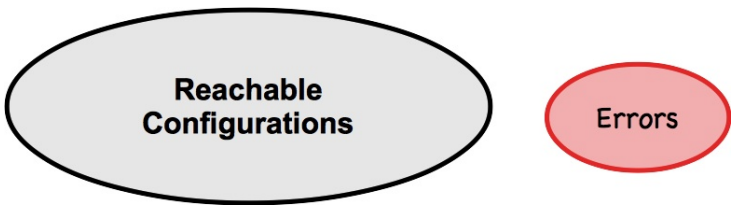
Hösttermin 2022

Outline

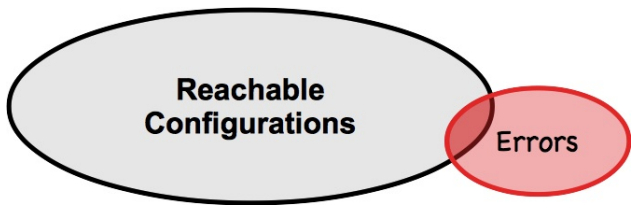
Outline

Program verification and Approximations

We often want to answer whether a program is **safe** or not (i.e., has some erroneous reachable configurations or not):



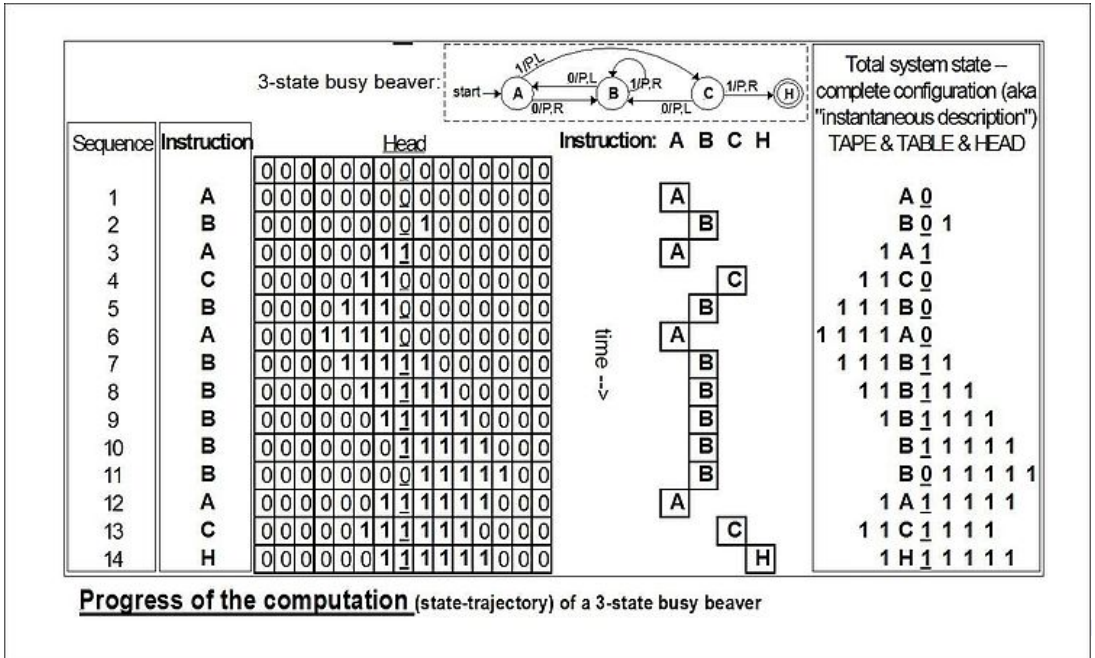
Safe Program



Unsafe Program

The general verification problem is "very difficult"

- Deciding whether all possible executions of the program are error-free is so hard that if we could write a program that could always do it for arbitrary computer-programs-to-be-analyzed then we would always be able to answer whether a Turing machine halts.
- This problem is proven to be undecidable, i.e., there is no algorithm that is guaranteed to terminate and to give an exact answer to the problem.



Problem is “very difficult”: what to do?

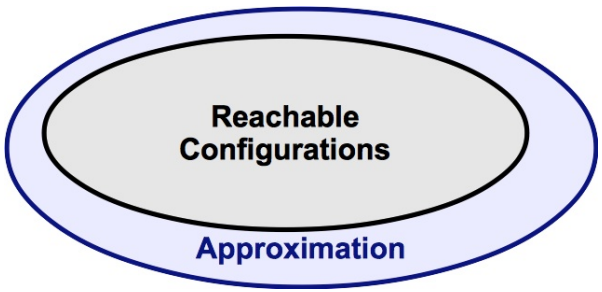
- Identify sub-problems on which one can decide: e.g. finite state machines, push-down automata, timed automata, Petri nets, well-structured transition systems.
- Proceed with approximations that will hopefully give some guarantees.

Verification problem and approximations

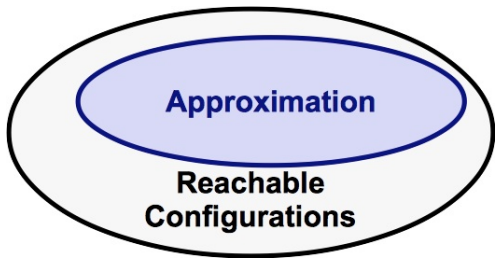
- An analysis procedure takes as input a program to be checked against a property. The procedure is an analysis algorithm if it is guaranteed to terminate.
- An analysis algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors (pessimistic analysis)
- An algorithm is **complete** in the case where each time it is given a program that is safe wrt. some errors, then it does report it to be safe wrt. those errors (optimistic analysis)
- In general, you have to give up on one of the three: termination, soundness or completeness.

Verification problem and approximations

- The idea is then to come up with efficient approximations to give correct answers in as many cases as possible.



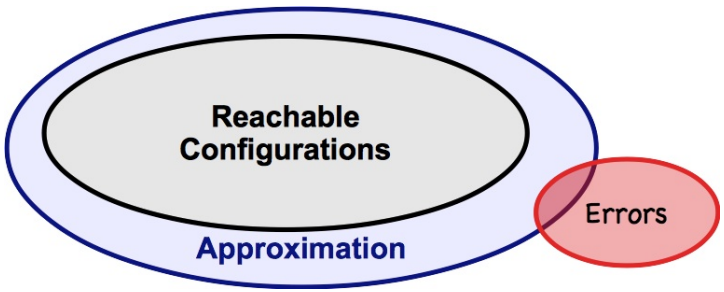
Over-approximation



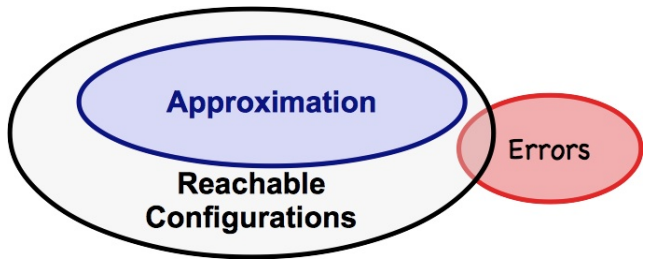
Under-approximation

Program verification and the price of approximations

- A sound analysis cannot give **false negatives**
- A complete analysis cannot give **false positives**



False Positive



False Negative

In this lecture

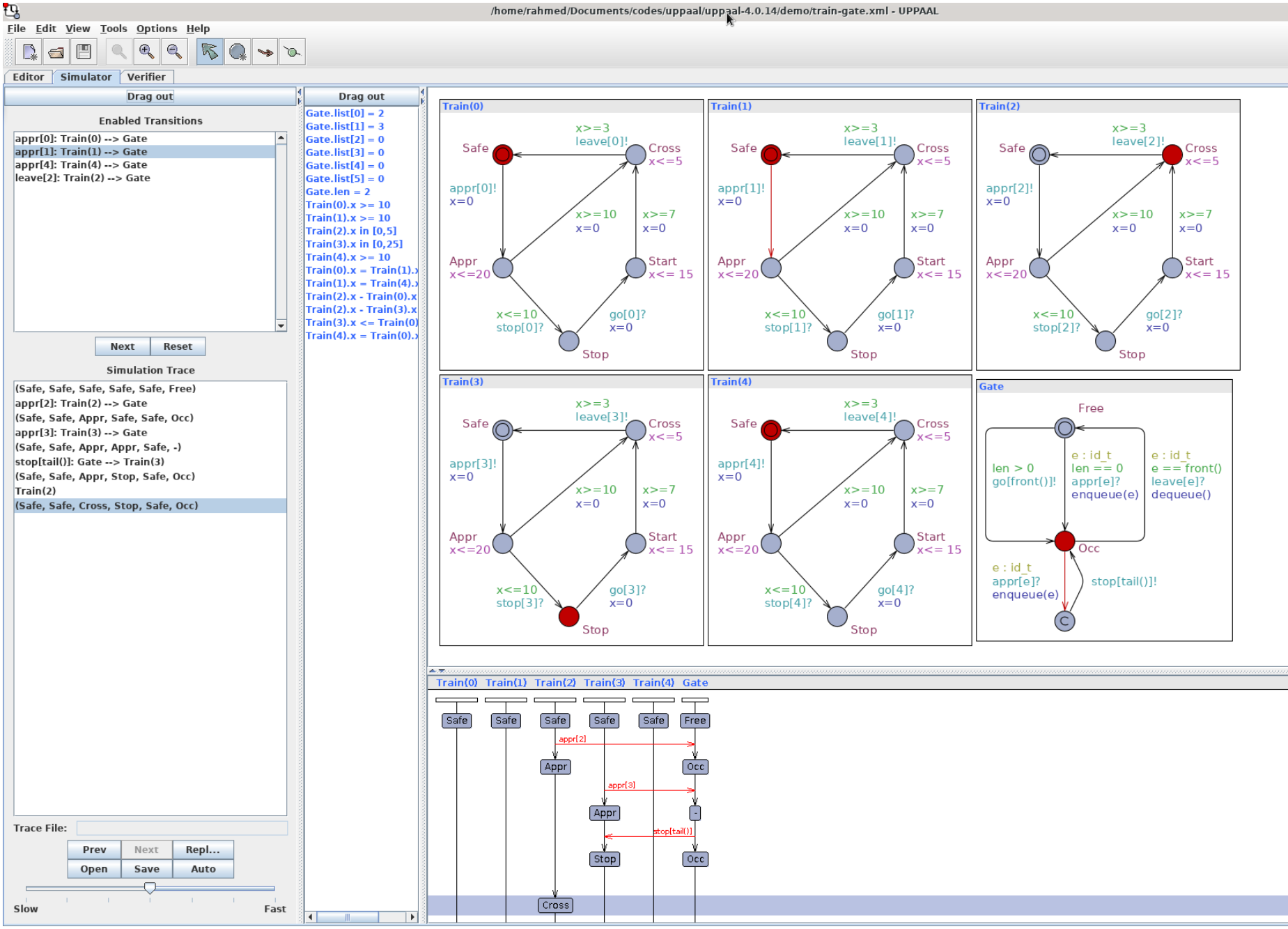
We will briefly introduce an under-approximation based verification method:

- Symbolic execution: partial, aims for completeness

More verification techniques (e.g., model checking, axiomatic reasoning, abstract interpretation): Software Verification (TDDE34)

Outline

The UPPAAL model checker



Outline

Testing

- Most common form of software validation
- Explores only one possible execution at a time
- For each new value, run a new test.
- On a 32 bit machine, `if(i==2022) bug()` would require 2^{32} different values to make sure there is no bug.
- The idea in symbolic testing is to associate **symbolic values** to the variables

Symbolic Testing

- Main idea by JC. King in "Symbolic Execution and Program Testing" in the 70s
- Use symbolic values instead of concrete ones
- Along the path, maintain a *Path Constraint* (PC) and a symbolic state (Σ)
- PC collects constraints on variables' values along a path,
- Σ associates variables to symbolic expressions,
- We get concrete values if PC is satisfiable
- The program can be run on these values
- Negate a condition in the path constraint to get another path

Introduction

Originates from automating proof-search for first order logic.

- Variables: $x, y, z, \ldots$
- Constants: $a, b, c, \ldots$
- N-ary functions: $f, g, h, \ldots$
- N-ary predicates: $p, q, r, \ldots$
- Atoms: $\bot, \top, p(t_1, \ldots, t_n)$
- Literals: atoms or their negation
- A FOL formula is a literal, boolean combinations of formulas, or quantified ($\exists$, $\forall$) formulas.

Evaluation of formula φ, with respect to interpretation I over non-empty (possibly infinite) domains for variables and constants gives true or false (resp. $I \models \varphi$ or $I \not\models \varphi$)

Satisfiability and Validity

A formula φ is:

- satisfiable if $I \models \varphi$ for **some** interpretation I
- valid if $I \models \varphi$ for **all** interpretations I

Satisfiability of FOL is undecidable. Instead, target decidable or domain-specific fragments.

Introduction

Given a quantifier free FOL formula and a combination of theories, is there an interpretation to the free variables that makes the formula true?

$$\varphi \triangleq g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

- EUF: Equality over Uninterpreted functions
- Satisfiable?

Introduction

Given a quantifier free FOL formula and a combination of theories, is there an interpretation to the free variables that makes the formula true?

$$\begin{aligned}\varphi \quad \triangleq \quad & (x_1 \geq 0) \wedge (x_1 < 1) \\ & \wedge((f(x_1) = f(0)) \Rightarrow (rd(wr(P, x_2, x_3), x_2 + x_1) = x_3 + 1)\end{aligned}$$

Introduction

Given a quantifier free FOL formula and a combination of theories, is there an interpretation to the free variables that makes the formula true?

$$\begin{aligned}\varphi \quad \triangleq \quad & (x_1 \geq 0) \wedge (x_1 < 1) \\ & \wedge((f(x_1) = f(0)) \Rightarrow (rd(wr(P, x_2, x_3), x_2 + x_1) = x_3 + 1)\end{aligned}$$

- Linear Integer Arithmetic (LIA)

Introduction

Given a quantifier free FOL formula and a combination of theories, is there an interpretation to the free variables that makes the formula true?

$$\begin{aligned}\varphi \quad \triangleq \quad & (x_1 \geq 0) \wedge (x_1 < 1) \\ & \wedge((f(x_1) = f(0)) \Rightarrow (rd(wr(P, x_2, x_3), x_2 + x_1) = x_3 + 1)\end{aligned}$$

- Linear Integer Arithmetic (LIA)
- Equality over Uninterpreted functions (EUF)
- Arrays (A)

Introduction

Given a quantifier free FOL formula and a combination of theories, is there an interpretation to the free variables that makes the formula true?

$$\begin{aligned}\varphi \quad \triangleq \quad & (x_1 \geq 0) \wedge (x_1 < 1) \\ & \wedge((f(x_1) = f(0)) \Rightarrow (rd(wr(P, x_2, x_3), x_2 + x_1) = x_3 + 1)\end{aligned}$$

- LIA: $x_1 = 0$
- EUF: $f(x_1) = f(0)$
- A: $rd(wr(P, x_2, x_3), x_2) = x_3$
- Bool: $rd(wr(P, x_2, x_3), x_2) = x_3 + 1$
- LIA: $\bot$

Introduction

- Sometimes more natural to express in logics other than propositional logic
- SMT decide satisfiablity of ground FO formulas wrt. background theory
- Many applications: Model checking, predicate abstraction, symbolic execution, scheduling, test generation, ...

Introduction: from SAT to SMT

- Eager approach with "bit-blasting" (UCLID):
  - Encode SMT formula in propositional logic
  - Use off-the-shelf SAT solver
  - Still dominant for bit-vector arithmetic
- Lazy-approach (CVC4, MathSat, Yices, Z3, ...)
  - Combine SAT (CDCL) and theory solvers
  - Sat-solver enumerates models for the boolean part
  - Theory solvers check satisfiability in the theory

Outline

SMT competition and SMTLIB

- Drive development, since 2005
- 15^{th} instance at `https://smt-comp.github.io/2020`
- Papers at SAT, CADE, CAV, FMCAD, TACAS, ...
- SMTLIB key initiative to promote common input and output for SMT solvers, benchmarks, tutorials, ...
- at `http://smtlib.cs.uiowa.edu/`

Equality with uninterpreted Functions (EUF)

- Consider $a * (f(b) + f(c)) = d \wedge b * (f(a) + f(c)) \neq d \wedge a = b$
- Formula is unsat, could be abstracted with
- $h(a, g(f(b), f(c))) = d \wedge h(b, g(f(b), f(c))) \neq d \wedge a = b$
- EUF used to abstracted non-supported theories such as non-linear multiplication or ALUs in circuits.

Arithmetic

Several restricted fragments, whether real or integer variables:

- Bounds $x \sim k$ with $\sim \in \{<, \leq, =, \geq, >\}$
- Difference logic $x - y \sim k$ with $\sim \in \{<, \leq, =, \geq, >\}$
- UTVPI $\pm x \pm y \sim k$ with $\sim \in \{<, \leq, =, \geq, >\}$
- Linear Arithmetic $x + 2y - 3z \leq 2$
- Non-linear arithmetic $xy - 4xy^2 + 2z \leq 2$

Arrays

- Special functions *read* and *write*
- Axioms:
  - $\forall a \forall i \forall v (read(write(a, i, v), i) = v)$
  - $\forall a \forall i \forall j \forall v (i \neq j \Rightarrow read(write(a, i, v), j)) = read(a, j))$
- Used for software (arrays) and hardware (memories) verification

Bit vectors

- Operations on vectors of bits
  - String like: concatenation, extraction, ...
  - Logical: bit-wise or, not, and...
  - Arithmetic: add, substract, multiply, ...
- $a[0:1] \neq b[0:1] \wedge (a|b) = c \wedge c[0] = 0 \wedge a[1] + b[1] = 0$

Symbolic Execution: a simple example

- Can we get to the ERROR? explore using SSA forms.
- Useful to check array out of bounds, assertion violations, etc.

```
1  foo(int x,y,z){
2    x = y - z;
3    if(x==z){
4      z = z - 3;
5      if(4*z < x + y){
6        if(25 > x + y) {
7          ...
8        }
9        else{
10         ERROR;
11       }
12     }
13   }
14   ...
```

$PC_1 = true$	
$PC_2 = PC_1$	$x \mapsto x_0, y \mapsto y_0, z \mapsto z_0$
$PC_3 = PC_2 \wedge x_1 = y_0 - z_0$	$x \mapsto (y_0 - z_0), y \mapsto y_0, z \mapsto z_0$
$PC_4 = PC_3 \wedge x_1 = z_0$	$x \mapsto (y_0 - z_0), y \mapsto y_0, z \mapsto z_0$
$PC_5 = PC_4 \wedge z_1 = z_0 - 3$	$x \mapsto (y_0 - z_0), y \mapsto y_0, z \mapsto (z_0 - 3)$
$PC_6 = PC_5 \wedge 4 * z_1 < x_1 + y_0$	$x \mapsto (y_0 - z_0), y \mapsto y_0, z \mapsto (z_0 - 3)$
$PC_{10} = PC_6 \wedge 25 \leq x_1 + y_0$	$x \mapsto (y_0 - z_0), y \mapsto y_0, z \mapsto (z_0 - 3)$

$PC = (x_1 = y_0 - z_0 \wedge x_1 = z_0 \wedge z_1 = z_0 - 3 \wedge 4 * z_1 < x_1 + y_0 \wedge 25 \leq x_1 + y_0)$

Check satisfiability with a solver (e.g., z3, cvc, yices, boolector,stp,...)

Symbolic execution today

- Leverages on the impressive advancements of SMT solvers
- Modern symbolic execution frameworks are not purely symbolic and are often dynamic: Sage, Klee (open source), Pex:
  - They can follow a concrete execution while collecting constraints along the way, or
  - They can treat some of the variables concretely, and some other symbolically
- This allows them to scale, to handle closed code or complex queries

Symbolic execution today

- C (actullay llvm) `http://klee.github.io/`
- Java (more than a symbolic executer)
  `http://babelfish.arc.nasa.gov/trac/jpf`
- C# (actually .net)
  `http://research.microsoft.com/en-us/projects/pex/`
- ...