Static Analysis: Symbolic Execution and Inductive Verification Methods

TDDC90: Software Security

Ahmed Rezine

IDA, Linköpings Universitet

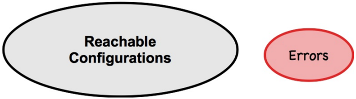
Hösttermin 2020

Outline

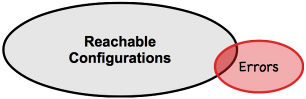
Outline

Static Program Analysis and Approximations

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



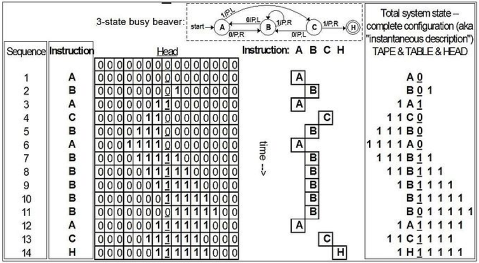
Safe Program



Unsafe Program

Static Program Analysis is a difficult problem

- Finding all configurations or behaviours (and hence errors) of arbitrary computer programs is so hard that if we could always do it (i.e., if we had an algorithm for it) 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.



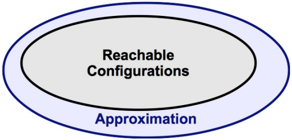
Progress of the computation (state-trajectory) of a 3-state busy beaver

Static Program Analysis and Approximations

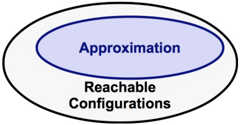
- An analysis procedure takes as input a program to be checked against a property. The analysis procedure is an analysis algorithm if it is guaranteed to terminate in a finite number of steps.
- An analysis algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors (informally, 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 (informally, optimistic analysis)

Static Program Analysis and Approximations

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



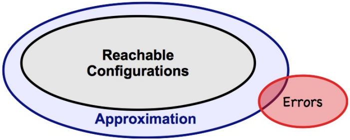
Over-approximation



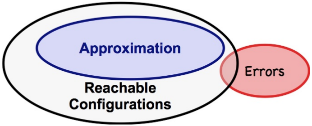
Under-approximation

Static Program Analysis and Approximations

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



False Positive



False Negative

Two Lectures on Static Analysis

These two lectures on static program analysis briefly introduce different types of analysis:

- Previous lecture:
  - syntactic analysis: scalable but neither sound nor complete
  - abstract interpretation sound but not complete
- This lecture:
  - symbolic executions: complete but not sound
  - inductive methods: may require heavy human interaction in proving the program correct

Two Lectures on Static Analysis

These two lectures on static program analysis briefly introduce different types of analysis:

- Previous lecture:
  - syntactic analysis: scalable but neither sound nor complete
  - abstract interpretation sound but not complete
- This lecture:
  - symbolic executions: complete but not sound
  - inductive methods: may require heavy human interaction in proving the program correct
- These two lectures are only appetizers:
  - There will be a deeper course with more tools and applications in the spring.
  - Possibilities of exjobbs with applications to verification and security.
  - Contact me if intreseted :-)

First, What are SMT Solvers?

- Stands for *Satisfiability Modulo Theory*
- Intuitively, these are constraint solvers that extend *SAT solvers* to richer theories
- Many solvers exist (Yices, CVC, STP, OpenSMT, Princess, Z3, etc),
- You will be using Z3 `https://github.com/Z3Prover/z3` in the lab (`http://rise4fun.com/z3` for a web interface)
- SAT solvers find a satisfying assignment to a formula where all variables are booleans or establishes its unsatisfiability
- SMT solvers find satisfying assignments to first order formulas where some variables may range over other values than just booleans

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: $\perp, \top, p(t_1, \ldots, t_n)$
- 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?

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

- 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: $\perp$

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, ...

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==2014) 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

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.

```
foo(int x,y,z){
  x = y - z;
  if(x==z){
    z = z - 3;
    if(4*z < x + y){
      if(25 > x + y) {
        ...
      }
      else{
        ERROR;
      }
    }
  }
  ...
```

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

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

Check satisfiability with a solver (e.g., `http://rise4fun.com/Z3`)

Symbolic execution today

- Leverages on the impressive advancements of SMT solvers
- Modern symbolic execution frameworks are not purely symbolic, and not necessarily purely static:
  - 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

Outline

Function Specifications and Correctness

- Contract between the caller and the implementation. **Total Correctness** requires that:
  - if the pre-condition (`-100 <= x && x <= 100`) holds
  - then the implementation terminates,
  - after termination, the following post-condition holds (`x>=0 && \result == x || x<0 && \result == -x`)
- **Partial Correctness** does not require termination

```
/*@ requires -100 <= x && x <= 100;
@ ensures x>=0 && \result == x || x<0 && \result == -x;
*/
int abs(int x){
  if(x < 0)
  return -x;
  return x;
}
```

Hoare Triples and Partial Correctness

- a Hoare triple $\{P\}$ *stmt* $\{R\}$ consists in:
  - a predicate pre-condition P
  - an instruction *stmt*,
  - a predicate post-condition R
- intuitively, $\{P\}$ *stmt* $\{R\}$ holds if whenever P holds and *stmt* is executed and terminates (**partial correctness**), then R holds after *stmt* terminates.
- For example:
  - $\{\mathit{true}\}\ x = y\ \{(x = y)\}$
  - $\{(x = 1) \wedge (y = 2)\}\ x = y\ \{(x = 2)\}$
  - $\{(x \geq 1)\}\ y = 2\ \{(x = 0) \vee (y \leq 10)\}$
  - $\{(x \geq 1)\}\ (\mathrm{if}(y == 2)\ \mathrm{then}\ x = 0)\ \{(x \geq 0)\}$
  - $\{\mathit{false}\}\ x = 1\ \{(x = 2)\}$

Weakest Precondition

- if $\{P\}$ *stmt* $\{R\}$ and $P' \Rightarrow P$ for any P' s.t. $\{P'\}$ *stmt* $\{R\}$, then P is the **weakest precondition** of R wrt. *stmt*, written $wp(stmt, R)$
- $wp(x = x + 1, x \geq 1) = (x \geq 0)$.
  $(x \geq 5), (x = 6), (x \geq 0 \wedge y = 8)$ are all valid preconditions, but they are not weaker than $x \geq 0$.
- Intuitively $wp(stmt, R)$ is the weakest predicate P for which $\{P\}$ *stmt* $\{R\}$ holds

Weakest Precondition of assignments

- $wp(x = E, R) = R[x/E]$, i.e., replace each occurrence of x in R by E.
- For instance:
  - $wp(x = 3, x == 5) = (x == 5)[x/3] = (3 == 5) = \mathit{false}$
  - $wp(x = 3, x >= 0) = (x >= 0)[x/3] = (3 >= 0) = \mathit{true}$
  - $wp(x = y + 5, x >= 0) = (x >= 0)[x/y + 5] = (y + 5 >= 0)$
  - $wp(x = 5 * y + 2 * z, x + y >= 0) = (x + y >= 0)[x/5 * y + 2 * z] = (6 * y + 2 * z >= 0)$

Weakest Precondition of sequences

- Assume a sequence of two instructions $stmt; stmt';$, for example $x = 2 * y; y = x + 3 * y;$
- the the weakest precondition is given by: $wp(stmt; stmt', R) = wp(stmt, wp(stmt', R))$,

$$\begin{array}{rl} & wp(x = 2 * y; y = x + 3 * y, y > 10) \\ = & wp(x = 2 * y, wp(y = x + 3 * y, y > 10)) \\ = & wp(x = 2 * y, (y > 10)[y/x + 3 * y]) \\ = & wp(x = 2 * y, x + 3 * y > 10) \\ = & (x + 3 * y > 10)[x/2 * y] \\ = & (2 * y + 3 * y > 10) \\ = & y > 2 \end{array}$$

Weakest Precondition of conditionals

- Assume a conditional (if(B) then *stmt* else *stmt'*), for example (if($x > y$) then $z = x$ else $z = y$)
- The weakest precondition is given by:
$$\left(\begin{array}{rl} & wp((\text{if}(B) \text{ then } stmt \text{ else } stmt'), R) \\ = & (B \Rightarrow wp(stmt, R))\&\&(!B \Rightarrow wp(stmt', R)) \end{array} \right)$$
- For example,
$$\begin{array}{rl} & wp((\text{if}(x > y) \text{ then } z = x \text{ else } z = y), z <= 10) \\ = & (x > y \Rightarrow wp(z = x, z <= 10)) \\ & \quad \&\&(x <= y \Rightarrow wp(z = y, z <= 10)) \\ = & (x > y \Rightarrow x <= 10)\&\&(x <= y \Rightarrow y <= 10) \end{array},$$

Hoare Triples for Loops, Partial Correctness

- In order to establish $\{P\}$ ($\texttt{while}(B)\texttt{do}\{stmt\}$) $\{R\}$, you will need to find an invariant *Inv* such that:
  - $P \Rightarrow Inv$
  - $\{Inv\&\&B\}$ $stmt$ $\{Inv\}$
  - $(Inv\&\&!B) \Rightarrow R$
- For example $\{i == j == 0\}$ ($\texttt{while}(i < 10)\texttt{do}\{i = i + 1; j = j + 1\}$) $\{j == 10\}$, we need to find *Inv* such that:
  - $(i == j == 0) \Rightarrow Inv$
  - $\{Inv\&\&(i < 10)\}$ $i = i + 1; j = j + 1$ $\{Inv\}$
  - $(Inv\&\&i >= 10) \Rightarrow j == 10$

Hoare Triples for Loops, Total Correctness

- $\{P\}$ $(\texttt{while}(B)\texttt{do}\{stmt\})$ $\{R\}$
- Partial correctness: if we start from P and $(\texttt{while}(B)\texttt{do}\{stmt\})$ terminates, then R terminates.
  - $P \Rightarrow Inv$
  - $\{Inv\&\&B\}$ $stmt$ $\{Inv\}$
  - $(Inv\&\&!B)\Rightarrow R$
- Total correctness: the loop does terminate: find a **variant function** v such that:
  - $(Inv\&\&B) \Rightarrow (v > 0)$
  - $\{Inv\&\&B\&\&v = v_0\}$ $stmt$ $\{v < v_0\}$
- For example $(\texttt{while}(i < 10)\texttt{do}\{i = i + 1; j = j + 1\})$ can be shown to terminate with $v = (10 - i)$ and $Inv = (i <= 10)$