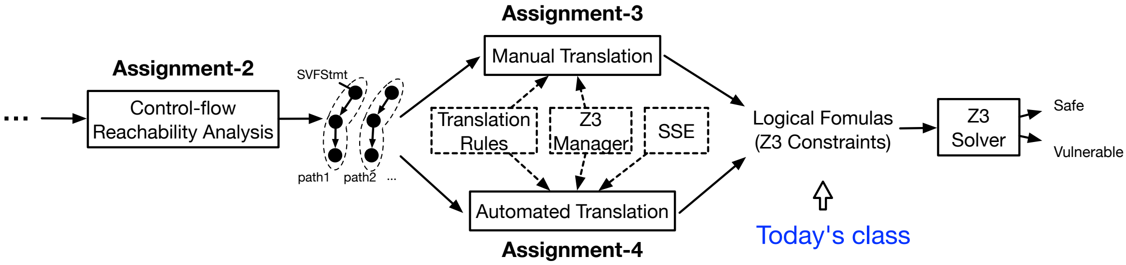
# Code Verification and Predicate Logic

Yulei Sui

University of Technology Sydney, Australia

# Today’s class




- In **this week**, we will learn the (first-order) **logical formulas**, which are the outputs translated from code for verification.
- We will take a look at **manual and automated translation** from code to formulas for verification **from next class**.

# Formal Verification For Code

$$\text{Specification} \overset{?}{\equiv} \text{code implementation}$$

# Formal Verification For Code

$$\begin{array}{ccc}
\text{Specification} & \stackrel{?}{\equiv} & \text{code implementation} \\
\Downarrow \text{translate} & & \Downarrow \text{translate} \\
\text{logical formulas of specification} & \stackrel{?}{\equiv} & \text{Logical formulas of code implementation.}
\end{array}$$

# Formal Verification For Code

$$
\begin{array}{ccc}
\text{Specification} & \stackrel{?}{\equiv} & \text{code implementation} \\
\Downarrow \text{translate} & & \Downarrow \text{translate} \\
\text{logical formulas of specification} & \stackrel{?}{\equiv} & \text{Logical formulas of code implementation.}
\end{array}
$$

- Proving the correctness of your code given a specification (or spec) using formal methods of mathematics
- Make the connection between specifications and implementations rigid, reliable and secure by translating specification and code into logical formulas.
- The application of theorem proving tools to perform satisfiability checking of logical formulas.

# Specification[2]

- Specifications **independent of** the source code
  - Formal specification in a separate file from source code and written in a specification language and accepted by theorem provers
- Specifications **embedded in** the source code (This subject)
  - `assert` embedded in the program following the Hoare triple form.
- Hoare logic[1]: $P\ \{prog\}\ Q$.
  - $P$ is the **pre-condition** (`assume`), expressed by predicate logic (first-order logic) formula
  - $Q$ is the **post-condition** (`assert`)
  - *prog* is the target program
  - The Hoare triple describes that when the precondition is met, executing the program *prog* establishes the postcondition.

[1] https://en.wikipedia.org/wiki/Hoare_logic
[2] https://www.hillelwayne.com/post/why-dont-people-use-formal-methods

# Assertion-Based Specification and Satisfiability

Prove whether the post-condition (`assert`) holds after executing the program given the pre-condition (`assume`).

```
assume(100 > x > 0);  // P
    foo(x){
        if(x > 10) {
            y = x + 1;
        }
        else {
            y = 10;
        }
    }
assert(y >= x + 1);  // Q
```

translate $\Longrightarrow$ $\phi(P\{\texttt{foo}\}Q)$ feed into $\Longrightarrow$ SAT/SMT Solver

logical formula

Will the assertion hold?

# Satisfiability Checking

Assertion verification as satisfiability checking. The assertion holds if the formula $\phi(P\{\texttt{foo}\}Q)$ is satisfiable (SAT).

- $\phi$ is satisfiable if a program *prog* is correct for all valid inputs.

$$\forall \texttt{x}\ \forall \texttt{y}\ \ P(\texttt{x}) \wedge S_{prog}(\texttt{x}, \texttt{y}) \rightarrow Q(\texttt{x}, \texttt{y})$$

  - $P(x)$ is the pre-condition formula over variables $\texttt{x}$, i.e., $\texttt{100} > \texttt{x} > \texttt{0}$.
  - $S_{prog}(\texttt{x}, \texttt{y})$ is the formula representing *prog* which accepts $\texttt{x}$ as its input, and terminates with output $\texttt{y}$.
  - $Q(\texttt{x}, \texttt{y})$ is the post-condition formula over variables $\texttt{x}, \texttt{y}$, i.e., $\texttt{y} >= \texttt{x} + \texttt{1}$.

[3] https://en.wikipedia.org/wiki/Logical_equivalence

# Satisfiability Checking

Assertion verification as satisfiability checking. The assertion holds if the formula $\phi(P\{\texttt{foo}\}Q)$ is satisfiable (SAT).

- $\phi$ is satisfiable if a program *prog* is correct for all valid inputs.

$$\forall \mathrm{x}\ \forall \mathrm{y} \quad P(\mathrm{x}) \wedge S_{prog}(\mathrm{x}, \mathrm{y}) \rightarrow Q(\mathrm{x}, \mathrm{y})$$

  - $P(x)$ is the pre-condition formula over variables $\mathrm{x}$, i.e., $100 > \mathrm{x} > 0$.
  - $S_{prog}(\mathrm{x}, \mathrm{y})$ is the formula representing *prog* which accepts $\mathrm{x}$ as its input, and terminates with output $\mathrm{y}$.
  - $Q(\mathrm{x}, \mathrm{y})$ is the post-condition formula over variables $\mathrm{x}, \mathrm{y}$, i.e., $\mathrm{y} >= \mathrm{x} + 1$.
- How to prove correctness for all inputs $\mathrm{x}$? Search for counterexample $\mathrm{x}$ where $\phi$ does not hold, that is

$$\begin{aligned} & \exists \mathrm{x}\ \exists \mathrm{y} \quad \neg(P(\mathrm{x}) \wedge S_{prog}(\mathrm{x}, \mathrm{y})) \rightarrow Q(\mathrm{x}, \mathrm{y})) \\ \Rightarrow \quad & \exists \mathrm{x}\ \exists \mathrm{y} \quad P(\mathrm{x}) \wedge S_{prog}(\mathrm{x}, \mathrm{y}) \wedge \neg Q(\mathrm{x}, \mathrm{y}) \qquad \text{(simplification}^{3}\text{)} \end{aligned}$$

Note that $P(\mathrm{x})$ is always true if a program has no pre-condition.

[3] https://en.wikipedia.org/wiki/Logical_equivalence

# Satisfiability Checking

Checking whether the logical formula $\phi$ is satisfiable by an SAT/SMT solver.

```
assume(100 > x > 0);
foo(x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
}
assert(y >= x + 1);
```

$$\xrightarrow{\text{translate}} \underbrace{\exists x\ \exists y\ \ P(x) \land S_{prog}(x, y)) \land \neg Q(x, y)}_{\text{logical formula } \phi} \xrightarrow{\text{feed into}} \text{SAT/SMT Solver}$$

# Satisfiability Checking

Checking whether the logical formula $\phi$ is satisfiable by an SAT/SMT solver.

```
assume(100 > x > 0);
foo(x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
}
assert(y >= x + 1);
```

$\xrightarrow{\text{translate}}$ $\underbrace{\exists x\ \exists y\ \ P(x) \wedge S_{prog}(x, y)) \wedge \neg Q(x, y)}_{\text{logical formula } \phi}$ $\xrightarrow{\text{feed into}}$ SAT/SMT Solver

Unsatisfiable! **couterexample** `x = 10` **found**!

# Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula $\phi$, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$

    $\phi(path_1)$: $\quad \exists \mathtt{x}\ P(\mathtt{x}) \land \big((\mathtt{x} > 10) \land (\mathtt{y} \equiv \mathtt{x} + 1)\big) \land \neg Q(\mathtt{x}, \mathtt{y})$ (`if` branch of `foo`)

    $\phi(path_2)$: $\quad \exists \mathtt{x}\ P(\mathtt{x}) \land \big((\mathtt{x} \leq 10) \land (\mathtt{y} \equiv 10)\big) \land \neg Q(\mathtt{x}, \mathtt{y})$ (`else` branch of `foo`)

# Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula $\phi$, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$

    $\phi(path_1)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} > \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{x} + \mathtt{1})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`if` branch)

    $\phi(path_2)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} \leq \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{10})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`else` branch)
  - $\phi(path_2)$ : **has a counterexample** `x = 10`!!

# Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula $\phi$, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$

    $\phi(path_1)$:$\exists x (100 > x > 0) \wedge \big((\mathtt{x} > \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{x} + \mathtt{1})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1}) \quad$ (`if` branch)

    $\phi(path_2)$:$\exists x (100 > x > 0) \wedge \big((\mathtt{x} \leq \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{10})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1}) \quad$ (`else` branch)
  - $\phi(path_2)$ : **has a counterexample** `x = 10`!!
  - **Manual translation** via theorem prover APIs (e.g., Z3 APIs) (Assignment-3)
  - **Automatic translation** during symbolic execution (Assignment-4)

# Proving each $\phi(path)$ formula

Exhaustively by not finding counterexamples via mathematical proofs.

- Direct Proof
- Proof by contradiction
- Proof by induction
- Proof by construction
- ...

# Proving each $\phi(path)$ formula

Exhaustively by not finding counterexamples via mathematical proofs.

- Direct Proof
- Proof by contradiction
- Proof by induction
- Proof by construction
- ...

**Manual proof** is **impractical** for software verification!

- Too many variables and logical relations between them (state exploration)
- Too many assertions as specifications.

This subject **does not** focus on formal mathematical proof, but rather the **application** of **automated theorem prover** tools

- Translating code into logical formulas
- Feeding the logical formulas into a theorem prover to check the satisfiability.

# Theorem Prover Tools [4]

- Interactive theorem provers (proof assistant)
  - Formal proofs by human-machine collaboration via expressive specification language and may not work directly on source code.
  - For example, ACL2, Coq, Isabelle and HOL provers
- Automated theorem provers
  - Proof automation (but less expressive than interactive provers) and can work on real-world source code.
  - For example, Z3 and CVC

[4] https://en.wikipedia.org/wiki/Theorem_prover

# Automated Theorem Provers

A prover/solver checks if a formula $\phi(P\{\texttt{foo}\}Q)$ is satisfiable (SAT).

- If yes, the solver returns a **model** *m*, a valuation of $\texttt{x}, \texttt{y}, \texttt{z}$ of `foo` that satisfies $\phi$ (i.e., *m* makes $\phi$ true).
- Otherwise, the solver returns unsatisfiable (UNSAT)

**SAT** vs. **SMT** solvers

- **SAT** solvers accept **propositional logic** (Boolean) formulas, typically in the conjunctive normal form (CNF).
- **SMT** (satisfiability modulo theories) solvers generalize the Boolean satisfiability problem (SAT), and accept more expressive **predicate logic** formulas, i.e., propositional logic with predicates and quantification.
  - Z3 Automated Theorem Prover[5], a cross-platform satisfiability modulo theories (SMT) solver developed by Microsoft (This subject).
  - More details next week..

[5] https://github.com/Z3Prover/z3/wiki#background