Code Verification and Automated Theorem Prover

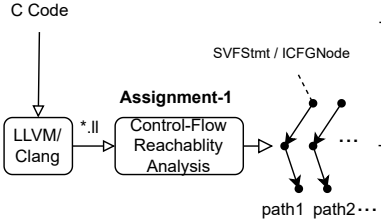
(Week 4)

Yulei Sui

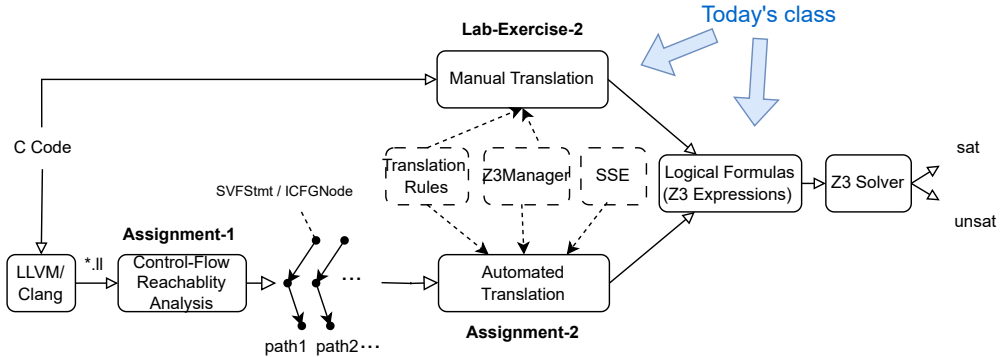
School of Computer Science and Engineering

University of New South Wales, Australia

Today's class



Today's class



- In Lab-Exercise-2 and Assignment-2, we will conduct code verification to **prove code assertions** on top of reachability analysis (Assignment-1).
- Translating **C statements (Lab-Exercise-2)** and **SVFStmt/ICFGNode (Assignment-2)** to **logical formulas/expressions** and solve them to verify code assertions using automated theorem prover (i.e., Z3)

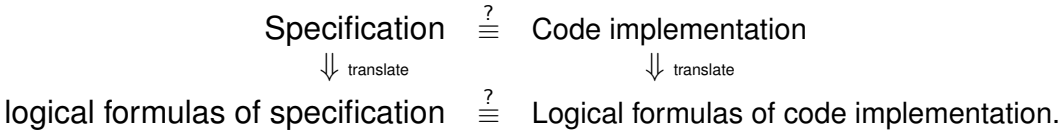
Formal Verification For Code

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

Formal Verification For Code

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

Formal Verification For Code



- 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

- Specifications **independent of** the source code
  - Formal specification in a separate file from the source code, written in a specification language and accepted by theorem provers

Specification

- Specifications **independent of** the source code
  - Formal specification in a separate file from the source code, written in a specification language and accepted by theorem provers
- Specifications **embedded in** the source code (This course)
  - `assume(expr)`: an assumed **precondition** of a program that expression `expr` always be true and uses this assumed knowledge to execute the program. `assmue` is often **optional** as programs or verification scenarios may not have preconditions, including Lab-Exercise-2 and Assignment-2.
  - `assert(expr)`: an expected **postcondition** embedded in the program to check that `expr` always holds for any execution, otherwise the program terminates. We use `svf_assert` in our lab/assignment as an alternative for verification purposes.

Specification

- Specifications **independent of** the source code
  - Formal specification in a separate file from the source code, written in a specification language and accepted by theorem provers
- Specifications **embedded in** the source code (This course)
  - `assume(expr)`: an assumed **precondition** of a program that expression `expr` always be true and uses this assumed knowledge to execute the program. `assmue` is often **optional** as programs or verification scenarios may not have preconditions, including Lab-Exercise-2 and Assignment-2.
  - `assert(expr)`: an expected **postcondition** embedded in the program to check that `expr` always holds for any execution, otherwise the program terminates. We use `svf_assert` in our lab/assignment as an alternative for verification purposes.
- Hoare logic triple $P\ \{prog\}\ Q$ representing a program expressed by a predicate (first-order) logic, describes that when the precondition P is met, executing the program *prog* establishes the postcondition Q.

Hoare logic: `https://en.wikipedia.org/wiki/Hoare_logic`

Formal specifications: `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 ⟹ $\psi(P\{\texttt{foo}\}Q)$ (logical formula) ⟹ feed into SAT/SMT Solver

Will the assertion hold?

Assertion as Specification in Code Verification

Assertions are program statements used to **test assumptions** made by software designers/programmers/testers. An assertion is a predicate or an expression that **always should evaluate to true** at that point during code execution.

```
assert(expr);

or

svf_assert(expr);
```

unfold →

```
if(expr is true){
    // continue normal execution
}
else{
    __assert_fail();
    // program failure and terminate the program
}
```

Assertions as Specifications in Code Verification (Example)

- Assertions are program statements used to test **program semantics and designs** made by software designers/programmers.
- An assertion is an expression/predicate that **always should evaluate to true** at that point during code execution.

Assertions as Specifications in Code Verification (Example)

- Assertions are program statements used to test **program semantics and designs** made by software designers/programmers.
- An assertion is an expression/predicate that **always should evaluate to true** at that point during code execution.

```
1  #include <assert.h>
2  void main() {
3      int x_axis = 5;
4      int y_axis = 1;
5      // assertion succeeds
6      assert(x_axis > 0 && y_axis > 0);
7      plot(x_axis, y_axis);
8      int y_axis = x_axis - 5;
9      // assertion fails and program crashes
10     assert(x_axis > 0 && y_axis > 0);
11 }
```

Assertions as Specifications in Code Verification (Example)

- Assertions are program statements used to test **program semantics and designs** made by software designers/programmers.
- An assertion is an expression/predicate that **always should evaluate to true** at that point during code execution.

```c
#include <assert.h>
void main() {
    int x_axis = 5;
    int y_axis = 1;
    // assertion succeeds
    assert(x_axis > 0 && y_axis > 0);
    plot(x_axis, y_axis);
    int y_axis = x_axis - 5;
    // assertion fails and program crashes
    assert(x_axis > 0 && y_axis > 0);
}
```

```c
void display_number(int* myInt) {
  assert(myInt != nullptr);
  printf("%d", *myInt);
}
int main () {
  int myptr = 5;
  int* first_ptr = &myptr;
  int* second_ptr = nullptr;
  // assertion succeeds
  display_number(first_ptr);
  // assertion fails and program crashes
  display_number(second_ptr);
}
```

Assertion-Based Code Verification

Assertions can be seen as partial software specifications that

- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertion-Based Code Verification

Assertions can be seen as partial software specifications that

- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertions are typically used in the following scenarios.

- Software **developers** can add assertions during their programming or implementation to verify their expected results.
- Software **testers** can add assertions as a part of the unit testing process.
- Project **managers** or third parties can add assertions in the middle or end of a program execution to verify and understand code bases.

Satisfiability Solving as Logic Inference Problem

Logic Inference Problem:

- Given Hoare logic triple $P\{prog\}Q$ expressed by a set of constraints (logical formulas) extracted from code statements of a program. We can express $P\{prog\}$ as KB **knowledge base** or **premises**, and Q is the **conclusion**. For example

Satisfiability Solving as Logic Inference Problem

Logic Inference Problem:

- Given Hoare logic triple $P\{prog\}Q$ expressed by a set of constraints (logical formulas) extracted from code statements of a program. We can express $P\{prog\}$ as KB **knowledge base** or **premises**, and Q is the **conclusion**. For example
  - KB : $((\mathtt{x} > \mathtt{z}) \wedge (\mathtt{y} == \mathtt{x} + 1)) \vee ((\mathtt{x} \leq \mathtt{z}) \wedge (\mathtt{y} == 10))$
  - Q : $\mathtt{y} \geq \mathtt{x} + 1$
- $KB \vdash Q$?
  - Does KB semantically entail Q?
  - If all constraints in KB are true, is the assertion true?
  - Is the specification Q satisfiable given constraints from code?
- Each element (**proposition** or **predicate**) in KB can be seen as one premise and Q is the conclusion.

Propositional Logic (Statement Logic)

A **proposition** is a statement that is either true or false. Propositional logic studies the ways statements can interact with each other.

- **Propositional variables** (e.g., S) represent propositions or statements in the formal system.
- A **propositional formula** is logical formula with **propositional variables** and **logical connectives** like `and` ($\wedge$) , `or` ($\vee$), `negation` ($\neg$), `implication` ($\Rightarrow$)
- **Inference rules** allow certain formulas to be derived. These derived formulas are called **theorems** (or true propositions). The derivation can be interpreted as proof of the proposition represented by the theorem.

`https://en.wikipedia.org/wiki/Propositional_calculus`

`http://discrete.openmathbooks.org/dmoi2/sec_propositional.html`

Predicate Logic (First-Order Logic)

First-order logic is propositional logic with predicates and quantification.

- **Propositional logic**: boolean logic which represents statements without reflecting their structures and relations
- **Predicate logic**: is more expressive and further analyzes proposition(s) by representing their entities' properties and relations and to group entities, i.e., additionally covers predicates and quantification.

Predicate Logic (First-Order Logic)

First-order logic is propositional logic with predicates and quantification.

- **Propositional logic**: boolean logic which represents statements without reflecting their structures and relations
- **Predicate logic**: is more expressive and further analyzes proposition(s) by representing their entities' properties and relations and to group entities, i.e., additionally covers predicates and quantification.
- A **predicate** P takes one or more variables/entities as input and outputs a proposition and has a truth value (either true or false).
  - A statement whose truth value is dependent on variables.
  - For example, in $P(x) : x > 5$, "x" is the variable and "> 5" is the predicate. After assigning x with the value 6, $P(x)$ becomes a proposition $6 > 5$.
- A **quantifier** is applied to a set of entities
  - Universal quantifier $\forall$, meaning all, every
  - Existential quantifier $\exists$, meaning some, there exists

https://en.wikipedia.org/wiki/First-order_logic https://www.youtube.com/watch?v=ARywou8HLQk

Predicate Logic (Natural Language Example)

Consider the two statements

- “Jack got a high distinction”
- “Peter got a high distinction”

In propositional logic, these statements are viewed as being unrelated and the sub-statements/words/entities are not further analyzed.

- **Predicate logic** allows us to define a **predicate** P representing “got a high distinction” which occurs in both sentences.
- $P(x)$ is the **predicate logic statement (formula)** which accepts a name x and output as “x got a high distinction”.

Predicate Logic (Code Example)

- S_1: x $>$ 20;
- S_2: x $>$ 10;
- $S_2 \rightarrow Q$: if(x $>$ 10) y = 15;
- Q: y $=$ 15;
- In **propositional logic**, each statement (including its variables and constants) is viewed as one proposition. Their relations are not further analyzed.
  - Given propositions S_1 and $S_2 \rightarrow Q$ as the knowledge base KB. Does the following semantically entail $\{S_1, S_2 \rightarrow Q\} \vdash Q$ or $(S_1 \wedge S_2 \rightarrow Q) \rightarrow Q$ hold?
  - Answer: No!

Predicate Logic (Code Example)

- S_1: x > 20;
- S_2: x > 10;
- $S_2 \rightarrow Q$: if(x > 10) y = 15;
- Q: y = 15;
- In **propositional logic**, each statement (including its variables and constants) is viewed as one proposition. Their relations are not further analyzed.
  - Given propositions S_1 and $S_2 \rightarrow Q$ as the knowledge base KB. Does the following semantically entail $\{S_1, S_2 \rightarrow Q\} \vdash Q$ or $(S_1 \wedge S_2 \rightarrow Q) \rightarrow Q$ hold?
  - Answer: No!
- **Predicate logic** allows us to define **three predicates**: $P_1(x)$ represents x > 20; $P_2(x)$ represents x > 10; $Q(y)$ represents y = 15 for the properties of x, y. Does the following hold using predicate logical for the inference?
  - $\{P_1(x), P_2(x) \rightarrow Q(y)\} \vdash Q(y)$ or $(P_1(x) \wedge P_2(x) \rightarrow Q(y)) \rightarrow Q(y)$

Predicate Logic (Code Example)

- S_1: x > 20;
- S_2: x > 10;
- $S_2 \rightarrow Q$: if(x > 10) y = 15;
- Q: y = 15;
- In **propositional logic**, each statement (including its variables and constants) is viewed as one proposition. Their relations are not further analyzed.
  - Given propositions S_1 and $S_2 \rightarrow Q$ as the knowledge base KB. Does the following semantically entail $\{S_1, S_2 \rightarrow Q\} \vdash Q$ or $(S_1 \wedge S_2 \rightarrow Q) \rightarrow Q$ hold?
  - Answer: No!
- **Predicate logic** allows us to define **three predicates**: $P_1(x)$ represents x > 20; $P_2(x)$ represents x > 10; $Q(y)$ represents y = 15 for the properties of x, y. Does the following hold using predicate logical for the inference?
  - $\{P_1(x), P_2(x) \rightarrow Q(y)\} \vdash Q(y)$ or $(P_1(x) \wedge P_2(x) \rightarrow Q(y)) \rightarrow Q(y)$
  - $\{x > 20, x > 10 \rightarrow y = 15\} \vdash y = 15$
  - Answer: Yes!

Satisfiability Checking for Code Verification

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

- ψ is satisfiable if a program *prog* is correct for all valid inputs.

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

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

Satisfiability Checking for Code Verification

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

- ψ is satisfiable if a program *prog* is correct for all valid inputs.

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

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

$$\exists \mathtt{x}\ \exists \mathtt{y}\ \ \neg(P(\mathtt{x}) \wedge S_{prog}(\mathtt{x},\mathtt{y})) \rightarrow Q(\mathtt{x},\mathtt{y}))$$
$$\Rightarrow \quad \exists \mathtt{x}\ \exists \mathtt{y}\ \ P(\mathtt{x}) \wedge S_{prog}(\mathtt{x},\mathtt{y}) \wedge \neg Q(\mathtt{x},\mathtt{y}) \qquad \text{(simplification)}$$

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

Logic formula simplification: https://en.wikipedia.org/wiki/Logical_equivalence

Satisfiability Checking for Code Verification

Checking whether the logical formula ψ is satisfiable by an SMT solver.

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

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

Satisfiability Checking for Code Verification

Checking whether the logical formula ψ is satisfiable by an SMT solver.

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

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

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

SMT: `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories`

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 ψ, and then check the satisfiablity for each path.
  - $\forall \mathit{path} \in \mathit{prog} \quad \mathit{checking}(\psi(\mathit{path}))$

    $\psi(\mathit{path}_1)$: $\exists \mathtt{x}\ P(\mathtt{x}) \wedge \big((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)\big) \wedge \neg Q(\mathtt{x}, \mathtt{y})$ (`if` branch of `foo`)

    $\psi(\mathit{path}_2)$: $\exists \mathtt{x}\ P(\mathtt{x}) \wedge \big((\mathtt{x} \leq 10) \wedge (\mathtt{y} \equiv 10)\big) \wedge \neg Q(\mathtt{x}, \mathtt{y})$ (`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 ψ, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\psi(path))$
    $\psi(path_1)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`if` branch)
    $\psi(path_2)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} \leq 10) \wedge (\mathtt{y} \equiv 10)\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`else` branch)
  - $\psi(path_2)$: **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 ψ, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\psi(path))$
    $\psi(path_1)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`if` branch)
    $\psi(path_2)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} \leq 10) \wedge (\mathtt{y} \equiv 10)\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`else` branch)
  - $\psi(path_2)$: **has a counterexample** `x = 10`!!
  - **Manual translation** of C statements to logic expressions via Z3 theorem prover APIs (`Z3Mgr.h/cpp`) (Lab-Exercise-2)
  - **Automatic translation** of SVFIR to logic expressions during control-flow reachability analysis (Assignment-2)

Proving each $\psi(path)$ formula

Exhaustively by not finding counterexamples via mathematical proofs.

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

Proving each $\psi(path)$ 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

- 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

Theorem prover tools: `https://en.wikipedia.org/wiki/Theorem_prover`

Automated Theorem Provers

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

- If yes, the solver returns a **model** m, a valuation of $\texttt{x}, \texttt{y}, \texttt{z}$ of `foo` that satisfies ψ (i.e., m makes ψ 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, a cross-platform satisfiability modulo theories (SMT) solver developed by Microsoft (This course).

Z3: `https://github.com/Z3Prover/z3/wiki#background`

Code to Logic Expressions with Z3 Theorem Prover

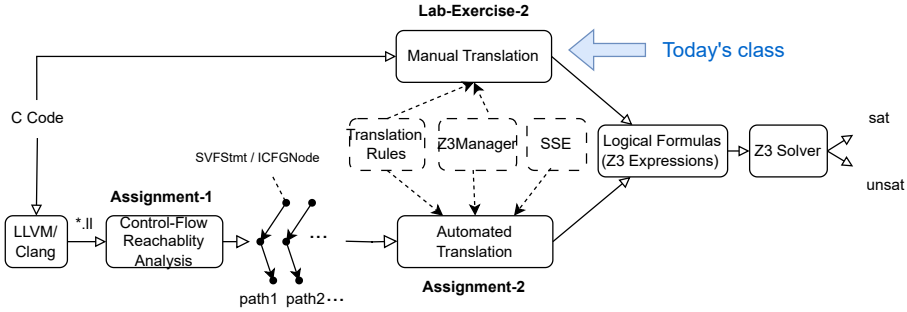
(Week 5)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's class



- We will learn how to manually translate C source code into logical formulas (Z3 constraints/expressions).
- We introduce Z3 solver, Z3 constraint format **Z3 manager** APIs.
- Then, we will demonstrate **examples** for **manual translation** from code to Z3 constraints.

Z3 Theorem Prover

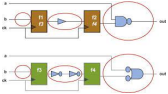
- Z3 is a Satisfiability Modulo Theories (SMT) solver from Microsoft Research[1].
- Targeted at solving problems in software verification and software analysis.
- Main applications are static checking, test case generation, and more ..

Hardware verification

Software analysis/testing

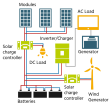
Architecture

Modeling



Geometrical solving

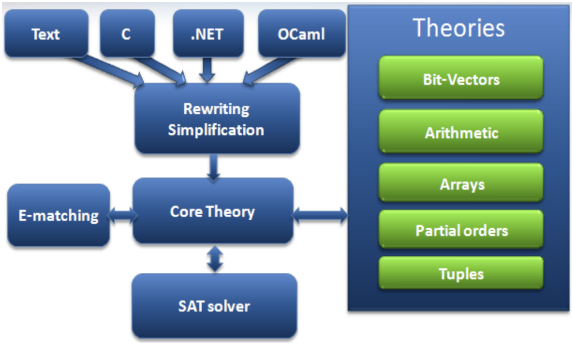
Biological analysis



Hybrid system analysis

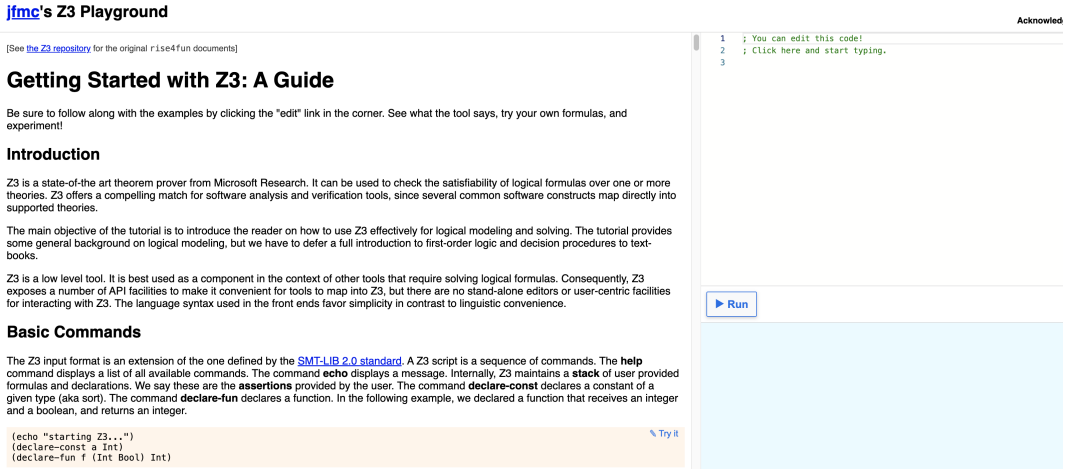
...

Z3 Framework[2]



- Z3 is an effective tool to solve **logical formulas** (Z3 constraints).
- `https://github.com/Z3Prover/z3`.
- Its SMT solver supports theories such as fixed-size bit-vectors, arithmetic, extensional arrays, datatypes, uninterpreted functions, and quantifiers.
- Z3 has official APIs for **C, C++, Python, .NET**, etc.
- **Z3 solver** can find one of the feasible solutions in a set of constraints.

Z3 Playground (https://jfmc.github.io/z3-play)



jfmc's Z3 Playground

Acknowled

[See the Z3 repository for the original rise4fun documents]

Getting Started with Z3: A Guide

Be sure to follow along with the examples by clicking the "edit" link in the corner. See what the tool says, try your own formulas, and experiment!

Introduction

Z3 is a state-of-the art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. Z3 offers a compelling match for software analysis and verification tools, since several common software constructs map directly into supported theories.

The main objective of the tutorial is to introduce the reader on how to use Z3 effectively for logical modeling and solving. The tutorial provides some general background on logical modeling, but we have to defer a full introduction to first-order logic and decision procedures to text-books.

Z3 is a low level tool. It is best used as a component in the context of other tools that require solving logical formulas. Consequently, Z3 exposes a number of API facilities to make it convenient for tools to map into Z3, but there are no stand-alone editors or user-centric facilities for interacting with Z3. The language syntax used in the front ends favor simplicity in contrast to linguistic convenience.

Basic Commands

The Z3 input format is an extension of the one defined by the SMT-LIB 2.0 standard. A Z3 script is a sequence of commands. The **help** command displays a list of all available commands. The command **echo** displays a message. Internally, Z3 maintains a **stack** of user provided formulas and declarations. We say these are the **assertions** provided by the user. The command **declare-const** declares a constant of a given type (aka sort). The command **declare-fun** declares a function. In the following example, we declared a function that receives an integer and a boolean, and returns an integer.

```
(echo "starting Z3...")
(declare-const a Int)
(declare-fun f (Int Bool) Int)
```

✎ Try it

```
1  ; You can edit this code!
2  ; Click here and start typing.
3
```

▶ Run

More Z3 Learning Materials

- Z3 GitHub repository `https://github.com/z3prover/z3`
- Z3 tutorials `https://github.com/philzook58/z3_tutorial`
- Z3 slides `https://github.com/Z3Prover/z3/wiki/Slides`
- Programming Z3 `http://theory.stanford.edu/~nikolaj/programmingz3.html#sec-logical-interface`

Z3 Solver and Z3 Formulas

Z3 solver accepts a first-order (predicate) logical formula ψ, and outputs one of the following results.

- `sat` if ψ is satisfiable
- `unsat` if there is a counterexample which make ψ unsatisfiable
- `unknown` if ψ is too complex and can not be solved within a time frame.

Z3 Solver and Z3 Formulas

Z3 solver accepts a first-order (predicate) logical formula ψ, and outputs one of the following results.

- `sat` if ψ is satisfiable
- `unsat` if there is a counterexample which make ψ unsatisfiable
- `unknown` if ψ is too complex and can not be solved within a time frame.

You play around and check the satisfiability of your Z3 constraints/formulas here:
`https://jfmc.github.io/z3-play` or
`https://compsys-tools.ens-lyon.fr/z3/index.php`

Z3's Logical Formula (Constants, Check-Sat and Evaluation)

The Z3 input format (formula format) is an extension of the SMT-LIB 2.0 standard[3]. A Z3 formula expression (`z3::expr`) has the following keywords:

- **`echo`** displays a message
- **`declare-const`** declares a constant of a given type (a.k.a `sort`)
- **`declare-fun`** declares a function
- **`assert`** adds a formula into the Z3 internal stack
- **`check-sat`** determines whether the current formulas on the Z3 stack are satisfiable or not
- **`get-model`** is used to retrieve an interpretation (one solution) that makes all formulas on the Z3 internal stack true
- **`eval`** evaluates a variable/expression produced by a model when the formulas is satisfiable.

Constants, Check-Sat and Evaluation (Example)

$$\psi : (\mathtt{x} > 10) \ \wedge \ (\mathtt{y} \equiv \mathtt{x} + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

Constants, Check-Sat and Evaluation (Example)

$$\psi : (x > 10) \wedge (y \equiv x + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

```
1 (echo "starting Z3...")
2 (declare-const x Int) /// Declare an Int type variable "x"
3 (declare-const y Int) /// Declare an Int type variable "y"
4 (assert (> x 10))  /// Add the first part (x>10) of the conjunction into the solver
5 (assert (= y (+ x 1))) /// Add the second part (y==x+1) of the conjunction
6 (check-sat) /// Check whether added formulas are satisfiable.
7 (eval x) /// Evaluate the value of x when the formula is satisfiable
8 (eval y) /// Evaluate the value of y when the formula is satisfiable
```

Constants, Check-Sat and Evaluation (Example)

$$\psi : (x > 10) \wedge (y \equiv x + 1)$$

How to represent this formula in Z3 and feed it into Z3's solver?

```
1 (echo "starting Z3...")
2 (declare-const x Int) /// Declare an Int type variable "x"
3 (declare-const y Int) /// Declare an Int type variable "y"
4 (assert (> x 10))  /// Add the first part (x>10) of the conjunction into the solver
5 (assert (= y (+ x 1))) /// Add the second part (y==x+1) of the conjunction
6 (check-sat) /// Check whether added formulas are satisfiable.
7 (eval x) /// Evaluate the value of x when the formula is satisfiable
8 (eval y) /// Evaluate the value of y when the formula is satisfiable
```

Outputs of Z3's solver:

```
1 starting Z3...
2 sat /// (check-sat) result
3 11 /// the value of x as one satisfiable solution
4 12 /// the value of y as one satisfiable solution
```

Z3's Logical Formula (Uninterpreted Function)

The basic building blocks of SMT formulas are constants and uninterpreted functions.

- An uninterpreted function **has no other property** (no priori interpretation) **than its signature** (i.e., function name and arguments).
- An uninterpreted functions in first-order logic have **no side-effects** (e.g., can not change argument values and never return different values for the same input)
- **Constants** in Z3 can also be seen as **functions that take no arguments**.
- **The details and characteristics** of uninterpreted functions are **ignored**. This can **generalize and simplify** theorems and proofs.

Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))   /// f(10) = 1
3 (check-sat)
```

Uninterpreted Function (Example)

```
1 (declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
2 (assert (= (f 10) 1))   /// f(10) = 1
3 (check-sat)
```

Outputs of Z3's solver:

```
1 sat
```

The solver returns `sat`, because `f` is an uninterpreted function (i.e., all that is known about `f` is its signature), so it is possible that `f(10) = 1`.

Uninterpreted Function (Example)

```
(declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
(assert (= (f 10) 1))   /// f(10) = 1
(check-sat)
```

Outputs of Z3's solver:

```
sat
```

The solver returns `sat`, because `f` is an uninterpreted function (i.e., all that is known about `f` is its signature), so it is possible that `f(10) = 1`.

```
(declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
(assert (= (f 10) 1))   /// f(10) = 1
(assert (= (f 10) 2))   /// f(10) = 2
(check-sat)
```

Uninterpreted Function (Example)

```
(declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
(assert (= (f 10) 1))   /// f(10) = 1
(check-sat)
```

Outputs of Z3's solver:

```
sat
```

The solver returns `sat`, because `f` is an uninterpreted function (i.e., all that is known about `f` is its signature), so it is possible that `f(10) = 1`.

```
(declare-fun f (Int) Int)   /// Function f accepts an Int argument and returns a Int
(assert (= (f 10) 1))   /// f(10) = 1
(assert (= (f 10) 2))   /// f(10) = 2
(check-sat)
```

Outputs of Z3's solver:

```
unsat
```

The solver returns `unsat`, because `f`, as an uninterpreted function, can never return different values for the same input.

Uninterpreted Function (Example)

$$\psi : \mathtt{f}(\mathtt{x}) \equiv \mathtt{f}(\mathtt{y}) \ \wedge \ \mathtt{x} ! = \mathtt{y}$$

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (declare-fun f (Int) Int) /// Function f accepts an Int argument and returns a Int
4 (assert (= (f x) (f y)))
5 (assert (not (= x y)))
6 (check-sat)
```

Uninterpreted Function (Example)

$$\psi : \mathtt{f(x)} \equiv \mathtt{f(y)} \ \wedge \ \mathtt{x} ! = \mathtt{y}$$

```
(declare-const x Int)
(declare-const y Int)
(declare-fun f (Int) Int) /// Function f accepts an Int argument and returns a Int
(assert (= (f x) (f y)))
(assert (not (= x y)))
(check-sat)
```

Outputs of Z3's solver:

```
sat
```

An uninterpreted function can have different inputs and return the same output. For example, `f` can always return `1` regardless the value of the input argument.

Constants as Uninterpreted Function (Example)

$$\psi : (x > 10) \wedge (y \equiv x + 1)$$

```
1 (declare-fun x () Int) /// "x" and "y" as an uninterpreted functions
2 (declare-fun y () Int) /// Accepts no argument and return an Int
3 (assert (> x 10))
4 (assert (= y (+ x 1)))
5 (check-sat)
6 (get-model)
```

Outputs of Z3's solver:

```
1 sat
2 (
3   (define-fun x () Int
4     11)                 /// x is evaluated to be 11 for this model
5   (define-fun y () Int
6     12)                 /// y is evaluated to be 11 for this model
7 )
```

`(declare-const x Int)` can be seen as the syntax sugar for `(declare-fun x () Int)`.

Z3's Logical Formula (Arithmetic)

- Z3 supports majority of commonly used arithmetic operators, such as +, -, $*$, /, $<<$, $>>$, $<$, $>$, &, | (The ones listed in SVFIR)
- Types of any two operands should be the same otherwise a type conversion is needed.
- Never mix types in arithmetic, and always be explicit.

```
1 (declare-const a Int)
2 (declare-const b Float32)
3 (assert (= a (+ b 1)))
4 (check-sat)
```

Outputs of Z3's solver:

```
1 Error: (error "line 3 column 19: Sort mismatch at argument #1 for function
2 (declare-fun + (Int Int) Int) supplied sort is (_ FloatingPoint 8 24)")
```

Z3's Logical Formula (`if-then-else` Expression)

- `ite(b, x, y)` represents a conditional expression, where `b` is the condition, `ite` returns `x` if `b` is evaluated true, otherwise `y` is returned
- Used for comparison or branches

```
1 (ite (and (= x!1 11) (= x!2 false)) 21 0)
```

The above Z3 formula evaluates (returns) 21 when x!1 is equal to 11, and x!2 is equal to false. Otherwise, it returns 0.

Z3's Logical Formula (Arrays)

Formulating a program of a mathematical theory of computation McCarthy proposed a basic theory of arrays as characterized by the **select-store** axioms.

- `(select a i)`: returns the value stored at position `i` of the array `a`;
- `(store a i v)`: returns a new array identical to `a`, but on position `i` it contains the value `v`.
- Z3 assumes that arrays are extensional over select. Z3 also enforces that if two arrays agree on all reads, then the arrays are equal.

Z3's Logical Formula (Arrays)

Formulating a program of a mathematical theory of computation McCarthy proposed a basic theory of arrays as characterized by the **select-store** axioms.

- `(select a i)`: returns the value stored at position `i` of the array `a`;
- `(store a i v)`: returns a new array identical to `a`, but on position `i` it contains the value `v`.
- Z3 assumes that arrays are extensional over select. Z3 also enforces that if two arrays agree on all reads, then the arrays are equal.

The following formulas store `y` to the `x`-th position of array `a` and then load the value at `a`'s `x`-th position to `z`

```
1 (declare-const x Int)
2 (declare-const y Bool)
3 (declare-const z Bool)
4 (declare-const a (Array Int Bool))  /// an array of Bools with Int as the indices
5 (assert (= (store a x y) a))    /// a[x] == y
6 (assert (= (select a x) z))     /// z == a[x]
```

Z3's Logical Formula (Scopes)

Z3 maintains a global stack of declarations and assertions via **push** and **pop**

- **push**: creates a new scope by saving the current stack size.
- **pop**: removes any assertion or declaration performed between it and the matching push.

The `check-sat` command always operates on the current global stack.

Z3's Logical Formula (Scopes)

Z3 maintains a global stack of declarations and assertions via **push** and **pop**

- **push**: creates a new scope by saving the current stack size.
- **pop**: removes any assertion or declaration performed between it and the matching push.

The `check-sat` command always operates on the current global stack.

```
1  (declare-const x Int)
2  (declare-const a (Array Int Int))  /// an array of Ints
3  (push)
4  (assert (= (store a 1 10) a))      /// a[1] == 10
5  (assert (= (select a 1) x))       /// x == a[1]
6  (assert (= x 20))                 /// x == 20
7  (check-sat)
8  (pop) ; remove the three assertions
9  (assert (= x 10))                 /// x == 10
10 (check-sat)
```

What is the output of the solver?

Translating Code to Z3 Formulas

We provide a Z3Mgr class (a wrapper class to manipulate Z3 APIs) to generate Z3 formulas or so-called `z3::expr`.

API	Meanings
`z3::expr getZ3Expr(std::string);`	Create a **variable** given a string name
`z3::expr getZ3Expr(int);`	Create a **variable** given an integer
`z3::expr getMemObjAddress(std::string);`	Create a **memory object** in program
`z3::expr getGepObjAddress(z3::expr, u32_t);`	Create a **field object** with an **offset** of an aggregate
`void addToSolver(z3::expr);`	**Add** a Z3 expression/formula to the solver
`void resetSolver();`	Clean all formulas in the the solver
`solver.check();`	**Check satisfiability** of an z3 formula
`z3::expr getEvalExpr(z3::expr);`	**Evaluate an expression** based on a model
`void printExprValues();`	**Print** the values of **all expressions** in the solver.

More details, refer to

`https://github.com/SVF-tools/Teaching-Software-Verification/wiki/SVF-APIs`

Translation Rules

`expr p = getZ3Expr("p")  expr q = getZ3Expr("q")  expr r = getZ3Expr("r")`

SVFStmt	C-Like form	Operations
`AddrStmt` (constant)	`p = c`	`addToSolver(p == c);`
`AddrStmt` (mem allocation)	`p = alloc`	`addToSolver(p == getMemObjAddress("alloc");)`
`CopyStmt`	`p = q`	`addToSolver(p == q);`
`LoadStmt`	$p = *q$	`addToSolver(p == loadValue(q));`
`StoreStmt`	$*p = q$	`storeValue(p, q);`
`GepStmt`	$p = \&(q \rightarrow i)$ or $p = \&q[i]$	`addToSolver(p == getGepObjAddress(q,i));`
`PhiStmt`	$r = \text{phi}(\ell_1 : p,\ \ell_2 : q)$	`if(executed from` l_1`)  addToSolver(p==r);` `if(executed from` l_2`)  addToSolver(q==r);`

`BranchStmt`	`if (p)` l_1 `else` l_2	`expr cond = getEvalExpr(p);`
		`if(cond.is_false())  execute` l_2
		`else  execute` l_1 `addToSolver(cond == true);`

`UnaryOPStmt`	$\neg p$	`addToSolver(!p);`
`BinaryOPStmt`	$r = p \otimes q$	`addToSolver(r == p` $\otimes$ `q);`
`CmpStmt`	$r = p \odot q$	`addToSolver(r == ite(p` $\odot$ `q, true, false));`
`CallPE/RetPE`	$r = f(\ldots, q, \ldots)$ $f(\ldots, p, \ldots)\{\ldots \text{ return } z\}$	
`CallPE`	$p = q$	`solver.push();  addToSolver(p == q);`
`RetPE`	$p = r$	`expr ret = getEvalExpr(r); solver.pop();` `addToSolver(p == ret);`

Translating Code to Z3 Formulas (Scalar Example)

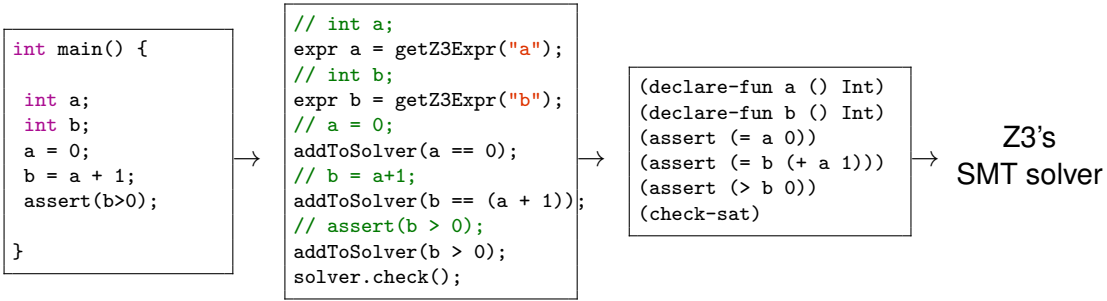
The target program code needs to be in **SSA form** (e.g., SVFIR).

- Top-level variables can only be defined once
  - a = 1; a = 2; $\Longrightarrow$ a1 = 1; a2 = 2;
- Memory objects can only be modified/read through top-level pointers at `StoreStmt` and `LoadStmt`.
  - `p = &a; *p = r;` The value of `a` can only be modified/read via dereferencing `p`.

Translating Code to Z3 Formulas (Scalar Example)

The target program code needs to be in **SSA form** (e.g., SVFIR).

- Top-level variables can only be defined once
  - a = 1; a = 2; $\Longrightarrow$ a1 = 1; a2 = 2;
- Memory objects can only be modified/read through top-level pointers at `StoreStmt` and `LoadStmt`.
  - `p = &a; *p = r;` The value of `a` can only be modified/read via dereferencing `p`.



Translating Code to Z3 Formulas (Memory Operation Example)

- Each memory object has a unique ID and allocated with a **virtual memory address**
- In our modeling, the virtual address starts from **0x7f..... + ID** (i.e., 2130706432 + ID in decimal)
- Memory operations will be through store and load values from `loc2ValMap`, an Z3 array.

```
int main() {
 int* p;
 int x;

 p = malloc(..);
 *p = 5;
 x = *p;
 assert(x==5);
}
```

→

```
//  int** p;
expr p = getZ3Expr("p");
//  int x;
expr x = getZ3Expr("x");
// p = malloc(..);
expr m = getMemObjAddress("malloc1");
addToSolver(p == m);
// *p = 5;
storeValue(p, getZ3Expr(5));
// x = *p;
addToSolver(x == loadValue(p));
// assert(x==5);
addToSolver(x == getZ3Expr(5));
solver.check();
```

→

```
(declare-fun p () Int)
(declare-fun loc2ValMap ()
    (Array Int Int))
(declare-fun x () Int)
(assert (= p 2130706435))
(assert (= x (select
    (store loc2ValMap 2130706435 5)
    2130706435)))
(assert (= x 5))
(check-sat)
```

C code | Translator | Z3 Formulas

What's next?

- (1) Understand Z3 formula format in the slides
- (2) Understand `Z3Mgr` class in the GitHub Repository of Software-Security-Analysis
- (3) Finish the Quiz-2 on WebCMS,
- (4) implement a manual translation from code to Z3 formulas using `Z3Mgr` in `Lab-Exercise-2` for code assertion verification.