Code Verification Using Symbolic Execution

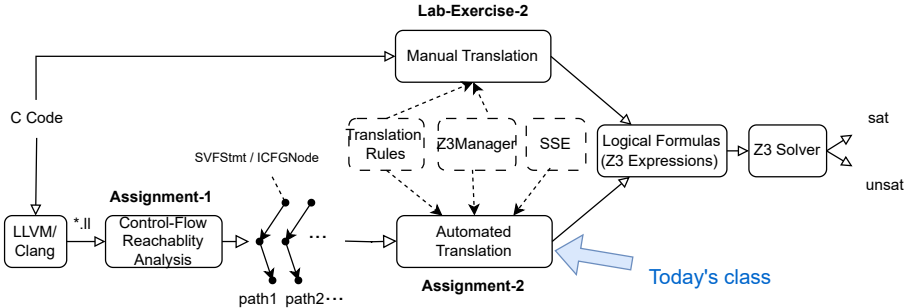
(Week 5)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Code Verification Using Static Symbolic Execution



Today's Roadmap

1. Understand symbolic execution: symbolic values, symbolic state, and branch conditions.
2. Turn assertion checking into a solver query: $BC \wedge State \wedge \neg Q$.
3. Translate ICFG paths into Z3 formulas.
4. Implement Assignment-2 translation rules: branches, calls/returns, integers, memory, and fields.

Key connection

Assignment-1 finds candidate ICFG paths; Assignment-2 translates each path into constraints and asks Z3 whether the path can violate an assertion.

Static Symbolic Execution (SSE)

- Symbolically analyzes a program without runtime execution.
- Use symbolic execution to explore program paths, verify assertions, and find assertion violations.
- A static interpreter follows the program, assuming symbolic values for variables and inputs rather than obtaining actual inputs as normal program execution would.
- International Competition on Software Verification (SV-COMP): `https://sv-comp.sosy-lab.org/`

Basic Concepts in Symbolic Execution

- **Symbolic value**: an unknown input or value represented by a symbol, e.g., $x = \alpha$ instead of $x = 5$.
- **Symbolic state**: a mapping from program variables to symbolic expressions, e.g., $y \mapsto \alpha + 1$.
- **Branch condition (BC)**: constraints that must hold for the current path to be feasible, e.g., $\alpha > 10$.
- **Path formula**: the conjunction of branch constraints and statement constraints collected along one path.
- **Assertion checking**: to find a bug, check whether $BC \wedge State \wedge \neg Q$ is satisfiable.

Key idea

Symbolic execution does not run the program with one concrete input; it reasons about many possible inputs using logical formulas.

Concrete vs Symbolic Execution

Example program

```
1 int foo(int x) {
2     int y = x + 1;
3     assert(y > x);
4 }
```

Concrete execution

Input	State	Assertion
$x = 5$	$y = 6$	$6 > 5$
$x = 10$	$y = 11$	$11 > 10$

Concrete testing checks one input at a time.

Symbolic execution

$x : \alpha$

$y : \alpha + 1$

Check counterexample:

$$\alpha + 1 \leq \alpha$$

unsat

One formula reasons about all possible inputs.

Symbolic State and Branch Condition

```
1 if (x > 10)
2     y = x + 1;
3 else
4     y = 10;
5 assert(y >= x + 1);
```

- **Symbolic state**: maps each variable to its symbolic expression.
- **Path condition**: constraints that must hold to reach the current path.
- Assertion checking on one path asks whether:

$$BC \wedge State \wedge \neg Q$$

is satisfiable.

Path	Path condition	Symbolic state
if branch	$x > 10$	$y \equiv x + 1$
else branch	$x \leq 10$	$y \equiv 10$

From Symbolic Execution to Assertion Checking

After symbolic execution collects constraints for one path, assertion checking asks whether the assertion can be violated:

$$BC \wedge State \wedge \neg Q$$

- **sat**: there exists an input that reaches this path and violates Q.
- **unsat**: no counterexample exists on this path.

Assignment-2 view

For every enumerated ICFG path, translate the path into a Z3 formula and ask whether the negated assertion is feasible.

SSE for Assertion-based Verification

- Given a Hoare triple $P\ \{prog\}\ Q$,
  - P represents pre-condition,
  - *prog* is the program,
  - Q is the post-condition i.e., assertion(s) specifications.

```
// no-precondition
assume(true);        // P
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);  //Q
```

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

SSE for Assertion-based Verification

- Given a Hoare triple $P\ \{prog\}\ Q$,
  - P represents pre-condition,
  - *prog* is the program,
  - Q is the post-condition i.e., assertion(s) specifications.

```
// no-precondition
assume(true);        // P
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);  //Q
```

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

Check **unsat** of ψ (non-existence of counterexamples)!

How to Read Z3 Results

`getSolver().check()` on $BC \wedge State \wedge \neg Q$

Z3 result	Meaning in Assignment-2
unsat	No counterexample exists on this path; the assertion holds on this path.
sat	A counterexample exists; use `getSolver().get_model()` to read concrete values.
unknown	Z3 cannot decide; this should normally not occur in the simplified assignment setting.

Example counterexample query

$$x \leq 10 \wedge y \equiv 10 \wedge \neg(y \geq x + 1)$$

Z3 can return

sat, $\{x = 10, y = 10\}$

The assertion (`y >= x + 1`) fails because $10 \not\geq 11$.

Z3 Expressions, Constraints, and Solver State

- **Symbolic term**: z3 expression, e.g., x, $x + 1$, or $ite(x > 0, 1, 0)$.
- **Constraint**: a Boolean formula (also can be z3 expression) added to the solver, e.g., $y \equiv x + 1$.
- **Solver state**: the conjunction of all constraints added so far.

```
1 z3::expr x = getZ3Expr(xId);
2 z3::expr y = getZ3Expr(yId);
3
4 addToSolver(y == x + 1);  // adds a constraint
5 addToSolver(!(y > x));    // asks for a counterexample
```

$$\text{solver state} = (y \equiv x + 1) \wedge \neg(y > x)$$

Assignment-2 view

Every handler retrieves the required Z3 expressions and adds them to the solver.

How One Path Becomes One Solver Query

For each source-to-assertion path:

1. Start with an empty solver state for the current path.
2. Traverse the path edge by edge.
3. For each branch edge, add the branch condition selected by that edge.
4. For each SVF statement, add its translation constraint.
5. At the assertion, add the negated assertion $\neg Q$.
6. Call `getSolver().check()`.

$$\psi_{path} = \mathit{BranchConstraints} \wedge \mathit{StmtConstraints} \wedge \neg Q$$

Result: `sat` gives a counterexample; `unsat` proves this path safe.

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.
- SSE translates `SVFStmts` of each ICFGNode (except the last one) on each *path* into Z3 expressions and validates whether the path conforms to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - Checking **unsat** of each ψ_{path}. A **sat** of ψ_{path} indicates that there exists at least one counterexample from the **model** from the Z3 solver.

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.
- SSE translates `SVFStmts` of each ICFGNode (except the last one) on each *path* into Z3 expressions and validates whether the path conforms to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - Checking **unsat** of each ψ_{path}. A **sat** of ψ_{path} indicates that there exists at least one counterexample from the **model** from the Z3 solver.

```
void main(int x){
  assume(true);
    if(x > 10)
        y = x + 1;
    else
        y = 10;
  assert(y >= x + 1);
}
```

ψ_{path_1}: $\exists x\ true \wedge \big((x > 10) \wedge (y \equiv x + 1)\big) \wedge \neg(y \geq x + 1)$ (`if` branch)
unsat (no counterexample found!)

ψ_{path_2}: $\exists x\ true \wedge \big((x \leq 10) \wedge (y \equiv 10)\big) \wedge \neg(y \geq x + 1)$ (`else` branch)
sat (a counterexample $x = 10$ found!)

Closed-World Programs and Assertion Checking

- In a **closed-world** program, no inputs or externally supplied values are fixed by assignments or constants.
- We still check assertions by asking whether the negated assertion is feasible:

$$\psi(N_1) \wedge \psi(N_2) \wedge \cdots \wedge \psi(N_i) \wedge \neg\psi(Q)$$

- Interpretation:
  - **unsat**: no feasible counterexample on this path.
  - **sat**: a logical error exists on this path, and the Z3 model gives a witness.

```
void main(){
  int x = 5;
  if (x > 10)
    y = x + 1;
  else
    y = 10;
  assert(y >= x + 1);
}
```

Path	Counterexample query	Result
if branch	$x \equiv 5 \wedge x > 10 \wedge y \equiv x + 1 \wedge y < x + 1$	unsat
else branch	$x \equiv 5 \wedge x \leq 10 \wedge y \equiv 10 \wedge y < x + 1$	unsat

Both paths have no counterexample. The if branch is infeasible; the else branch is feasible and satisfies the assertion.

Reachable Path vs. Feasible Path

```
1 void main() {
2     int x = 5;
3     if (x > 10)
4         assert(false);
5 }
```

- **Reachability analysis** finds syntactic ICFG paths from source to sink.
- **Symbolic execution** checks whether each path is feasible given branch conditions.
- A CFG path may exist even when no concrete input can execute it.

CFG path	entry $\rightarrow$ if.then $\rightarrow$ assertion
Path formula	$x \equiv 5 \wedge x > 10$
Solver result	**unsat**: path is infeasible

Assignment-1 enumerates candidate paths; Assignment-2 uses Z3 to filter infeasible paths and validate assertions.

Finding/Translating Reachability Paths (Recall Assignment-1)

Goal: Find all source-to-sink ICFG paths under matched call/return semantics.

Idea: Use DFS with a bounded call stack to distinguish calling contexts.

1. Represent each DFS state as: $\langle \text{current edge}, \text{call stack} \rangle$
2. Start DFS from an ICFGEdge. // ICFGNode changed to ICFGEdge in Ass-2.
3. If the current state is already on the DFS path, stop exploring to avoid cycles.
4. Add the current edge/node to the current path.
5. If reaching the sink, translating current path via `collectAndTranslatePath`.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call:** push callsite to stack and visit callee (can explore recursion at most once).
   - **Return:**
     - **Non-empty stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

General Translation Pattern

Most SVF statement translations follow the same idea:

program statement $\Longrightarrow$ Z3 constraint describing the statement

Program-level meaning	Z3 constraint
$x = y$	$expr(x) \equiv expr(y)$
$x = y + z$	$expr(x) \equiv expr(y) + expr(z)$
$b = (x > y)$	$expr(b) \equiv ite(expr(x) > expr(y), 1, 0)$
$x = *p$	$expr(x) \equiv load(expr(p))$
$*p = x$	$store(expr(p), expr(x))$
...	...

Key point

The translation does not execute the program. It builds constraints that describe one path.

From SVF Values to Z3 Expressions

- Each SVF variable (`ValVar` and `ObjVar`) is represented by a `NodeID`.
- `getZ3Expr(NodeID)` creates or retrieves the Z3 expression for a top-level variable (`ValVar`) and `getMemObjAddress(NodeID)` retrieves the z3 expression of a memory object (`ObjVar`).
- Think of `getZ3Expr(x)` as the symbolic value of `x`, not as a concrete runtime value.

SVF statement	Informal meaning	Formula added to Z3
`p = q`	copy value	$\texttt{getZ3Expr(p)} \equiv \texttt{getZ3Expr(q)}$
`r = p + q`	arithmetic operation	$\texttt{getZ3Expr(r)} \equiv \texttt{getZ3Expr(p)} + \texttt{getZ3Expr(q)}$
`b = (p == q)`	comparison result	$\texttt{getZ3Expr(b)} \equiv \mathit{ite}(\texttt{getZ3Expr(p)} \equiv \texttt{getZ3Expr(q)}, 1, 0)$

Implementation rule

Retrieve the correct Z3 expressions first, then add the corresponding equality or constraint to the solver.

Rhs/Lhs/Operators of a Statement

SVFStmt	C-Like	PAG Edge	Lhs/Rhs/Operator
AddrStmt	`p = &o`	$p \xleftarrow{Addr} o$	`p`: lhs var, `o`: rhs var
CopyStmt	`p = q`	$p \xleftarrow{Copy} q$	`p`: lhs var, `q`: rhs var
StoreStmt	`*p = q`	$p \xleftarrow{Store} q$	`p`: lhs var, `q`: rhs var
LoadStmt	`p = *q`	$p \xleftarrow{Load} q$	`p`: lhs var, `q`: rhs var
GepStmt	`p = &q->fld`	$p \xleftarrow{Gep,fld} q$	`p`: lhs var, `q`: rhs var
CallPE	`p = q` (caller arg to callee arg)	$p \xleftarrow{Copy} q$	`p`: lhs var (callee), `q`: rhs var (caller)
RetPE	`p = q` (callee ret to caller ret var)	$p \xleftarrow{Copy} q$	`p`: lhs var (caller), `q`: rhs var (callee)
CmpStmt	`r = op(p, q)`	`r`: result, `p`: operand 0, `q`: operand 1	
BinaryOPStmt	`r = op(p, q)`	`r`: result, `p`: operand 0, `q`: operand 1	
SelectStmt	`r = sel(c,tval,fval)`	`r`: result, condition `c`: 0 or 1 `r` is `tval` when c≡1, `r` is `fval` when c≡0	

`lhs` and `rhs` When Handling Calls/Returns

Let us see the example of `lhs` and `rhs` variables for call and parameter passing (i.e., `CallPE` and `RetPE`)

```
1 int foo(int x){ // CallPE: x = m;  lhs: x, rhs: m
2     int y = x;
3     return y;
4 }
5
6 main(){
7     int m = 0;
8     n = foo(m); // RetPE: n = y;   lhs: n, rhs : y
9 }
```

Parameter passing is from actual parameter `m` at the callsite (Line 8) to formal parameter `x` at the entry of `foo`. Return value passing is from return variable `y` in `foo` to `n` at the callsite (Line 8).

Why Context-Sensitive Expressions Matter (More in Next Lecture)

```
1 int id(int x) {
2     return x;
3 }
4
5 void main() {
6     int a = id(1);  // call1
7     int b = id(2);  // call2
8     assert(a != b);
9 }
```

- The same formal parameter `x` in `id` is used by two different calls.
- Without calling context, formulas from different invocations may be mixed.
- With context sensitivity, each expression is identified by:

$$\langle callingCtx, NodeID \rangle$$

Call context	Formula
$[call_1]$	$\langle [call_1], x \rangle \equiv 1$
$[call_2]$	$\langle [call_2], x \rangle \equiv 2$

Overview of SSE Algorithms: (Assignment-2 starter code)

Algorithm 1: `translatePath(path)`

```
1 foreach edge ∈ path do
2     if intra ← dyn_cast⟨Intra⟩(edge) then
3         if handleIntra(intra) == false then
4             return false
5     else if call ← dyn_cast⟨CallEdge⟩(edge) then
6         handleCall(call)
7     else if ret ← dyn_cast⟨RetEdge⟩(edge) then
8         handleRet(ret)
9     return true
```

Algorithm 2: `handleIntra(intraEdge)`

```
1 if intraEdge.getCondition() then
2     if !handleBranch(intraEdge) then
3         return false;
4     else
5         return handleNonBranch(intraEdge);
6 else
7     return handleNonBranch(intraEdge);
```

Algorithm 3: `handleCall(callEdge)`

```
1 // Your code starts here ..;
2
3 // For each CallPE lhs = rhs:
4 // (1) Retrieve two Z3 expressions ⟨[c], rhs⟩ and
  ⟨[c′], lhs⟩ under contexts c (caller's context) and
  c' (callee's context); Note that pushCallingCtx
  will make c become c'.
5 // (2) Add formula ⟨[c], rhs⟩ ≡ ⟨[c′], lhs⟩ into the
  solver.
```

Algorithm 4: `handleRet(retEdge)`

```
1 // Your code starts here ..;
2
3 // For each RetPE lhs = rhs:
4 // (1) Retrieve two Z3 expressions ⟨[c′], rhs⟩ and
  ⟨[c], lhs⟩ under contexts c (caller's context) and
  c' (callee's context); Note that popCallingCtx
  will make c' become c.
5 // (2) Add formula ⟨[c′], rhs⟩ ≡ ⟨[c], lhs⟩ into the
  solver.
```

Handle Intra-procedural CFG Edges (`handleIntra`)

Algorithm 5: `handleIntra(intraEdge)`

```
1  if intraEdge.getCondition() then
2      if !handleBranch(intraEdge) then
3          return false;
4      else
5          return handleNonBranch(intraEdge);
6  else
7      return handleNonBranch(intraEdge);
```

Algorithm 6: `handleBranch(intraEdge)`

```
1  // Your code starts here ..;
2  // Retrieve branch condition (0 or 1);
3  cond = ...;
4  // Retrieve the condition on the current CFG edge
   (0 or 1);
5  succ = ...;
6  // Evaluate path feasibility only when the two
   conditions are equal; add (cond==succ) to the
   solver only if it holds.  Return false if the path
   is infeasible.;
```

Algorithm 7: `HandleNonBranch(intraEdge)`

```
1   dst ← intraEdge.getDstNode();
    src ← intraEdge.getSrcNode();
2   foreach stmt ∈ dst.getSVFStmts() do
3       if addr ← dyn_cast⟨AddrStmt⟩(stmt) then
4           // handle AddrStmt
5       else if copy ← dyn_cast⟨CopyStmt⟩(stmt) then
6           // handle CopyStmt
7       else if load ← dyn_cast⟨LoadStmt⟩(stmt) then
8           // handle LoadStmt
9       else if store ← dyn_cast⟨StoreStmt⟩(stmt) then
10          // handle StoreStmt
11      else if gep ← dyn_cast⟨GepStmt⟩(stmt) then
12          // handle GepStmt
13      else if binary ← dyn_cast⟨BinaryStmt⟩(stmt) then
14          // handle BinaryStmt
15      else if cmp ← dyn_cast⟨CmpStmt⟩(stmt) then
16          // handle CmpStmt
17      else if phi ← dyn_cast⟨PhiStmt⟩(stmt) then
18          // handle PhiStmt
19      else if select ← dyn_cast⟨SelectStmt⟩(stmt) then
20          // handle SelectStmt
21      ...
```

Example 1: CmpStmt and BinaryOPStmt

```
1 void main(int x) {
2  int y, z, b;
3  y = x;
4  // C-like CmpStmt
5  b = (x == y);
6  // C-like BinaryOPStmt
7  z = x + y;
8  assert(z == 2 * x)
9 }
```

Example 1: CmpStmt and BinaryOPStmt

```
1 void main(int x) {
2  int y, z, b;
3  y = x;
4  // C-like CmpStmt
5  b = (x == y);
6  // C-like BinaryOPStmt
7  z = x + y;
8  assert(z == 2 * x)
9 }
```

Concrete Execution
(Concrete states)

One execution:

x :	5
y :	5
b :	1
z :	10

Another execution:

x :	10
y :	10
b :	1
z :	20

Example 1: CmpStmt and BinaryOPStmt

```
1 void main(int x) {
2   int y, z, b;
3   y = x;
4   // C-like CmpStmt
5   b = (x == y);
6   // C-like BinaryOPStmt
7   z = x + y;
8   assert(z == 2 * x)
9 }
```

Concrete Execution
(Concrete states)

One execution:

x :	5
y :	5
b :	1
z :	10

Another execution:

x :	10
y :	10
b :	1
z :	20

Symbolic Execution
(`getZ3Expr(x)` **represents** `x`'s **symbolic state**)

```
x : getZ3Expr(x)
y : getZ3Expr(y)
b : ite(getZ3Expr(x)≡getZ3Expr(y), 1, 0)
z : getZ3Expr(x) + getZ3Expr(y)
```

Checking satisfiability using "`getSolver().check()`".

Checking non-existence of counterexamples: $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$\mathtt{y} \equiv \mathtt{x} \wedge \mathtt{b} \equiv \mathtt{ite}(\mathtt{x} \equiv \mathtt{y}, 1, 0) \wedge \mathtt{z} \equiv \mathtt{x} + \mathtt{y} \wedge \mathtt{z} \neq 2 * \mathtt{x}$	unsat

Example 1: CmpStmt and BinaryOPStmt

```
1 void main(int x) {
2   int y, z, b;
3   y = x;
4   // C-like CmpStmt
5   b = (x == y);
6   // C-like BinaryOPStmt
7   z = x + y;
8   assert(z == 2 * x)
9 }
```

Concrete Execution
(Concrete states)

One execution:

x :	5
y :	5
b :	1
z :	10

Another execution:

x :	10
y :	10
b :	1
z :	20

Symbolic Execution
(`getZ3Expr(x)` **represents** `x`'s **symbolic state**)

```
x : getZ3Expr(x)
y : getZ3Expr(y)
b : ite(getZ3Expr(x)≡getZ3Expr(y), 1, 0)
z : getZ3Expr(x) + getZ3Expr(y)
```

In `Assignment-2`, we **only handle signed integers** including both positive and negative numbers and assume that the program is **integer-overflow-free** in this assignment.

Handling CmpStmt

Algorithm 8: Handle CmpStmt

```
1  op0 ← get z3 expression of operand 0;
2  op1 ← get z3 expression of operand 1;
3  res ← get z3 expression of result;
4  // refer to SVF's CmpStmt::Predicate;
5  switch predicate of cmp do
6      case CmpStmt :: ICMP_EQ do
7          // handle equal
           Use the ite (if − then − else) API to return 1
8            if operands are equal, 0 otherwise
9            then add to z3 solver
10     case CmpStmt :: ICMP_NE do
11         // handle not equal;
12     case CmpStmt :: ICMP_UGT do
13         Use the ite (if − then − else) API to return 1
14         // handle unsigned greater than;
15     case CmpStmt :: ICMP_SGT do
16         // handle signed greater than;
17     ...
```

Algorithm 9: Handling CmpStmt

```
1  case CmpStmt :: ICMP_UGE do
2      // handle unsigned greater equal;
3  case CmpStmt :: ICMP_SGE do
4      // handle signed greater equal;
5  case CmpStmt :: ICMP_ULT do
6      // handle unsigned less than;
7  case CmpStmt :: ICMP_SLT do
8      // handle signed less than;
9  case CmpStmt :: ICMP_ULE do
10     // handle unsigned less equal;
11 case CmpStmt :: ICMP_SLE do
12     // handle signed less equal;
```

Handle BinaryOPStmt

Algorithm 9: Handle BinaryOpStmt

```
1  op0 ← getZ3Expr(binary.getOpVarID(0));
2  op1 ← getZ3Expr(binary.getOpVarID(1));
3  res ← getZ3Expr(binary.getResID());
4  switch binary.getOpcode() do
5      case BinaryOperator :: Add do
6          addToSolver(res == op0 + op1);
7      case BinaryOperator :: Sub do
8          addToSolver(res == op0 − op1);
9      case BinaryOperator :: Mul do
10         addToSolver(res == op0 × op1);
11     case BinaryOperator :: SDiv do
12         addToSolver(res == op0/op1);
13     case BinaryOperator :: SRem do
14         addToSolver(res == op0%op1);
15     case BinaryOperator :: Xor do
16         addToSolver(res ==
17           bv2int(int2bv(32, op0) ⊕ int2bv(32, op1), 1)));
18     case BinaryOperator :: And do
19         addToSolver(res ==
20           bv2int(int2bv(32, op0)&int2bv(32, op1), 1)));
21     ...
```

Algorithm 9: Handle BinaryOpStmt

```
1  case BinaryOperator :: Or do
2      addToSolver(res ==
3        bv2int(int2bv(32, op0)|int2bv(32, op1), 1)));
4  case BinaryOperator :: AShr do
5      addToSolver(res ==
6        bv2int(ashr(int2bv(32, op0), int2bv(32, op1)), 1)));
7  case BinaryOperator :: Shl do
8      addToSolver(res ==
9        bv2int(shl(int2bv(32, op0), int2bv(32, op1)), 1)));
```

A Simple Symbolic Memory Model

```
1 p = malloc(...);  // memory object o
2 *p = x + 5;
3 y = *p;
```

- `malloc` creates a memory object o with a virtual address $addr_o$.
- `StoreStmt` updates the symbolic value stored at an address.
- `LoadStmt` reads the symbolic value from an address.

Concept	Assignment-2 abstraction
Object address	`getMemObjAddress(ObjVarID)`
Store	`z3Mgr.storeValue(addr, val)`
Load	`z3Mgr.loadValue(addr)`

So the code above becomes:

$$p \equiv addr_o, \quad store(addr_o, x + 5), \quad y \equiv load(addr_o)$$

Example 2: Memory Operation

```
1 void main(int x) {
2  int* p;
3  int y;
4
5  p = malloc(..);
6  *p = x + 5;
7  y = *p;
8  assert(y==x+5);
9 }
```

Example 2: Memory Operation

```
1 void main(int x) {
2   int* p;
3   int y;
4
5   p = malloc(..);
6   *p = x + 5;
7   y = *p;
8   assert(y==x+5);
9 }
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
0x1234	:	15
y	:	15

Another execution:

x	:	0
p	:	0x1234
0x1234	:	5
y	:	5

Example 2: Memory Operation

```
1 void main(int x) {
2  int* p;
3  int y;
4
5  p = malloc(..);
6  *p = x + 5;
7  y = *p;
8  assert(y==x+5);
9 }
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
0x1234	:	15
y	:	15

Another execution:

x	:	0
p	:	0x1234
0x1234	:	5
y	:	5

Symbolic Execution
(Symbolic states)

x	:	`getZ3Expr(x)`
p	:	0x7f000001
		`virtual address from getMemObjAddress(ObjVarID)`
0x7f000001	:	`getZ3Expr(x)` + 5
y	:	`getZ3Expr(x)` + 5

Checking non-existence of counterexamples:

$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$\mathtt{p} \equiv \mathtt{0x7f000001} \wedge \mathtt{y} \equiv \mathtt{x} + 5 \wedge \mathtt{y} \neq \mathtt{x} + 5$	unsat

Handling Memory Operation

Algorithm 10: Handle AddrStmt

```
1 obj ← get memory address of rhsvar;
2 lhs ← get z3 expression of lhsvar;
3 add obj == lhs to z3 solver;
```

Algorithm 11: Handle LoadStmt

```
1 lhs ← get z3 expression of lhsvar;
2 rhs ← get z3 expression of rhsvar;
3 add lhs == z3Mgr.loadValue(rhs) to z3 solver;
```

Algorithm 12: Handle StoreStmt

```
1 lhs ← get z3 expression of lhsvar;
2 rhs ← get z3 expression of rhsvar;
3 store rhs to lhs;
```

Example 3: Field Access for Struct and Array

```
1  struct st{
2      int a;
3      int b;
4  }
5  void main(int x) {
6   struct st* p = malloc(..);
7   q = &(p->b);
8   *q = x;
9   int k = p->b;
10  assert(k == x);
11 }
```

Example 3: Field Access for Struct and Array

```
1  struct st{
2      int a;
3      int b;
4  }
5  void main(int x) {
6   struct st* p = malloc(..);
7   q = &(p->b);
8   *q = x;
9   int k = p->b;
10  assert(k == x);
11 }
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	10
k	:	10

Another execution:

x	:	20
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	20
k	:	20

Example 3: Field Access for Struct and Array

```
1  struct st{
2      int a;
3      int b;
4  }
5  void main(int x) {
6   struct st* p = malloc(..);
7   q = &(p->b);
8   *q = x;
9   int k = p->b;
10  assert(k == x);
11 }
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	10
k	:	10

Another execution:

x	:	20
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	20
k	:	20

Symbolic Execution
(Symbolic states)

x	:	getZ3Expr(x)
p	:	0x7f000001
		virtual address from
		getMemObjAddress(ObjVarID)
&(p→b)	:	0x7f000002
q	:	0x7f000002
		field virtual address from
		getGepObjAddress(base, offset)
0x7f000002	:	getZ3Expr(x)
k	:	getZ3Expr(x)

The virtual address for modeling a field is based on the index of the field offset from the base pointer of a struct (nested struct will be flattened to allow each field to have a unique index)

Example 3: Field Access for Struct and Array

```
1  struct st{
2      int a;
3      int b;
4  }
5  void main(int x) {
6   struct st* p = malloc(..);
7   q = &(p->b);
8   *q = x;
9   int k = p->b;
10  assert(k == x);
11 }
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	10
k	:	10

Another execution:

x	:	20
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	20
k	:	20

Symbolic Execution
(Symbolic states)

x	:	getZ3Expr(x)
p	:	0x7f000001
		virtual address from
		getMemObjAddress(ObjVarID)
&(p→b)	:	0x7f000002
q	:	0x7f000002
		field virtual address from
		getGepObjAddress(base, offset)
0x7f000002	:	getZ3Expr(x)
k	:	getZ3Expr(x)

Checking non-existence of counterexamples:

$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$\mathtt{p} \equiv \mathtt{0x7f000001} \wedge \mathtt{q} \equiv \mathtt{0x7f000002} \wedge \mathtt{k} \equiv \mathtt{x} \wedge \mathtt{k} \neq \mathtt{x}$	unsat

Handling Field and Array Access (GepStmt)

Algorithm 12: Handle GepStmt

```
1 lhs ← get z3 expression of lhs var;
2 rhs ← get z3 expression of rhs var;
3 offset ← get gep offset under curCallCtx;
4 gepAddress ← get gep address of rhs given offset;
5 add lhs == gepAddress to z3 solver;
```

Method `getGepObjAddress` supports both `struct` and `array` accesses using a base pointer and element index.

In `Assignment-2`, **we do not consider object byte sizes or low-level incompatible type casting** in `Assignment-2`.

```
z3::expr Z3SSEMgr::getGepObjAddress(z3::expr pointer, u32_t offset) {
    NodeID obj = getInternalID(z3Expr2NumValue(pointer));
    // Find the baseObj and return the field object.
    // The indices of sub-elements of a nested aggregate object has been flattened
    NodeID gepObj = svfir->getGepObjVar(obj, offset);
    if (obj == gepObj)
        return getZ3Expr(obj);
    else
        return createExprForObjVar(SVFUtil::cast<GepObjVar>(svfir->getGNode(gepObj)));
}
```

Branch Translation Rule

```
1 b = (x > 10);
2 if (b)
3     y = x + 1;
4 else
5     y = 10;
```

- A branch condition is represented as an integer Boolean: `1` for true, `0` for false.
- `handleBranch(intraEdge)` compares the symbolic branch condition with the current CFG edge condition.

CFG edge	Condition on edge	Constraint added
true edge	`succ == 1`	$b \equiv 1$
false edge	`succ == 0`	$b \equiv 0$

If the solver cannot satisfy the accumulated constraints, the current path is infeasible.

Example 4: Branches

```
1 void main(int x){
2     if(x > 10) {
3         y = x + 1;
4     }
5     else {
6         y = 10;
7     }
8 assert(y >= x + 1);
9 }
```

Example 4: Branches

```
1 void main(int x){
2     if(x > 10) {
3         y = x + 1;
4     }
5     else {
6         y = 10;
7     }
8 assert(y >= x + 1);
9 }
```

Concrete Execution
(concrete states)

One execution:
x : 20
y : 21

Another execution:
x : 8
y : 10

Example 4: Branches

```
1 void main(int x){
2     if(x > 10) {
3         y = x + 1;
4     }
5     else {
6         y = 10;
7     }
8 assert(y >= x + 1);
9 }
```

Concrete Execution (concrete states)	Symbolic Execution (Symbolic states)
One execution:	If branch:
x : 20	x : getZ3Expr(x) $>$ 10
y : 21	y : getZ3Expr(x) + 1
Another execution:	Else branch:
x : 8	x : getZ3Expr(x) $\leq$ 10
y : 10	y : 10

Checking non-existence of counterexamples:

Path	$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability	Counterexample
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_8$ (if.then branch)	$x > 10 \wedge y \equiv x + 1 \wedge y < x + 1$	unsat	$\varnothing$
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_6 \rightarrow \ell_8$ (if.else branch)	$x \leq 10 \wedge y \equiv 10 \wedge y < x + 1$	sat	$\{x : 10, y : 10\}$

Getting the potential counterexample via "`getSolver().get_model()`" after "`getSolver().check()`".

Scope of Assignment-2

- Verify assertion violations on each enumerated ICFG path.
- Context-sensitive reachability analysis with the callstack size (e.g., size of 3) applied only to recursive call contexts.
- Handle signed integers and assume the program is integer-overflow-free.
- Represent branch conditions as integer Booleans: 1 for true and 0 for false.
- Model memory using virtual object addresses and symbolic load/store.
- Handle context sensitivity by pairing variables with the current calling context.

Goal

Implement a clear path-to-formula translation for the assignment setting.

What's next?

1. Understand SSE algorithms and examples in the slides.
2. Finish Quiz-2 and Lab-Exercise-2 on WebCMS.
3. Start implementing the automated translation from code to Z3 formulas using `SSE` and `Z3SSEMgr` in Assignment 2.