

Code Verification Using Symbolic Execution

(Week 5)

Yulei Sui

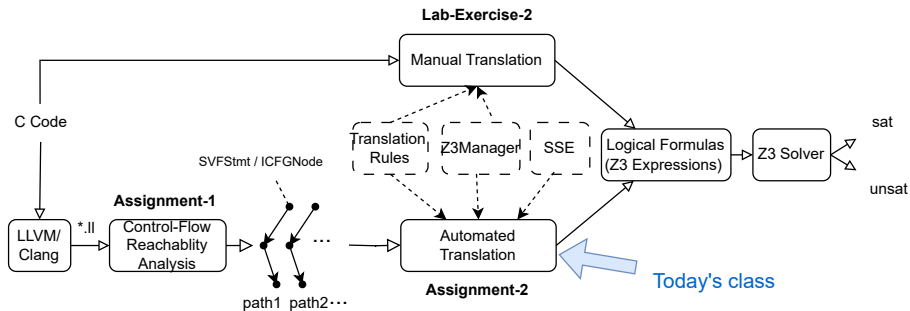
School of Computer Science and Engineering

University of New South Wales, Australia

Revisit Lab-Exercise-2 Cases

- Lab-Exercise-2 Validation Code
 - `test0()` has an example for validating your translation. Given a program `prog` and an assert `Q`, you are expected to (1) use `checkNegateAssert` to translate the negation of `Q` and check **unsat** of $prog \wedge \neg Q$; `checkNegateAssert` returns true means the non-existence of counterexamples, and (2) also **evaluate individual variables' values** (e.g., `a`) if you know `a`'s value is 3. For example, `z3Mgr->getEvalExpr(getZ3Expr("a")) == getZ3Expr(3)`.
 - Closed-world programs, checking **sat** of $prog \wedge Q \equiv$ checking **unsat** $prog \wedge \neg Q$
- `addToSolver(e1)` vs `getEvalExpr(e2)`
 - `e1` is added as a constraint to the solver, while `e2` is not added to the solver hence its truth depends on a particular model (one solution).
- Memory allocations: `p = &a`;
 - `a` is address-taken by `p`, hence an object `&a` needs to be created via `a_addr = getMemObjAddress("&a");`
- Interprocedural (call and return)
 - Bookkeeping the calling context to distinguish local variables.

Code Verification Using Static Symbolic Execution



Static Symbolic Execution (SSE)

- Automated analysis and testing technique that symbolically analyzes a program without runtime execution.
- Use symbolic execution to explore all program paths to find bugs and assertion validations.
- 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/>

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

$$\Longrightarrow \frac{\exists x \exists y \ P \wedge S_{prog}(x, y)) \wedge \neg Q(x, y)}{\text{logical formula } \psi}$$

feed into

\Longrightarrow 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

\Longrightarrow

$\frac{\exists x \exists y \ P \wedge S_{prog}(x, y) \wedge \neg Q(x, y)}{\text{logical formula } \psi}$

logical formula ψ

feed into

\Longrightarrow

SMT
Solver

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

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \dots, N_i, Q]$, 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 $path = [N_1, N_2, \dots, N_i, Q]$, 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 validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
 - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \dots \wedge \psi(N_i) \wedge \neg\psi(Q)$
 - 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 $path = [N_1, N_2, \dots, N_i, Q]$, 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 validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
 - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \dots \wedge \psi(N_i) \wedge \neg\psi(Q)$
 - 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)            $\psi_{path_1} : \exists x \text{ true} \wedge ((x > 10) \wedge (y \equiv x + 1)) \wedge \neg(y \geq x + 1)$  (if branch)
        y = x + 1;      unsat (no counterexample found!)
    else
        y = 10;           $\psi_{path_2} : \exists x \text{ true} \wedge ((x \leq 10) \wedge (y \equiv 10)) \wedge \neg(y \geq x + 1)$  (else branch)
    assert(y >= x + 1);  sat (a counterexample  $x = 10$  found!)
}
```

Closed-World Programs and Assertion Checking

- If the program operates in a **closed-world** (value initializations are fixed and there are no inputs from externals and always has a single execution path), there is no need to find the existence of invalid inputs or counterexamples.
- For closed-world programs, only **logical errors** are verified against assertions, rather than finding the **counterexamples**. Simply checking satisfiability is **the same** as checking the non-existence of counterexamples.
 - Checking **unsat** of the $\psi(N_1) \wedge \psi(N_2) \wedge \dots \psi(N_i) \wedge \neg\psi(Q)$.
 - Checking **sat** of the $\psi(N_1) \wedge \psi(N_2) \wedge \dots \psi(N_i) \wedge \psi(Q)$.

```
void main(int x){  
    x = 5;                                 $\psi_{path_1}$ : (if branch)  
    if(x > 10)                             checking unsat of  $x \equiv 5 \wedge ((x > 10) \wedge (y \equiv x + 1)) \wedge \neg(y \geq x + 1)$   
        y = x + 1;                       checking sat of  $x \equiv 5 \wedge ((x > 10) \wedge (y \equiv x + 1)) \wedge (y \geq x + 1)$   
    else  
        y = 10;                           $\psi_{path_2}$ : (else branch)  
    assert(y >= x + 1);                  checking unsat of  $x \equiv 5 \wedge ((x \leq 10) \wedge (y \equiv 10)) \wedge \neg(y \geq x + 1)$   
}
```

checking **sat** of $x \equiv 5 \wedge ((x \leq 10) \wedge (y \equiv 10)) \wedge (y \geq x + 1)$

Reachability Paths (Recall Assignment-1)

Algorithm 1: Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector(ICFGNode) callstack : vector(SVFInstruction)
visited : set(ICFGNode, callstack);

```
1 dfs(curNode, snk)           // Argument curNode becomes to curEdge in Assignment-2
2 pair = <curNode, callstack>;
3 if pair ∈ visited then
4   return;
5 visited.insert(pair);
6 path.push_back(curNode);
7 if src == snk then
8   collectICFGPath(path);    // collectAndTranslatePath in Assignment-2
9 foreach edge ∈ curNode.getOutEdges() do
10  if edge.isIntraCFGEde() then
11    | dfs(edge.dst, snk);
12  else if edge.isCallCFGEde() then
13    | callstack.push_back(edge.getCallSite());
14    | dfs(edge.dst, snk);
15    | callstack.pop_back();
16  else if edge.isRetCFGEde() then
17    | if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18    |   | callstack.pop_back();
19    |   | dfs(edge.dst, snk);
20    |   | callstack.push_back(edge.getCallSite());
21    | else if callstack == ∅ then
22    |   | dfs(edge.dst, snk);
23 visited.erase(pair);
24 path.pop_back();
```

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	$p = \&q \rightarrow fld$	$p \xleftarrow{Gep, fld} q$	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 rev)	$p \xleftarrow{Copy} q$	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 $c \equiv 1$, r is fval when $c \equiv 0$	

lhs and rhs When Handling Calls/Returns

Let us see the example of lhs and rhs variables for call and parameter passings (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 }
```

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

Context-Sensitive `getZ3Expr(NodeID)`

- `callingCtx`: current calling context stack as a sequence of callsites (`ICFGNodes`) during your traversal
- `pushCallingCtx(ICFGNode*)` add a callsite on to the current calling context stack
- `popCallingCtx()` pop the top callsite from the current calling context stack
- `getZ3Expr(NodeID)` in Assignment-2 is context-sensitive, i.e., retrieving an `z3` expr based on **variable's ID + current calling context**.

Context-Sensitive `getZ3Expr(NodeID)`

- `callingCtx`: current calling context stack as a sequence of callsites (ICFGNodes) during your traversal
- `pushCallingCtx(ICFGNode*)` add a callsite on to the current calling context stack
- `popCallingCtx()` pop the top callsite from the current calling context stack
- `getZ3Expr(NodeID)` in Assignment-2 is context-sensitive, i.e., retrieving an z3 expr based on **variable's ID + current calling context**.
- Interprocedural context-sensitive variable copies:
 - $r = \text{call_f}(\dots, q, \dots) \quad f(\dots, p, \dots) \{ \dots \text{return } z \}$
 - Context-insensitive z3 formula $p \equiv q, \quad r \equiv z$
 - Context-sensitive z3 formula (each expr corresponds to a pair) $\langle [c'], p \rangle \equiv \langle [c], q \rangle$
 $\langle [c], r \rangle \equiv \langle [c'], z \rangle$, where $c' \equiv c + \text{ICFGNode}_{\text{call_f}}$ (Assignment-2)
 - Use `pushCallingCtx` and `popCallingCtx` before retrieving an z3 expression.

Overview of SSE Algorithms: Translate Paths into Z3 Formulas

Algorithm 2: `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 3: `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 4: `handleCall(callEdge)`

```
1 // Your code starts here ..;
2
3 // For each CallPE lhs = rhs:
4 // (1) Retrieve two z3 expressions  $\langle [c], rhs \rangle$  and  $\langle [c'], lhs \rangle$  under contexts  $c$  (caller's context) and  $c'$  (callee's context); Note that pushCallingCtx will make  $c$  become  $c'$ .
5 // (2) Add formula  $\langle [c], rhs \rangle \equiv \langle [c'], lhs \rangle$  into the solver.
```

Algorithm 5: `handleRet(retEdge)`

```
1 // Your code starts here ..;
2
3 // For each RetPE lhs = rhs:
4 // (1) Retrieve two z3 expressions  $\langle [c'], rhs \rangle$  and  $\langle [c], lhs \rangle$  under contexts  $c$  (caller's context) and  $c'$  (callee's context); Note that popCallingCtx will make  $c'$  become  $c$ .
5 // (2) Add formula  $\langle [c'], rhs \rangle \equiv \langle [c], lhs \rangle$  into the solver.
```

Handle Intra-procedural CFG Edges (handleIntra)

Algorithm 6: 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 7: 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 8: 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

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

```
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

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)

```
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 }
```

Checking satisfiability using “getSolver().check()”.

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

Example 1: CMPSTMT and BINARYOPSTMT

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)

```
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 }
```

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

Handling CMPSTMT

Algorithm 9: 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
8     Use the ite (if – then – else) API to return 1
9     if operands are equal, 0 otherwise
10    then add to z3 solver
11   case CmpStmt :: ICMP_NE do
12     // handle not equal;
13   case CmpStmt :: ICMP_UGT do
14     Use the ite (if – then – else) API to return 1
15     // handle unsigned greater than;
16   case CmpStmt :: ICMP_SGT do
17     // handle signed greater than;
18   ...
```

Algorithm 10: 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 10: 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 10: 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));
```

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

Concrete Execution
(Concrete states)

```
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 }
```

One execution:

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

Another execution:

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

Example 2: Memory Operation

Concrete Execution
(Concrete states)

```
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 }
```

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:

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

Handling Memory Operation

Algorithm 11: Handle ADDRSTMT

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

Algorithm 12: 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 13: 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

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

```
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

	Concrete Execution (Concrete states) One execution:		Symbolic Execution (Symbolic states)
	x : 10		
1 struct st{	p : 0x1234	x : getZ3Expr(x)	
2 int a;	&(p→b) : 0x1238	p : 0x7f000001	
3 int b;	q : 0x1238		virtual address from
4 }	0x1238 : 10		getMemObjAddress(ObjVarID)
5 void main(int x) {	k : 10	&(p→b) : 0x7f000002	
6 struct st* p = malloc(..);	Another execution:	q : 0x7f000002	
7 q = &(p→b);	x : 20		field virtual address from
8 *q = x;	p : 0x1234		getGepObjAddress(base, offset)
9 int k = p→b;	&(p→b) : 0x1238	0x7f000002 : getZ3Expr(x)	
10 assert(k == x);	q : 0x1238	k : getZ3Expr(x)	
11 }	0x1238 : 20		
	k : 20		

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

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)
```

```
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 }
```

Checking non-existence of counterexamples:

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

Handling Field and Array Access (GEPSTMT)

Algorithm 13: 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 don't consider object byte sizes** and 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)));  
}
```


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)

One execution:

x : 20
y : 21

Another execution:

x : 8
y : 10

Symbolic Execution
(Symbolic states)

If branch:

x : $\text{getZ3Expr}(x) > 10$
y : $\text{getZ3Expr}(x) + 1$

Else branch:

x : $\text{getZ3Expr}(x) \leq 10$
y : 10

Checking non-existence of counterexamples:

Path	$\psi(N_1) \wedge \psi(N_2) \wedge \dots \wedge \psi(N_i) \wedge \neg\psi(Q)$	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	\emptyset
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_6 \rightarrow \ell_8$ (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()`”.

What's next?

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