Code Verification Using Symbolic Execution

(Week 5)

Yulei Sui

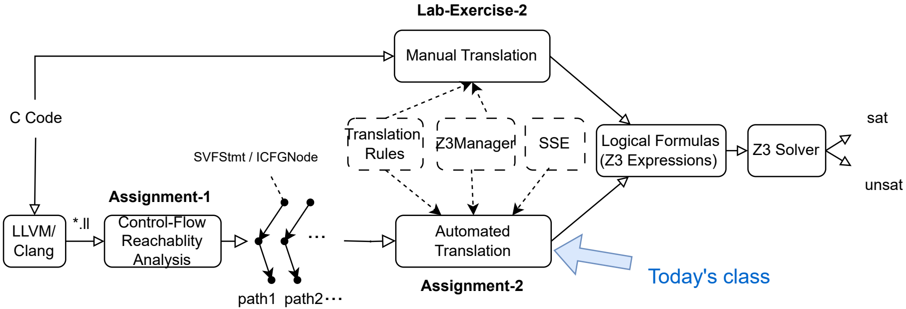
School of Computer Science and Engineering

University of New South Wales, Australia

Revsit Lab-Exercise-2 Cases

- Lab-Exercise-2 Validation Code
  - The validation code in `test1()` to `test2()` is not meant to be complete. Given a program `prog` and an assert `Q`, you are expected to (1) translate the negation of `Q` and check **unsat** of $prog \wedge \neg Q$ to prove 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("a") == 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.
- Branches (path feasibility)

Code Verification Using Static Symbolic Execution



Today's class

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
```

$$\xrightarrow{\text{translate}} \underset{\text{logical formula } \psi}{\underline{\exists \mathtt{x}\ \exists \mathtt{y}\ \ P \land S_{prog}(\mathtt{x}, \mathtt{y})) \land \neg Q(\mathtt{x}, \mathtt{y})}} \xrightarrow{\text{feed into}} \text{SMT Solver}$$

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
```

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

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 $path = [N_1, N_2, \ldots, N_i, Q]$, 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 validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - 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 validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \land \psi(N_2) \land \ldots \psi(N_i) \land \neg\psi(Q)$
  - 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}: $\exists \mathtt{x}\ true \land ((\mathtt{x} > 10) \land (\mathtt{y} \equiv \mathtt{x} + 1)) \land \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`if` branch)

unsat (no counterexample found!)

ψ_{path_2}: $\exists \mathtt{x}\ true \land ((\mathtt{x} \leq 10) \land (\mathtt{y} \equiv 10)) \land \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`else` branch)

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 $\psi(N_1) \land \psi(N_2) \land \ldots \psi(N_i) \land \neg\psi(Q)$.
  - Checking **sat** of the $\psi(N_1) \land \psi(N_2) \land \ldots \psi(N_i) \land \psi(Q)$.

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

ψ_{path_1}: (`if` branch)
checking **unsat** of $x \equiv 5 \land ((\mathrm{x} > 10) \land (\mathrm{y} \equiv \mathrm{x} + 1)) \land \neg(\mathrm{y} \geq \mathrm{x} + 1)$
checking **sat** of x $x \equiv 5 \land ((\mathrm{x} > 10) \land (\mathrm{y} \equiv \mathrm{x} + 1)) \land (\mathrm{y} \geq \mathrm{x} + 1)$

ψ_{path_2}: (`else` branch)
checking **unsat** of $x \equiv 5 \land ((\mathrm{x} \leq 10) \land (\mathrm{y} \equiv 10)) \land \neg(\mathrm{y} \geq \mathrm{x} + 1)$
checking **sat** of $x \equiv 5 \land ((\mathrm{x} \leq 10) \land (\mathrm{y} \equiv 10)) \land (\mathrm{y} \geq \mathrm{x} + 1)$

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⟩;
dfs(curNode, snk)                  // Argument curNode becomes to curEdge in Assignment-2
  pair = ⟨curNode, callstack⟩;
  if pair $\in$ visited then
      return;
  visited.insert(pair);
  path.push_back(curNode);
  if src == snk then
    collectICFGPath(path);       // collectAndTranslatePath in Assignment-2
  foreach edge $\in$ curNode.getOutEdges() do
    if edge.isIntraCFGEdge() then
        dfs(edge.dst, snk);
    else if edge.isCallCFGEdge() then
        callstack.push_back(edge.getCallSite());
        dfs(edge.dst, snk);
        callstack.pop_back();
    else if edge.isRetCFGEdge() then
        if callstack $\neq \emptyset$ && callstack.back() == edge.getCallSite() then
            callstack.pop_back();
            dfs(edge.dst, snk);
            callstack.push_back(edge.getCallSite());
        else if callstack == $\emptyset$ then
            dfs(edge.dst, snk);
  visited.erase(pair);
  path.pop_back();
```

Overview of SSE Algorithms: Translate Paths into Z3 Formulas

Algorithm 2: `translatePath(path)`

```
foreach edge ∈ path do
    if intra ← dyn_cast⟨Intra⟩(edge) then
        if handleIntra(intra) == false then
            return false
    else if call ← dyn_cast⟨CallEdge⟩(edge) then
        handleCall(call)
    else if ret ← dyn_cast⟨RetEdge⟩(edge) then
        handleRet(ret)
    return true
```

Algorithm 3: `handleIntra(intraEdge)`

```
if intraEdge.getCondition() then
    if !handleBranch(intraEdge) then
        return false;
    else
        return handleNonBranch(intraEdge);
else
    return handleNonBranch(intraEdge);
```

Algorithm 4: `handleCall(callEdge)`

```
expr_vector preCtxExprs(getCtx()); // rhs of call edges
callPEs ← calledge → getCallPEs();
foreach callPE ∈ callPEs do
    preCtxExprs.push_back(rhs); //rhs under the context before entering callee
pushCallingCtx(calledge → getCallSite());
for i = 0; i < callPEs.size(); + + i do
    lhs ← getZ3Expr(callPEs[i] → getLHSVarID()); //lhs under the context after entering callee
    addToSolver(lhs == preCtxExprs[i]);
```

Algorithm 5: `handleRet(retEdge)`

```
rhs(getCtx());        // expr for rhs of the return edge
if retPE ← retEdge.getRetPE() then
    rhs ← getZ3Expr(retPE.getRHSVarID()); //rhs under the context before returning to caller
popCallingCtx();
if retPE ← retEdge.getRetPE() then
    lhs ← getZ3Expr(retPE.getLHSVarID()); //lhs under the context after returning to caller
    addToSolver(lhs == rhs);
```

Handle Intra-procedural CFG Edges (`handleIntra`)

Algorithm 6: `handleIntra(intraEdge)`

```
if intraEdge.getCondition() then
    if !handleBranch(intraEdge) then
        return false;
    else
        return handleNonBranch(intraEdge);
else
    return handleNonBranch(intraEdge);
```

Algorithm 7: `handleBranch(intraEdge)`

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Algorithm 8: `HandleNonBranch(intraEdge)`

```
dst ← intraEdge.getDstNode();
src ← intraEdge.getSrcNode();
foreach stmt ∈ dst.getSVFStmts() do
    if addr ← dyn_cast⟨AddrStmt⟩(stmt) then
        // handle AddrStmt
    else if copy ← dyn_cast⟨CopyStmt⟩(stmt) then
        // handle CopyStmt
    else if load ← dyn_cast⟨LoadStmt⟩(stmt) then
        // handle LoadStmt
    else if store ← dyn_cast⟨StoreStmt⟩(stmt) then
        // handle StoreStmt
    else if gep ← dyn_cast⟨GepStmt⟩(stmt) then
        // handle GepStmt
    else if binary ← dyn_cast⟨BinaryStmt⟩(stmt) then
        // handle BinaryStmt
    else if cmp ← dyn_cast⟨CmpStmt⟩(stmt) then
        // handle CmpStmt
    else if phi ← dyn_cast⟨PhiStmt⟩(stmt) then
        // handle PhiStmt
    else if select ← dyn_cast⟨SelectStmt⟩(stmt) then
        // handle SelectStmt
    ...
```

Example 1: CMPSTMT and BINARYOPSTMT

```
void main(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

Example 1: CMPSTMT and BINARYOPSTMT

```
void main(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

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

```c
void main(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

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

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

Checking non-existence of counterexamples: $\psi(N_1) \land \psi(N_2) \land \ldots \psi(N_i) \land \neg\psi(Q)$	Satisfiability
$\mathtt{y} \equiv \mathtt{x} \land \mathtt{b} \equiv \mathtt{ite}(\mathtt{x} \equiv \mathtt{y}, 1, 0) \land \mathtt{z} \equiv \mathtt{x} + \mathtt{y} \land \mathtt{z} \not\equiv 2 * \mathtt{x}$	unsat

Example 1: CMPSTMT and BINARYOPSTMT

```
void main(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

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(x)`
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 the assume that the program is **integer-overflow-free** in this assignment.

Pseudo-Code for Handling CMPSTMT

Algorithm 9: `Handle CMPSTMT`

```
op0 ← getZ3Expr(cmp.getOpVarID(0));
op1 ← getZ3Expr(cmp.getOpVarID(1));
res ← getZ3Expr(cmp.getResID());
switch cmp.getPredicate() do
    case CmpInst :: ICMP_EQ do
        addToSolver(res == ite(op0 == op1,
          getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_NE do
        addToSolver(res == ite(op0! = op1,
          getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_UGT do
        addToSolver(res == ite(op0 > op1,
          getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_SGT do
        addToSolver(res == ite(op0 > op1,
          getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_UGE do
        addToSolver(res == ite(op0 >= op1,
          getCtx().int_val(1), getCtx().int_val(0)));
    ...
```

Algorithm 10: `Pseudo-Code for Handling CMPSTMT`

```
case CmpInst :: ICMP_SGE do
    addToSolver(res == ite(op0 >= op1,
      getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_ULT do
    addToSolver(res == ite(op0 < op1,
      getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_SLT do
    addToSolver(res == ite(op0 < op1,
      getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_ULE do
    addToSolver(res == ite(op0 <= op1,
      getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_SLE do
    addToSolver(res == ite(op0 <= op1,
      getCtx().int_val(1), getCtx().int_val(0)));
```

Handle BINARYOPSTMT

Algorithm 10: Handle BINARYOPSTMT

```
op0 ← getZ3Expr(binary.getOpVarID(0));
op1 ← getZ3Expr(binary.getOpVarID(1));
res ← getZ3Expr(binary.getResID());
switch binary.getOpcode() do
    case BinaryOperator :: Add do
        addToSolver(res == op0 + op1);
    case BinaryOperator :: Sub do
        addToSolver(res == op0 − op1);
    case BinaryOperator :: Mul do
        addToSolver(res == op0 × op1);
    case BinaryOperator :: SDiv do
        addToSolver(res == op0/op1);
    case BinaryOperator :: SRem do
        addToSolver(res == op0%op1);
    case BinaryOperator :: Xor do
        addToSolver(res ==
          bv2int(int2bv(32, op0) ⊕ int2bv(32, op1), 1)));
    case BinaryOperator :: And do
        addToSolver(res ==
          bv2int(int2bv(32, op0)&int2bv(32, op1), 1)));
    ...
```

Algorithm 10: Handle BINARYOPSTMT

```
case BinaryOperator :: Or do
    addToSolver(res ==
      bv2int(int2bv(32, op0)|int2bv(32, op1), 1)));
case BinaryOperator :: AShr do
    addToSolver(res ==
      bv2int(ashr(int2bv(32, op0), int2bv(32, op1)), 1)));
case BinaryOperator :: Shl do
    addToSolver(res ==
      bv2int(shl(int2bv(32, op0), int2bv(32, op1)), 1)));
```

Example 2: Memory Operation

```c
void main(int x) {
 int* p;
 int y;

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

Example 2: Memory Operation

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

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

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

```c
void main(int x) {
 int* p;
 int y;

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

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

Pseudo-Code for Handling Memory Operation

Algorithm 11: Handle ADDRSTMT

```
obj ← getMemObjAddress(addr.getRHSVarID());
lhs ← getZ3Expr(addr.getLHSVarID());
addToSolver(obj == lhs);
```

Algorithm 13: Handle STORESTMT

```
lhs ← getZ3Expr(store.getLHSVarID());
rhs ← getZ3Expr(store.getRHSVarID());
z3Mgr.storeValue(lhs, rhs);
```

Algorithm 12: Handle LOADSTMT

```
lhs ← getZ3Expr(load.getLHSVarID());
rhs ← getZ3Expr(load.getRHSVarID());
addToSolver(lhs == z3Mgr.loadValue(rhs));
```

Example 3: Field Access for Struct and Array

```
struct st{
    int a;
    int b;
}
void main(int x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 int k = p->b;
 assert(k == x);
}
```

Example 3: Field Access for Struct and Array

```c
struct st{
    int a;
    int b;
}
void main(int x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 int k = p->b;
 assert(k == x);
}
```

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

```
struct st{
    int a;
    int b;
}
void main(int x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 int k = p->b;
 assert(k == x);
}
```

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

```c
struct st{
    int a;
    int b;
}
void main(int x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 int k = p->b;
 assert(k == x);
}
```

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
$\texttt{p} \equiv \texttt{0x7f000001} \wedge \texttt{q} \equiv \texttt{0x7f000002} \wedge \texttt{k} \equiv \texttt{x} \wedge \texttt{k} \neq \texttt{x}$	unsat

Pseudo-Code for Handling Field and Array Access (GEPSTMT)

Algorithm 13: `Handle GEPSTMT`

```
lhs ← getZ3Expr(gep.getLHSVarID());
rhs ← getZ3Expr(gep.getRHSVarID());
offset ← z3Mgr.getGepOffset(gep, curCallCtx);
gepAddress ← z3Mgr.getGepObjAddress(rhs, offset);
addToSolver(lhs == gepAddress);
```

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

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

Example 4: Branches

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

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)	$\mathtt{x} > 10 \wedge \mathtt{y} \equiv \mathtt{x} + 1 \wedge \mathtt{y} < \mathtt{x} + 1$	unsat	$\varnothing$
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_6 \rightarrow \ell_8$ (if.else branch)	$\mathtt{x} \leq 10 \wedge \mathtt{y} \equiv 10 \wedge \mathtt{y} < \mathtt{x} + 1$	sat	$\{\mathtt{x} : 10, \mathtt{y} : 10\}$

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