Code Verification Using Symbolic Execution

(Week 5)

Yulei Sui

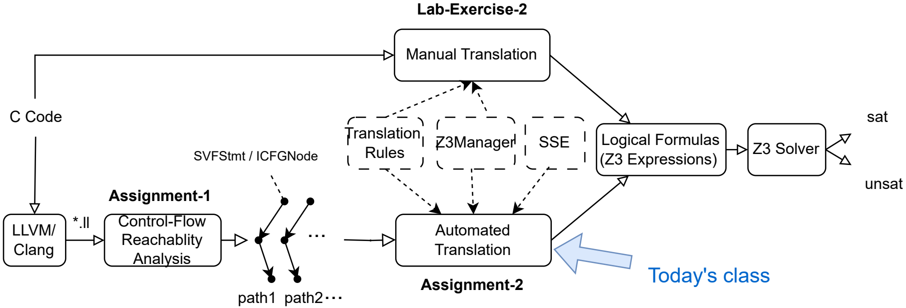
School of Computer Science and Engineering

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("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.

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
```

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

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).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - 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}: $\exists \mathtt{x}\ true \wedge ((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`if` branch)
unsat (no counterexample found!)

ψ_{path_2}: $\exists \mathtt{x}\ true \wedge ((\mathtt{x} \leq 10) \wedge (\mathtt{y} \equiv 10)) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`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 $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$.
  - Checking **sat** of the $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \psi(Q)$.

```
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 $x \equiv 5 \wedge ((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$
checking **sat** of $\mathtt{x}$ $x \equiv 5 \wedge ((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)) \wedge (\mathtt{y} \geq \mathtt{x} + 1)$

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

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 ∈ visited then
      return;
  visited.insert(pair);
  path.push_back(curNode);
  if src == snk then
    collectICFGPath(path);         // collectAndTranslatePath in Assignment-2
  foreach edge ∈ 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 ≠ ∅ && callstack.back() == edge.getCallSite() then
        callstack.pop_back();
        dfs(edge.dst, snk);
        callstack.push_back(edge.getCallSite());
      else if callstack == ∅ then
        dfs(edge.dst, snk);
  visited.erase(pair);
  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->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≡1, `r` is `fval` when c≡0	

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

Let us see the example of `lhs` and `rhs` varaiables 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 (`ICFGNode`s) 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 (`ICFGNode`s) 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:
  - $\mathtt{r} = \mathtt{call\_f}(\ldots, \mathrm{q}, \ldots) \quad \mathtt{f}(\ldots, \mathrm{p}, \ldots)\{\ldots\ \mathtt{return}\ \mathrm{z}\}$
  - Context-insensitive z3 formula $\mathrm{p} \equiv \mathrm{q}, \quad \mathrm{r} \equiv \mathrm{z}$
  - Context-sensitive z3 formula (each expr corresponds to a pair) $\langle [\mathrm{c}'], \mathrm{p}\rangle \equiv \langle [\mathrm{c}], \mathrm{q}\rangle$ $\langle [\mathrm{c}], \mathrm{r}\rangle \equiv \langle [\mathrm{c}'], \mathrm{z}\rangle$, where $\mathrm{c}' \equiv \mathrm{c} + \mathtt{ICFGNode}_{\mathtt{call\_f}}$ (Assignment-2)
  - Use `pushCallingCtx` and `popCallingCtx` before retrieving an z3 expression.

Overview of SSE Algorithms: Translate Paths into Z3 Formulas

Algorithm 2: translatePath(path)

```
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)

```
// Your code starts here ..;

// For each CallPE lhs = rhs:
// (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'.
// (2) Add formula ⟨[c], rhs⟩ ≡ ⟨[c′], lhs⟩ into the solver.
```

Algorithm 5: handleRet(retEdge)

```
// Your code starts here ..;

// For each RetPE lhs = rhs:
// (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.
// (2) Add formula ⟨[c′], rhs⟩ ≡ ⟨[c], lhs⟩ into the solver.
```

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)

```
// Your code starts here ..;
// Retrieve branch condition (0 or 1);
cond = ...;
// Retrieve the condition on the current CFG edge (0 or 1);
succ = ...;
// 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)

```
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

```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)
}
```

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

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)`

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
$\mathtt{y} \equiv \mathtt{x} \wedge \mathtt{b} \equiv \mathtt{ite}(\mathtt{x} \equiv \mathtt{y}, 1, 0) \wedge \mathtt{z} \equiv \mathtt{x} + \mathtt{y} \wedge \mathtt{z} \not\equiv 2 * \mathtt{x}$	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)`$\equiv$`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.

Handling CmpStmt

Algorithm 9: Handle CmpStmt

```
op0 ← get z3 expression of operand 0;
op1 ← get z3 expression of operand 1;
res ← get z3 expression of result;
// refer to SVF's CmpStmt::Predicate;
switch predicate of cmp do
    case CmpStmt :: ICMP_EQ do
        // handle equal
        Use the ite (if − then − else) API to return 1
          if operands are equal, 0 otherwise
          then add to z3 solver
    case CmpStmt :: ICMP_NE do
        // handle not equal;
    case CmpStmt :: ICMP_UGT do
        Use the ite (if − then − else) API to return 1
        // handle unsigned greater than;
    case CmpStmt :: ICMP_SGT do
        // handle signed greater than;
    ...
```

Algorithm 10: Handling CmpStmt

```
case CmpStmt :: ICMP_UGE do
    // handle unsigned greater equal;
case CmpStmt :: ICMP_SGE do
    // handle signed greater equal;
case CmpStmt :: ICMP_ULT do
    // handle unsigned less than;
case CmpStmt :: ICMP_SLT do
    // handle signed less than;
case CmpStmt :: ICMP_ULE do
    // handle unsigned less equal;
case CmpStmt :: ICMP_SLE do
    // handle signed less equal;
```

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

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

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

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

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
$\mathtt{p} \equiv \mathtt{0x7f000001} \wedge \mathtt{y} \equiv \mathtt{x} + 5 \wedge \mathtt{y} \not\equiv \mathtt{x} + 5$	unsat

Handling Memory Operation

Algorithm 11: Handle AddrStmt

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

Algorithm 12: Handle LoadStmt

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

Algorithm 13: Handle StoreStmt

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

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);
}
```

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
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	10
k	:	10

Another execution:

x	:	20
p	:	0x1234
$\&(p \rightarrow b)$	:	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

```
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
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	10
k	:	10

Another execution:

x	:	20
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	20
k	:	20

Symbolic Execution
(Symbolic states)

x	:	getZ3Expr(x)
p	:	0x7f000001 virtual address from getMemObjAddress(ObjVarID)
$\&(p \rightarrow b)$	:	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
$p \equiv$ 0x7f000001 $\wedge\ q \equiv$ 0x7f000002 $\wedge\ k \equiv x \wedge k \neq x$	unsat

Handling Field and Array Access (GEPSTMT)

Algorithm 13: Handle GEPSTMT

```
lhs ← get z3 expression of lhs var;
rhs ← get z3 expression of rhs var;
offset ← get gep offset under curCallCtx;
gepAddress ← get gep address of rhs given offset;
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

```
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)

One execution:

$\texttt{x}$: 20

$\texttt{y}$: 21

Another execution:

$\texttt{x}$: 8

$\texttt{y}$: 10

Symbolic Execution (Symbolic states)

If branch:

$\texttt{x}$: $\texttt{getZ3Expr(x)} > 10$

$\texttt{y}$: $\texttt{getZ3Expr(x)} + 1$

Else branch:

$\texttt{x}$: $\texttt{getZ3Expr(x)} \leq 10$

$\texttt{y}$: 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
$\ell_1 \to \ell_2 \to \ell_3 \to \ell_8$ (if.then branch)	$\texttt{x} > 10 \wedge \texttt{y} \equiv \texttt{x} + 1 \wedge \texttt{y} < \texttt{x} + 1$	unsat	$\varnothing$
$\ell_1 \to \ell_2 \to \ell_6 \to \ell_8$ (if.else branch)	$\texttt{x} \leq 10 \wedge \texttt{y} \equiv 10 \wedge \texttt{y} < \texttt{x} + 1$	sat	$\{\texttt{x} : 10, \texttt{y} : 10\}$

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