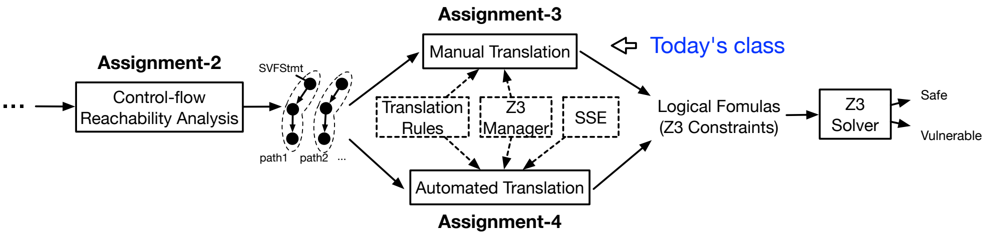
Software Verification and Z3 Theorem Prover

Yulei Sui

University of Technology Sydney, Australia

Today's class



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

Z3 Theorem Prover

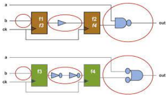
- Z3 is a Satisfiability Modulo Theories (SMT) solver from Microsoft Research[1].
- Targeted at solving problems in software verification and software analysis.
- Main applications are static checking, test case generation, and more ..

Hardware verification

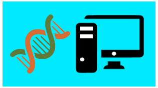
Software analysis/testing

Architecture

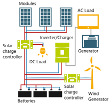
Modeling



Geometrical solving



Biological analysis

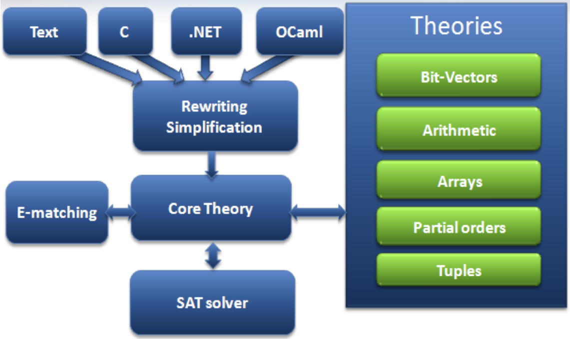


Hybrid system analysis

...

[1] http://research.microsoft.com/projects/z3

Z3 Framework[2]



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

[2] `https://nikolajbjorner.github.io/slides/Z3_System.pdf`

Z3 Learning Materials

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

Z3 Solver and Z3 Formulas

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

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

Z3 Solver and Z3 Formulas

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

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

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

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

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

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

[3] `https://homepage.cs.uiowa.edu/~tinelli/papers/BarST-SMT-10.pdf`

Constants, Check-Sat and Evaluation (Example)

$$\phi : (\mathtt{x} > \mathtt{10}) \ \land \ (\mathtt{y} \equiv \mathtt{x} + \mathtt{1})$$

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

Constants, Check-Sat and Evaluation (Example)

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

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

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

Constants, Check-Sat and Evaluation (Example)

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

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

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

Outputs of Z3's solver:

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

Z3's Logical Formula (Uninterpreted Function)

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

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

Uninterpreted Function (Example)

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

Uninterpreted Function (Example)

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

Outputs of Z3's solver:

```
sat
```

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

Uninterpreted Function (Example)

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

Outputs of Z3's solver:

```
sat
```

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

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

Uninterpreted Function (Example)

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

Outputs of Z3's solver:

```
sat
```

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

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

Outputs of Z3's solver:

```
unsat
```

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

Uninterpreted Function (Example)

$$\phi : \mathtt{f}(\mathtt{x}) \equiv \mathtt{f}(\mathtt{y}) \ \land \ \mathtt{x}! = \mathtt{y}$$

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

Uninterpreted Function (Example)

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

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

Outputs of Z3's solver:

```
sat
```

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

Constants as Uninterpreted Function (Example)

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

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

Outputs of Z3's solver:

```
sat
(
  (define-fun x () Int
    11)                     /// x is evaluated to be 11 for this model
  (define-fun y () Int
    12)                     /// y is evaluated to be 11 for this model
)
```

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

Z3's Logical Formula (Arithmetic)

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

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

Outputs of Z3's solver:

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

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

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

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

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

Z3's Logical Formula (Arrays)

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

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

Z3's Logical Formula (Arrays)

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

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

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

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

Z3's Logical Formula (Scopes)

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

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

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

Z3's Logical Formula (Scopes)

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

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

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

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

What is the output of the solver?

Translating Code to Z3 Formulas

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

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

More details, refer to

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

Translation Rules

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

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

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

UnaryOPStmt	$\neg p$	addToSolver(!p);
BinaryOPStmt	$r = p \otimes q$	addToSolver(r == p $\otimes$ q);
CmpStmt	$r = p \odot q$	addToSolver(r == ite(p $\odot$ q, true, false));
CallPE/RetPE	$r = f(\ldots, q, \ldots)$ $f(\ldots, p, \ldots)\{\ldots$ return $z\}$	
CallPE	p = q	solver.push(); addToSolver(p == q);
RetPE	p = r	expr ret = getZ3Expr(r); solver.pop(); addToSolver(p == ret);

Translating Code to Z3 Formulas (Scalar Example)

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

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

Translating Code to Z3 Formulas (Scalar Example)

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

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

```c
int main() {

 int a;
 int b;
 a = 0;
 b = a + 1;
 assert(b>0);

}
```

C code

$\rightarrow$

```cpp
// int a;
expr a = getZ3Expr("a");
// a = 0;
addToSolver(a == 0);
// int b;
expr b = getZ3Expr("b");
// b = a+1;
addToSolver(b == (a + 1));
// assert(b > 0);
addToSolver(b > 0);
solver.check();
```

Translator

$\rightarrow$

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (= a 0))
(assert (= b (+ a 1)))
(assert (> b 0))
(check-sat)
```

Z3 Formulas

$\rightarrow$ Z3's SMT solver

Translating Code to Z3 Formulas (Memory Operation Example)

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

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

 p = malloc1(..);
 *p = 5;
 x = *p;
 assert(x==5);
}
```

→

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

→

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

C code | Translator | Z3 Formulas

What's next?

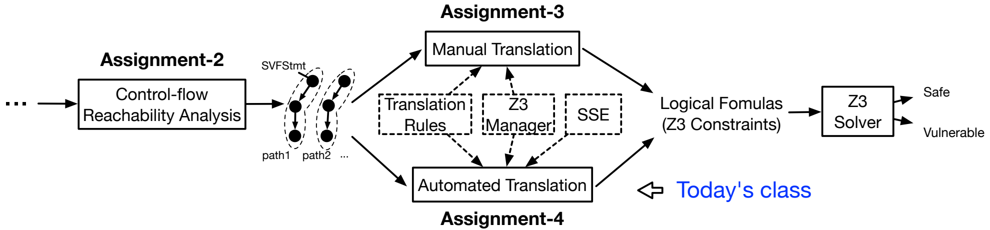
- (1) Understand Z3 formula format in the slides
- (2) Understand `Z3Mgr` class in the GitHub Repository of Teaching-Software-Verification
- (3) Finish the quizzes of Assignment 3 on Canvas
- (4) Implement a manual translation from code to Z3 formulas using `Z3Mgr` i.e., coding task in Assignment 3.

Assertion-based Verification Using Static Symbolic Execution

Yulei Sui

University of Technology Sydney, Australia

Automated Assertion-based Verification



Static Symbolic Execution (SSE)

- An static interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would.
- Automated testing technique that symbolically executes a program.
- Use symbolic execution to explore all program paths to find latent bugs.

Static Symbolic Execution for Assertion-based Verification

- (1) Given a Hoare triple $P\ \{prog\}\ Q$,
  - P represents program inputs,
  - *prog* is the actual source code,
  - Q is the assertion(s) to be verified.
- (2) SSE translates `SVFStmt` of each program path (which ends with an assertion) into an Z3 logical formula.
  - In our project, the path of each loop is bounded once for verification.
- (3) Proving satisibility of the logic formulas of each program path from the program entry to each assertion on the ICFG.

Driver Program of SSE (What We Have From Assignment 2)

Algorithm 1 Context sensitive control-flow reachability

```
Input : src : ICFGNode   dst : ICFGNode
        path : vector⟨ICFGNode⟩   visited : set⟨ICFGNode⟩;
dfs(path, src, dst)
  visited.insert(src)
  path.push_back(src)
  if src == dst then
    print path
  foreach edge ∈ src.getOutEdges() do
    if edge.dst ∉ visited then
      if edge.isIntraCFGEdge() then
        if handleIntra(edge) then
          dfs(path, edge.dst, dst)
      else if edge.isCallCFGEdge() then
        if handleCall(edge) then
          dfs(path, edge.dst, dst)
      else if edge.isRetCFGEdge() then
        if handleRet(edge) then
          dfs(path, edge.dst, dst)
  visited.erase(src)
  path.pop_back(src)
```

Algorithm 2 handleIntra(intraEdge) (Override in SSE)

```
return true
```

Algorithm 3 handleCall(callEdge) (Override in SSE)

```
callNode ← getSrcNode(callEdge)
callstack.push_back(callNode)
return true
```

Algorithm 4 handleRet(retEdge) (Override in SSE)

```
retNode ← getDstNode(retEdge)
if callstack ≠ ∅ then
  if callstack.back() == getCallICFGNode(retNode) then
    callstack.pop()
    return true
  else
    return false
return true
```

Driver Program of SSE (What We Have From Assignment 2)

Algorithm 1 Context sensitive control-flow reachability

Input : src : ICFGNode dst : ICFGNode
path : vector⟨ICFGNode⟩ visited : set⟨ICFGNode⟩;

```
dfs(path, src, dst)
  visited.insert(src)
  path.push_back(src)
  if src == dst then
    print path
  foreach edge ∈ src.getOutEdges() do
    if edge.dst ∉ visited then
      if edge.isIntraCFGEdge() then
        if handleIntra(edge) then
          dfs(path, edge.dst, dst)
      else if edge.isCallCFGEdge() then
        if handleCall(edge) then
          dfs(path, edge.dst, dst)
      else if edge.isRetCFGEdge() then
        if handleRet(edge) then
          dfs(path, edge.dst, dst)
  visited.erase(src)
  path.pop_back(src)
```

Algorithm 2 handleIntra(intraEdge) (Override in SSE)

```
return true
```

Algorithm 3 handleCall(callEdge) (Override in SSE)

```
callNode ← getSrcNode(callEdge)
callstack.push_back(callNode)
return true
```

Algorithm 4 handleRet(retEdge) (Override in SSE)

```
retNode ← getDstNode(retEdge)
if callstack ≠ ∅ then
  if callstack.back() == getCallICFGNode(retNode) then
    callstack.pop()
    return true
  else
    return false
return true
```

Override the above three methods in SSE implementation!

Handle Intra-procedural CFG Edges (`handleIntra`)

Algorithm 2 `handleIntra(intraEdge)`

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

`handleBranch(intraEdge)`

```
cond = intraEdge.getCondition()
successorVal = intraEdge.getSuccessorCondValue()
res = getEvalExpr(cond == suc)
if res.is_false() then
    addToSolver(cond! = suc)
    return false
else if res.is_true() then
    addToSolver(cond == suc)
    return true
else
    return true
```

`HandleNonBranch(intraEdge)`

```
dst ← intraEdge.getDstNode(); src ← intraEdge.getSrcNode()
foreach stmt ∈ dst.getSVFStmts() do
    if addr ∈ dyn_cast⟨AddrStmt⟩(stmt) then
        obj ← getMemObjAddress(addr.getRHSVarID())
        lhs ← getZ3Expr(addr.getLHSVarID())
        addToSolver(obj == lhs)
    else if copy ∈ dyn_cast⟨CopyStmt⟩(stmt) then
        lhs ← getZ3Expr(copy.getLHSVarID())
        rhs ← getZ3Expr(copy.getRHSVarID())
        addToSolver(rhs == lhs)
    else if load ∈ dyn_cast⟨LoadStmt⟩(stmt) then
        lhs ← getZ3Expr(load.getLHSVarID())
        rhs ← getZ3Expr(load.getRHSVarID())
        addToSolver(lhs == z3Mgr.loadValue(rhs))
    else if store ∈ dyn_cast⟨StoreStmt⟩(stmt) then
        lhs ← getZ3Expr(store.getLHSVarID())
        rhs ← getZ3Expr(store.getRHSVarID())
        z3Mgr.storeValue(lhs, rhs)
    else if gep ∈ dyn_cast⟨GepStmt⟩(stmt) then
        lhs ← getZ3Expr(gep.getLHSVarID())
        rhs ← getZ3Expr(gep.getRHSVarID())
        offset = z3Mgr.getGepOffset(gep)
        gepAddress = z3Mgr.getGepObjAddress(rhs, offset)
        addToSolver(lhs == gepAddress)
```

Handel Call (`handleCall`) and Return (`handleRet`) CFG Edges

Algorithm 3 `handleCall(callEdge)`

```
callNode ← callEdge.getSrcNode();
FunEntryNode ←callEdge.getDstNode();
callstack.push_back(callNode);
getSolver().push();
foreach callPE ∈ calledge.getCallPEs() do
    lhs ← getZ3Expr(callPE.getLHSVarID());
    rhs ← getZ3Expr(callPE.getRHSVarID());
    addToSolver(lhs == rhs);
return true;
```

Algorithm 4 `handleRet(retEdge)`

```
retNode ← retEdge.getDstNode();
rhs(getCtx());
lhs(getCtx());
if retPE = retEdge.getRetPE() then
    rhs ← getEvalExpr(getZ3Expr(retPE.getRHSVarID()));
    lhs ← getZ3Expr(retPE.getLHSVarID());
if callstack ≠ ∅ then
    if callstack.back() == getCallICFGNode(retNode) then
        callstack.pop_back();
        getSolver().pop();
    else
        return false;
if retEdge.getRetPE() then
    addToSolver(lhs == rhs);
return true;
```

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

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

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```c
void foo(unsigned x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);
}
```

Concrete Execution
(Concrete states of x, y)

One execution:
x : 20
y : 21

Another execution:
x : 8
y : 9

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

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

Concrete Execution
(Concrete states of `x`, `y`)

One execution:

`x` : 20
`y` : 21

Another execution:

`x` : 8
`y` : 9

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

If branch:

`x` : $\texttt{getZ3Expr(x)} > 10 \wedge \texttt{getZ3Expr(x)} < \text{UINT\_MAX}$
`y` : $\texttt{getZ3Expr(x)} + 1$

Else branch:

`x` : $\texttt{getZ3Expr(x)} > 0 \wedge \texttt{getZ3Expr(x)} < 10$
`y` : 10

Memory Operation Example

```
void foo(unsigned x) {
 int* p;
 int y;

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

Memory Operation Example

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

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

Memory Operation Example

```
void foo(unsigned x) {
 int* p;
 int y;

 p = malloc(..);
 *p = x + 5;
 y = *p;
 assert(y>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("malloc")
0x7f000001	:	getZ3Expr(x) + 5
y	:	getZ3Expr(x) + 5

Field Access for Struct and Array Example

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

Field Access for Struct and Array Example

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

Concrete Execution
(Concrete states)
One execution:

x	:	10
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	10

Another execution:

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

Field Access for Struct and Array Example

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

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

Symbolic Execution
(Symbolic states)

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

Call and Return Example

```
1  int foo(int z) {
2      k = z;
3      return k;
4  }
5  int main(unsigned z) {
6    int x;
7    int y;
8    x = foo(3);
9    y = foo(z);
10   assert(x == 3);
11 }
```

Concrete Execution
(Concrete states)

One execution:

`z` : 10
stack push (calling foo at line 8)
`k` : 3
stack pop (returning from foo at line 4)
`x` : 3
stack push (calling foo at line 9)
`k` : 10
stack pop (returning from foo line 4)
`y` : 10

Symbolic Execution
(Symbolic states)

One execution:

`z` : `getZ3Expr(z)`
stack push (calling foo at line 8)
`k` : 3
stack pop (returning from foo at line 4)
`x` : 3
stack push (calling foo at line 9)
`k` : `getZ3Expr(z)`
stack pop (returning from foo line 4)
`y` : `getZ3Expr(z)`

What's next?

- (1) Understand SSE algorithms in the slides
- (2) Finish the quizzes of Assignment 4 on Canvas
- (3) Implement a automated translation from code to Z3 formulas using `SSE` and `Z3Mgr` i.e., coding task in Assignment 3