本 科 实 验 报 告

实验名称：简单 DSE 实现

学　　员：马坚　　学　　号：201912509012

培养类型：指挥　　年　　级：2019 级

专　　业：软件工程　　所属学院：计算机学院

指导教员：陈振邦　　职　　称：

实 验 室：306-705　　实验日期：2022-6-26

国防科技大学训练部制

1. 实验目的和要求

1.1. 目的

通过编程实现一个简单的符号执行引擎，并通过测试用例。最终期望通过实验使大家掌握 Z3 求解器的相关 API 以及如何利用 LLVM Pass 进行插桩，完成对程序代码的分析。

1.2. 要求

1. 基于 LLVM 和 Z3，实现一个面向简单（基本数据类型、只有一个 main 函数）C 程序的动态符号执行工具。提供基本的代码框架，以及 PPT 讲解。

2. 提交实验报告+代码

3. 评判规则：《基本要求》是能正确分析提供的测试代码，在此基础上支持高级 C 程序特征（比如浮点、数组、结构体、过程间等）可作为加分项。代码会查重，被认定双方都以 0 分计。对提交的源码包的要求：保证给定的代码框架原始的编译命令可以编译通过。如果你有特殊编译需求，请在项目根目录下创建 compileCmd.md 给出你的编译命令以及说明。

2. 实验内容和原理

2.1. 内容

参考“实验二.pptx”与代码框架压缩包，基于已有框架实现动态符号执行工具。

2.2. 原理

动态符号执行，llvm pass 插桩，DFS 搜索策略。

3. 代码实现与编译

3.1. 代码实现

1. 插桩

1) *DSE _Init*

此函数用于初始阶段读取 input.txt 中的输入，即符号的初始值，因此需要将桩 DSE_Init 插在最开始的地方。

```
7 define dso_local i32 @main() #0 {
8 entry:
9   call void @__DSE_Init__()
10  %retval = alloca i32, align 4
11  %x = alloca i32, align 4
```

图 3-1-1 DSE_Init 桩函数位置

2) *DSE _Alloca*

当定义了一个变量时，程序会为此变量分配一个地址，在中间码体现为 alloca，这有点儿 malloca 分配内存的感觉在其中。在此处需要将桩 DSE_Alloca 插在其

后，在模拟的内存中为其分配一个寄存器，其值为变量的地址。

```
%x = alloca i32, align 4
call void @__DSE_Alloca__(i32 0, i32* %x)
%y = alloca i32, align 4
call void @__DSE_Alloca__(i32 1, i32* %y)
%z = alloca i32, align 4
call void @__DSE_Alloca__(i32 2, i32* %z)
```

图 3-1-2 DSE_Alloca 桩函数位置

```
extern "C" void __DSE_Alloca__(int R, int *Ptr) {
    expr e = SI.getContext().int_val( n: *Ptr);
    auto address = Address( ID: R);
    SI.getMemory().insert( x: make_pair( x: address,  y: e));
}
```

图 3-1-3 DSE_Alloca 桩函数实现代码

3) *DSE _Store*

当为变量赋值时，中间码会调用 store，这时候需要在其后插桩 DSE_Store，但需要注意赋值语句右侧的类型，按常量与变量划分两种不同情形。

a）常量

形如 x＝1；右侧为常量数字 1，此时搭配桩函数 DSE_Const，将常量 1 压入栈中，随后调用桩函数 DSE_Store 从栈中弹出常量 1，并将其存入对应地址的寄存器中去，格式为 z3::expr。

```
store i32 %0, i32* %y, align 4
call void @__DSE_Register__(i32 3)
call void @__DSE_Store__(i32* %y)
store i32 9, i32* %z, align 4
call void @__DSE_Const__(i32 9)
call void @__DSE_Store__(i32* %z)
```

图 3-1-4 DSE_Store 桩函数位置

```
extern "C" void __DSE_Const__(int X) {
  z3::expr SE = SI.getContext().int_val( n: X);
  SI.getStack().push( x: SE);
}
```

图 3-1-5 DSE_Const 桩函数实现代码

b）变量

形如 y＝x;右侧为变量 x，此时搭配桩函数 DSE_Register，将变量 x 对应的寄存器地址压入栈中，随后调用桩函数 DSE_Store 从栈中弹出 x 的地址，在模拟的内存 Menory 中找到对应地址的值，将值赋给 y 地址对应的寄存器。

```
extern "C" void __DSE_Register__(int X) {
  std::string Name = "R" + std::to_string( val: X);
  z3::expr SE = SI.getContext().int_const( name: Name.c_str());
  SI.getStack().push( x: SE);
}
```

图 3-1-6 DSE_Register 桩函数实现代码

```
extern "C" void __DSE_Store__(int *X) {
    expr e = SI.getStack().top();
    SI.getStack().pop();
    if(e.to_string().at( n: 0) == 'R'){
        e = SI.getMemory().at( k: e);
    }
    auto address = Address( Ptr: X);
    SI.getMemory().insert( x: make_pair( x: address,  y: e));
    SI.getMemory().at( k: address) = e;
}
```

图 3-1-7 DSE_Store 桩函数实现代码

4) *DSE _Load*

在上述的第二种情况下，其中间码的实现并不是直接 x 到 y，而是需要先读取 x 的值，此时就用到了 load 方法。对应的在其后需要插入桩函数 DSE_Load，读取变量 x 的值，赋给中间变量。

```
%0 = load i32, i32* %x, align 4
call void @__DSE_Load__(i32 3, i32* %x)
```

图 3-1-8 DSE_Load 桩函数位置

```
extern "C" void __DSE_Load__(int Y, int *X) {
    expr e = SI.getMemory().at( k: Address( Ptr: X));
    auto address = Address( ID: Y);
    SI.getMemory().insert( x: make_pair( x: address,  y: e));
    SI.getMemory().at( k: address) = e;
}
```

图 3-1-9 DSE_Load 桩函数代码实现

5) *DSE _BinOp*

基本的运算操作，如加减乘除取模。即使表达式右侧有若干变量，但在底层实现时都是两两运算得出结果再与接下来的参数运算，因此每次操作的操作数只有 2 个。这里与 DSE_Store 有异曲同工之处，同样要判断两个操作数的类型，是否是常量，这里不再赘述。

将操作数压入栈中，依据 DSE_BinOp 的第二个参数确定运算类型，将结果写入到寄存器中去。

```
%add = add nsw i32 %0, 1
call void @__DSE_Const__(i32 1)
call void @__DSE_Register__(i32 2)
call void @__DSE_BinOp__(i32 3, i32 13)
```

图 3-1-10 DSE_BinOp 桩函数位置

```
#define ADD         13      ///    +
#define SUB         15      ///    -
#define MUL         17      ///    *
#define SDIV        20      ///    /
#define SREM        23      ///    %
```

图 3-1-11 常量定义

```
extern "C" void __DSE_BinOp__(int R, int Op) {
    expr el = SI.getStack().top();
    SI.getStack().pop();
    expr er = SI.getStack().top();
    SI.getStack().pop();
    if(el.to_string().at( n: 0) == 'R'){
        el = SI.getMemory().at( k: el);
    }
    if(er.to_string().at( n: 0) == 'R'){
        er = SI.getMemory().at( k: er);
    }
```

图 3-1-12 DSE_BinOp 桩函数代码实现

6) *DSE _Icmp*

条件比较语句会用到 icmp，这时候需要插入桩函数 DSE_Icmp，这一个函数的实现与 DSE_BinOp 类似，除了参数不同，其余几乎一模一样。简单带过，将操作数压入栈中，调用 DSE_Icmp 弹出栈中内容，根据比较类型进行条件判断，将结果表达式压入对应地址的寄存器中。

```
%cmp = icmp eq i32 %0, 1
call void @__DSE_Const__(i32 1)
call void @__DSE_Register__(i32 3)
call void @__DSE_ICmp__(i32 4, i32 32)
```

图 3-1-13 DSE_Icmp 桩函数位置

```
#define ICMP_EQ     32  ///< equal
#define ICMP_NE     33  ///< not equal
#define ICMP_SGT    38  ///< signed greater than
#define ICMP_SGE    39  ///< signed greater or equal
#define ICMP_SLT    40  ///< signed less than
#define ICMP_SLE    41  ///< signed less or equal
```

图 3-1-14 常量定义

```
extern "C" void __DSE_ICmp__(int R, int Op) {
    expr el = SI.getStack().top();
    SI.getStack().pop();
    expr er = SI.getStack().top();
    SI.getStack().pop();
    if(el.to_string().at( n: 0) == 'R'){
        el = SI.getMemory().at( k: el);
    }
    if(er.to_string().at( n: 0) == 'R'){
        er = SI.getMemory().at( k: er);
    }
```

图 3-1-15 DSE_Icmp 桩函数代码实现

7) *DSE _Branch*

出现分支语句时，就需要插入桩函数 DSE_Branch，这也是最关键的函数，因为这直接影响到后面的动态符号执行求解。我的思路是，在遇到分支语句时，在 br 语句前插入桩函数 DSE_Label，将真假分支的地址及条件表达式存入到寄存器中，再插入 DSE_Loop 函数用于区分当前 br 语句是否为条件判断语句。

DSE_Label 从寄存器中读取 icmp 的表达式，通过参数 label 的值判断真假分支，如果是真分支则将 icmp==true 写入到真分支对应地址的寄存器中去，假分支同理。当 br 语句是条件判断语句时，栈中压入 1，反之压入 0。

在真假分支语句对应的地址处，插入 DSE_Branch 桩函数，读取寄存器中的条件表达式。

```
  call void @__DSE_Label__(i32 4, i32 5, i32 1)
  call void @__DSE_Label__(i32 4, i32 6, i32 0)
  call void @__DSE_Loop__(i32 1)
  br i1 %cmp, label %if.then, label %if.else

if.then:
  call void @__DSE_Branch__(i32 5, i32 0)
```

图 3-1-16 DSE_Branch 桩函数位置

```
extern "C" void __DSE_Label__(int RID, int BID, int label){
    expr el = SI.getMemory().at( k: Address( ID: RID));
    auto address = Address( ID: BID);
    expr t = SI.getContext().bool_val( b: true);
    expr f = SI.getContext().bool_val( b: false);
    if(label){
        SI.getMemory().insert( x: make_pair( x: address,  y: el == t));
        SI.getMemory().at( k: address) = (el == t);
    } else{
        SI.getMemory().insert( x: make_pair( x: address,  y: el == f));
        SI.getMemory().at( k: address) = (el == f);
    }
}
```

图 3-1-17 DSE_Label 桩函数代码实现

```
extern "C" void __DSE_Loop__(int Type){
    expr type = SI.getContext().int_val( n: Type);
    SI.getStack().push( x: type);
}
```

图 3-1-18 DSE_Loop 桩函数代码实现

```
extern "C" void __DSE_Branch__(int RID, int BID) {
  if(SI.getStack().top().to_string() == "0"){
      SI.getStack().pop();
      return;
  }
  MemoryTy &Mem = SI.getMemory();
  Address Addr( ID: RID);
  /// (Addr(RID), SE)
  z3::expr SE = Mem.at( k: Addr);
  SI.getPathCondition().push_back(std::make_pair( x: BID,  y: SE));
}
```

图 3-1-19 DSE_Branch 桩函数代码实现

2. DFS 回溯

插桩是实验的准备工作，完成之后当迭代次数为 1 时可以求解，但是此时是无法完成多次迭代的，这时候需要通过 DFS 或者 BFS 算法来实现求解，这里我选择的是 DFS 回溯。

定义新方法 recordBranches 用于保存已经走过的路径，用于后面回溯时判断当前节点的真假分支是否都已走到。

```
void recordBranches(z3::expr_vector &OldVec){
    for (auto && E :expr&& : OldVec) {
        if(E.to_string().find( c: 'X') == std::string::npos){
            continue;
        }
        if(std::find( first: Branches.begin(), last: Branches.end(), val: E.to_string()) == Branches.end()){
            Branches.push_back(E.to_string().c_str());
        }
    }
}
```

图 3-2-1 recordBranches 方法代码实现

在 searchStrategy 中每次对当前路径的最后一个单元条件进行判断。首先如果容器中没有路径直接返回，如果最后一个条件判断中不含符号，删除此不含符号的表达式，进行下一次 search。接着，如果最后一次条件的另一个分支没有执行到，将表达式取反进行下一次迭代；如果已经执行过了，那么删除该节点并进行回溯。最后，当取反进行迭代时，可能会出现 unsat 的情形，这时候如果不处理，那么会陷入死循环，于是，如果最后一条判断语句如果是 not 开头，那么意味着 unsat，这时候删除此语句，进行下一次 search。

```
void searchStrategy(z3::expr_vector &OldVec) {
    int i = OldVec.size()-1;
    if(i<0){
        return;
    }
    auto e :expr = OldVec[i];
    if(e.to_string().find( c: 'X') == std::string::npos){
        OldVec.pop_back();
        searchStrategy( &: OldVec);
        return;
    }
```

图 3-2-2 先决条件判断

```
if(e.to_string().substr( pos: 1, n: 3) != NOT){
    std::string newExpr = e.to_string();
    if(newExpr.find( s: "false") != std::string::npos){
        newExpr = newExpr.replace( i1: newExpr.end()-6, i2: newExpr.end()-1, s: "true");
    } else{
        newExpr = newExpr.replace( i1: newExpr.end()-5, i2: newExpr.end()-1, s: "false");
    }
    if(std::find( first: Branches.begin(), last: Branches.end(), val: newExpr) == Branches.end()){
        OldVec.pop_back();
        OldVec.push_back( e: not e);
    } else{
        OldVec.pop_back();
        searchStrategy( &: OldVec);
    }
}
else {
    OldVec.pop_back();
    searchStrategy( &: OldVec);
}
```

图 3-2-3 回溯部分核心代码

3.2. 编译

```
mkdir build && cd build
cmake -DZ3_DIR=/usr/local/ -DLLVM_DIR=/usr/local/ ..
make
cd ../test
make clean && make && touch formula.smt2 && touch input.txt
export LD_LIBRARY_PATH=../build/
../build/dse ./simple0 1
```

4. 实验结果与分析

4.1. 实验结果

1. *simple tests*

a) *simple*0

```
../build/dse ./simple0 1
```

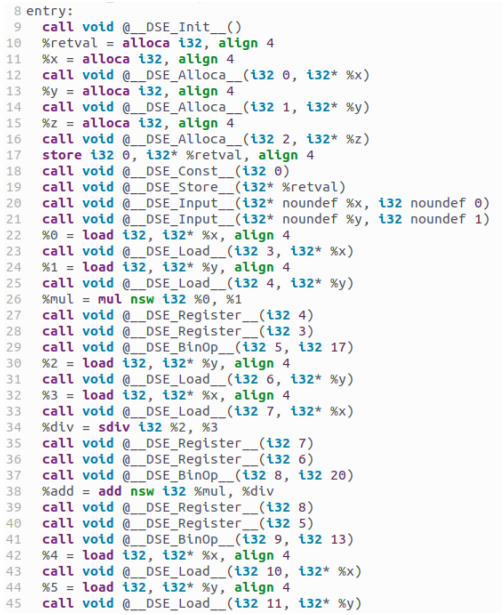
```
define dso_local i32 @main() #0 {
entry:
  call void @__DSE_Init__()
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  call void @__DSE_Alloca__(i32 0, i32* %x)
  %y = alloca i32, align 4
  call void @__DSE_Alloca__(i32 1, i32* %y)
  %z = alloca i32, align 4
  call void @__DSE_Alloca__(i32 2, i32* %z)
  store i32 0, i32* %retval, align 4
  call void @__DSE_Const__(i32 0)
  call void @__DSE_Store__(i32* %retval)
  store i32 1, i32* %x, align 4
  call void @__DSE_Const__(i32 1)
  call void @__DSE_Store__(i32* %x)
  %0 = load i32, i32* %x, align 4
  call void @__DSE_Load__(i32 3, i32* %x)
  store i32 %0, i32* %y, align 4
  call void @__DSE_Register__(i32 3)
  call void @__DSE_Store__(i32* %y)
  store i32 9, i32* %z, align 4
  call void @__DSE_Const__(i32 9)
  call void @__DSE_Store__(i32* %z)
  ret i32 0
}
```

```
1  === Inputs ===
2
3  === Symbolic Memory ===
4  140723167503632 : 9
5  140723167503636 : 1
6  140723167503640 : 1
7  140723167503644 : 0
8  R0 : (- 925938963)
9  R1 : 32764
10 R2 : (- 1435949744)
11 R3 : 1
12
13 === Path Condition ===
14
15 === New Path Condition ===
```

图 4-1-1 Simple0 实验结果

b) *simple*1

```
../build/dse ./simple1 1
```



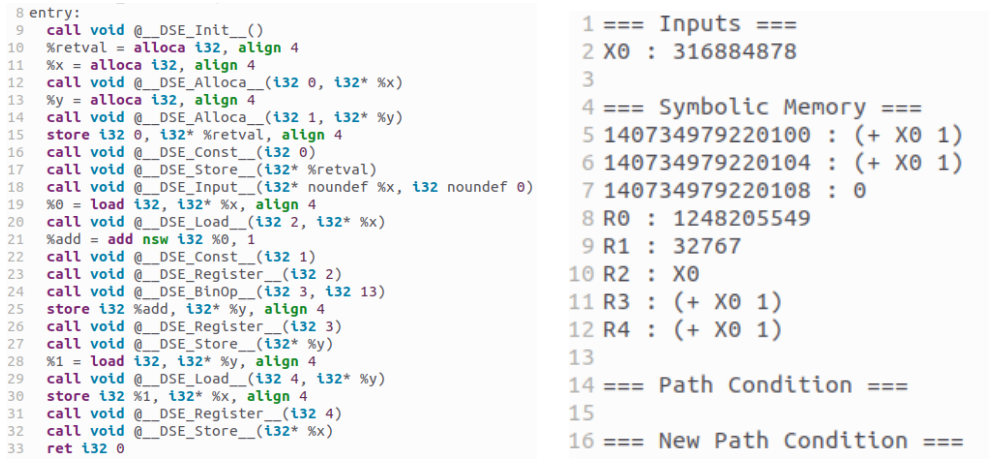
```
entry:
  call void @__DSE_Init__()
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  call void @__DSE_Alloca__(i32 0, i32* %x)
  %y = alloca i32, align 4
  call void @__DSE_Alloca__(i32 1, i32* %y)
  %z = alloca i32, align 4
  call void @__DSE_Alloca__(i32 2, i32* %z)
  store i32 0, i32* %retval, align 4
  call void @__DSE_Const__(i32 0)
  call void @__DSE_Store__(i32* %retval)
  call void @__DSE_Input__(i32* noundef %x, i32 noundef 0)
  call void @__DSE_Input__(i32* noundef %y, i32 noundef 1)
  %0 = load i32, i32* %x, align 4
  call void @__DSE_Load__(i32 3, i32* %x)
  %1 = load i32, i32* %y, align 4
  call void @__DSE_Load__(i32 4, i32* %y)
  %mul = mul nsw i32 %0, %1
  call void @__DSE_Register__(i32 4)
  call void @__DSE_Register__(i32 3)
  call void @__DSE_BinOp__(i32 5, i32 17)
  %2 = load i32, i32* %y, align 4
  call void @__DSE_Load__(i32 6, i32* %y)
  %3 = load i32, i32* %x, align 4
  call void @__DSE_Load__(i32 7, i32* %x)
  %div = sdiv i32 %2, %3
  call void @__DSE_Register__(i32 7)
  call void @__DSE_Register__(i32 6)
  call void @__DSE_BinOp__(i32 8, i32 20)
  %add = add nsw i32 %mul, %div
  call void @__DSE_Register__(i32 8)
  call void @__DSE_Register__(i32 5)
  call void @__DSE_BinOp__(i32 9, i32 13)
  %4 = load i32, i32* %x, align 4
  call void @__DSE_Load__(i32 10, i32* %x)
  %5 = load i32, i32* %y, align 4
  call void @__DSE_Load__(i32 11, i32* %y)
```

```
=== Inputs ===
X0 : 375028757
X1 : 1851776166

=== Symbolic Memory ===
140728793169552 : (- (+ (* X0 X1) (div X1 X0)) (+ X0 X1))
140728793169556 : X1
140728793169560 : (- (+ (* X0 X1) (div X1 X0)) (+ X0 X1))
140728793169564 : 0
R0 : 1163762413
R1 : 32765
R2 : (- 105251120)
R3 : X0
R4 : X1
R5 : (* X0 X1)
R6 : X1
R7 : X0
R8 : (div X1 X0)
R9 : (+ (* X0 X1) (div X1 X0))
R10 : X0
R11 : X1
R12 : (+ X0 X1)
R13 : (- (+ (* X0 X1) (div X1 X0)) (+ X0 X1))
R14 : (- (+ (* X0 X1) (div X1 X0)) (+ X0 X1))

=== Path Condition ===

=== New Path Condition ===
```

图 4-1-2 Simple1 实验结果

c) *simple2*

../build/dse ./simple2 1



```
entry:
  call void @__DSE_Init__()
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  call void @__DSE_Alloca__(i32 0, i32* %x)
  %y = alloca i32, align 4
  call void @__DSE_Alloca__(i32 1, i32* %y)
  store i32 0, i32* %retval, align 4
  call void @__DSE_Const__(i32 0)
  call void @__DSE_Store__(i32* %retval)
  call void @__DSE_Input__(i32* noundef %x, i32 noundef 0)
  %0 = load i32, i32* %x, align 4
  call void @__DSE_Load__(i32 2, i32* %x)
  %add = add nsw i32 %0, 1
  call void @__DSE_Const__(i32 1)
  call void @__DSE_Register__(i32 2)
  call void @__DSE_BinOp__(i32 3, i32 13)
  store i32 %add, i32* %y, align 4
  call void @__DSE_Register__(i32 3)
  call void @__DSE_Store__(i32* %y)
  %1 = load i32, i32* %y, align 4
  call void @__DSE_Load__(i32 4, i32* %y)
  store i32 %1, i32* %x, align 4
  call void @__DSE_Register__(i32 4)
  call void @__DSE_Store__(i32* %x)
  ret i32 0
```

```
=== Inputs ===
X0 : 316884878

=== Symbolic Memory ===
140734979220100 : (+ X0 1)
140734979220104 : (+ X0 1)
140734979220108 : 0
R0 : 1248205549
R1 : 32767
R2 : X0
R3 : (+ X0 1)
R4 : (+ X0 1)

=== Path Condition ===

=== New Path Condition ===
```

图 4-1-3 Simple2 实验结果

2. *branch tests*

a) *branch0*

../build/dse ./branch0 5

```
jackyma@cyber:~/Study/Codes/C/dse/test$ ../build/dse ./branch0 5
Floating point exception (core dumped)
Crashing input found (3 iters)
```

图 4-1-4 Branch0 发现除 0 错

input.txt

```
X2,0
X1,0
X0,1
```

图 4-1-5 Branch0 最终输入

```c
#include <stdio.h>

#include "../include/Runtime.h"

int main() {
  int x;
  DSE_Input(x);
  int y;
  DSE_Input(y);
  int z;
  DSE_Input(z);

  if (x == 1) {
  //if(x + y == x - z) {
        if (y == z) {
              x = x / (y-z);
        }
  }
  else {
        if(x == 0){
              x = 0;
        }
  }

  return 0;
```

```
=== Inputs ===
X0 : 1
X1 : 1648516831
X2 : 1189478266

=== Symbolic Memory ===
140724173342480 : X2
140724173342484 : X1
140724173342488 : X0
140724173342492 : 0
R0 : (- 437724435)
R1 : 32764
R2 : (- 430110896)
R3 : X0
R4 : (= X0 1)
R5 : (= (= X0 1) true)
R6 : (= (= X0 1) false)
R7 : X1
R8 : X2
R9 : (= X1 X2)
R10 : (= (= X1 X2) true)
R11 : (= (= X1 X2) false)

=== Path Condition ===
B0 : (= (= X0 1) true)
B2 : (= (= X1 X2) false)

=== New Path Condition ===
(= (= X0 1) true)
(not (= (= X1 X2) false))
```

图 4-1-6 Branch0 实验结果

b) *branch1*

../build/dse ./branch1 5

```
jackyma@cyber:~/Study/Codes/C/dse/test$ ../build/dse ./branch1 5
Floating point exception (core dumped)
Crashing input found (2 iters)
```

图 4-1-7 Branch1 发现除 0 错

input.txt

```
X2,0
X0,0
X1,0
```

图 4-1-8 Branch1 最终输入

```c
#include <stdio.h>

#include "../include/Runtime.h"

int main() {
  int x;
  DSE_Input(x);
  int y;
  DSE_Input(y);
  int z;
  DSE_Input(z);

  if (x == y && y == z) {
        x = x / (y-z);
  }
  else{
        x = 1;
  }

  return 0;
}
```

```
=== Inputs ===
X0 : 0
X1 : 0
X2 : 1927440769

=== Symbolic Memory ===
140727125036624 : X2
140727125036628 : X1
140727125036632 : 1
140727125036636 : 0
R0 : 1579465453
R1 : 32765
R2 : (- 1773384048)
R3 : X0
R4 : X1
R5 : (= X0 X1)
R6 : (= (= X0 X1) true)
R7 : (= (= X1 X2) false)
R8 : X1
R9 : X2
R10 : (= X1 X2)
R11 : (= (= X1 X2) true)

=== Path Condition ===
B0 : (= (= X0 X1) true)
B2 : (= (= X1 X2) false)

=== New Path Condition ===
(= (= X0 X1) true)
(not (= (= X1 X2) false))
```

图 4-1-9 Branch1 实验结果

c) *branch2*

../build/dse ./branch2 5

```
jackyma@cyber:~/Study/Codes/C/dse/test$ ../build/dse ./branch2 5
Floating point exception (core dumped)
Crashing input found (2 iters)
```

图 4-1-10 Branch2 发现除 0 错

input.txt

```
X1,1
X0,2
X2,0
```

图 4-1-11 Branch2 最终输入

```c
#include <stdio.h>

#include "../include/Runtime.h"

int main() {
  int x;
  DSE_Input(x);
  int y;
  DSE_Input(y);
  int z;
  DSE_Input(z);


  if (x > y && y > z && z == 0) {
        x = x / z;
  }

  return 0;
}
```

```
=== Inputs ===
X0 : 1
X1 : 0
X2 : -1

=== Symbolic Memory ===
140725332437920 : X2
140725332437924 : X1
140725332437928 : X0
140725332437932 : 0
R0 : (- 2083398931)
R1 : 32765
R2 : 728984544
R3 : X0
R4 : X1
R5 : (> X0 X1)
R6 : (= (> X0 X1) true)
R7 : (= (= X2 0) false)
R8 : X1
R9 : X2
R10 : (> X1 X2)
R11 : (= (> X1 X2) true)
R12 : X2
R13 : (= X2 0)
R14 : (= (= X2 0) true)

=== Path Condition ===
B0 : (= (> X0 X1) true)
B1 : (= (> X1 X2) true)
B3 : (= (= X2 0) false)

=== New Path Condition ===
(= (> X0 X1) true)
(= (> X1 X2) true)
(not (= (= X2 0) false))
```

图 4-1-12 Branch2 实验结果

3. *infeasable*

../build/dse ./infeasible 10

初始时在 input.txt 中输入 x0,2

```
jackyma@cyber:~/Study/Codes/C/dse/test$ ../build/dse ./infeasable 10
Floating point exception (core dumped)
Crashing input found (8 iters)
```

图 4-1-13 Infeasable 发现除 0 错

input.txt

```
X0,10
```

图 4-1-14 Infeasable 最终输入

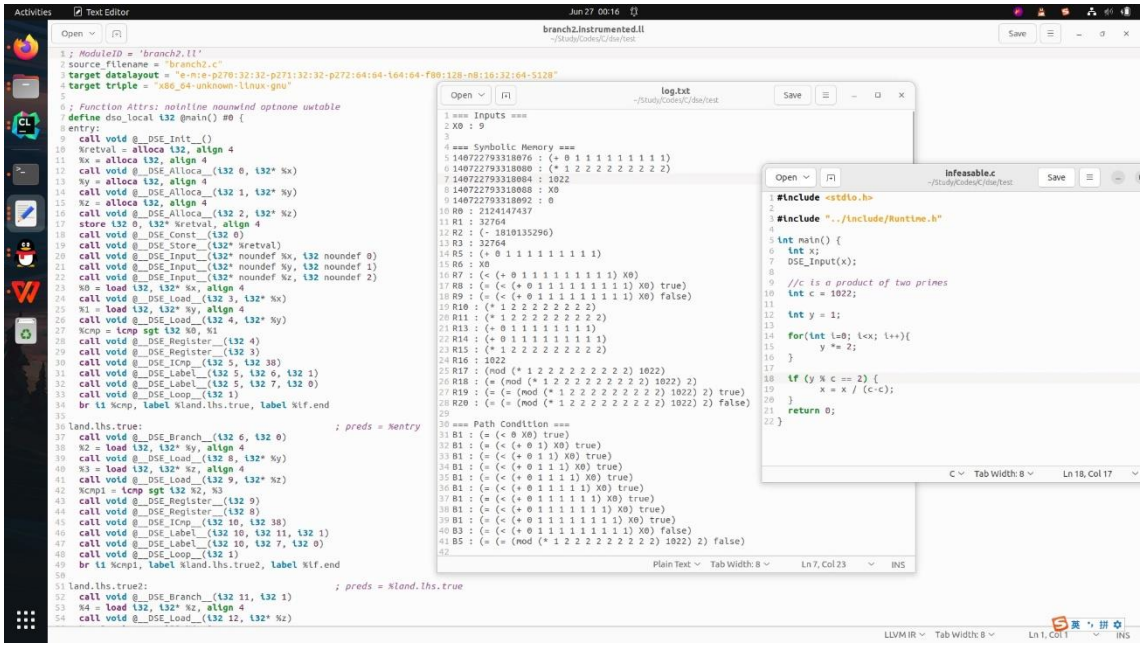


图 4-1-15 Infeasable 实验结果

4.2. 实验分析

在所给的测试用例中没有涉及到回溯，即是说每一次只需要对路径的最后一个条件语句进行取反即可通过测试用例，无需向父节点回溯。这样的测试强度肯定是不够的，于是我自己编写一小段代码用来检测回溯的效果。

```c
#include <stdio.h>

#include "../include/Runtime.h"

int main() {
  int x;
  DSE_Input(x);
  int y;
  DSE_Input(y);
  int z;
  DSE_Input(z);

  if (x == 1) {
        if (y == z) {
              x = x / (y-z);
        }
  } else {
       if(x == 0){
              x = 1;
       }
  }

  return 0;
}
```

图 4-2-1 回溯测试代码

初始时在 input.txt 中输入 X0,-1。第 0 此迭代的路径为 X0 != 1∩X0 != 0；第 1 此迭代路径为 X0 != 1∩X0 == 0；第 2 次迭代的路径为 X0 == 1∩X1 != X2；第 3 次迭代的路径为 X0 == 1∩X1 == X2，发生除 0 错，检测错误。运行得到的结果也是预期的。

```
jackyma@cyber:~/Study/Codes/C/dse/test$ ../build/dse ./branch0 5
Floating point exception (core dumped)
Crashing input found (3 iters)
```

图 4-2-2 回溯测试结果

另外，在 for 循环的探索中有一个较为严重的缺陷，就是对于 x 的探索，只能顺着判断条件递增，因为每一次回溯时，只对最后一个条件取反，前面的条件都要满足，这样就意味着 x'只能比 x 大。也会导致很多原本应该探测到的路径却没有探测到，针对回溯的搜索策略，只能满足于测试用例。

5. 总结与致谢

通过本次实验，提升了 C++编程能力，在代码编写过程中学习到了栈与容器的使用，加深了对 map 的运用。另外实验的框架基于 llvm 和 z3 求解器，在熟悉其 api 的过程中也使我掌握了许多方法，当遇到困难时，首先想到的应该是官方文档，在里面可以解决 90%的问题，剩下的 10%则是通过自己不断地测试摸索得到。

另外，在算法上，我学习到了 DFS 深度搜索，如何在二叉树中进行回溯是我对算法的理解更进一步，总之，通过本次实验，丰富了我的编程经历，提升了我个人能力，增强了信心，在未来将要面对的各种困难面前，我将会更加游刃有余。

补充一点，代码框架中，对于 z3 求解有一个逻辑上的错误，但由于测试用例没有涉及到回溯，因此并无影响。

```
Solver.reset();
/// Vec 格式:
for (const auto &E :const expr& : Vec) {
  Solver.add( e: E);
}
```

图 5-1 z3 求解相关代码缺陷

如果发生回溯，不加 Solver.reset()，那么每一次求解都是 unsat，因为分支的真假条件都在 Solver 中。

在实验的过程中有许多不懂的地方，感谢教员与教辅的耐心指导，使我在面对困难时有了更多的信心，最终成功完成实验。也感谢我身边的同学，当我有疑惑时，总是热情第帮助我分析问题并给予解决方案，没有同学的帮助也不会有现在的成果。