Report of CourseWork3: z3 SMT Solver & Basic Program Verification

20163660 Junmin Choe

June 2, 2016

1 GETTING FAMILIAR WITH Z3 SMT SOLVER(PUZZLE)

1.1 DEVELOPMENT ENVIRONMENT

- OS - *OS X 10.11.5(El Capitan)*
- Shell - *GNU bash, 3.2.57 (x86_64-apple-darwin15)*
- Language - *SMT-LIB 2.0 standard*
- Tools - *z3 SMT Solver*

You can also run this program in ubuntu or other linux environment.

1.2 HOW TO APPROACH THE PROBLEM

According to description of coursework, we can list three problems which we should solve.

1. Write a z3 formula that can answer question on condition(pick apple on crate labeled "Mixed").
2. Prove that there is only one possible assignment for the labels given the information.
3. Would the information (pick apple on crate labeled "Orange") had been enough to uniquely determine the solution?

Let me explain detail approaches of each problem.

PROBLEM 1 There is another condition to write z3 formula. It can be found in description:

> ...involving the variables realApple, realOrange and realMixed, such that the variable realOrange will tell you the real contents of the crate labeled "orange", the variable realApple will tell you the real contents of the crate labeled "apple", and similarly for realMixed. The values of these three variables should be consistent with the fact that you picked an apple from the mixed crate.

and there is a hint.

> **Hint:** consult the enumeration type(Scalas) in the z3 SMT Solver tutorial (`http://rise4fun.com/z3/tutorial`).

So, i thought that there needs to declare those variables with the enumeration type(Scalas). And last, i will give the formula which is corresponding to our assumption(picking condition). The formula is easily obtained as follows:

(realMixed == Apples) or (realMixed == Mixes)

The whole process can be implemented with syntax of z3(SMT).

PROBLEM 2 The task of this problem is to prove that there is only one assignment on problem1. It can be also easily done with adding additional assertion which is not allow answer assignment of problem1. After adding the assertion, confirm the satisfiable of formula. If final result is 'unsat', we can conclude that there is only one assignment. Otherwise, there is not.

PROBLEM 3 First, the picking condition changed in this problem. So, instead of previous picking assertion(formula) i provided new formula:

(realOrange == Apples) or (realOrange == Mixes)

Using this formula, we can get one solution. Our final job is to determine uniqueness of the solution. Having restriction of previous solution (same as problem2), we can prove this uniqueness.

1.3 IMPLEMENTATION

PROBLEM1 In first step, i declared variables and 'Scalas'. The detail code is:

```
 (declare-datatypes () ((Content Apples Oranges Mixes)))
(declare-const realApple Content)
(declare-const realOrange Content)
(declare-const realMixed Content)
```

and we have the basic assertions which are:

```
(assert (distinct realApple realOrange realMixed)) ; contents of each crates are all distinct
(assert (not (= realApple Apples)))
(assert (not (= realOrange Oranges)))
(assert (not (= realMixed Mixes))) ; all the labels have been shuffled
```

These should be obviously included in three problems. Now, i explain remains for each problem.

PROBLEM 1 Problem1 needs picking condition. This can be like:

```
(assert (or (= realMixed Apples) (= realMixed Mixes))) ; pick an apple from the mixed crate
```

And satisfiability check and print solution(assignment) can be done with 'check-sat' and 'get-model' in z3 script.

PROBLEM 2 Implementation of the assertion i described in 'apporach' section is as follows:

```
(assert (not (and (and
        (= realApple Oranges)
        (= realMixed Apples))
        (= realOrange Mixes))
)) ; the assignment of problem1 is not allowed
```

The given state of crates is given from problem1 result. I just added this and satisfiable check code in this problem. The result will give solution of the problem to us.

PROBLEM 3 As mentioned earlier, this problem is like mixture of problem 1 and 2. Change picking condition and test uniqueness of solution like problem2. The real code is shown below square box.

```
(assert (or (= realOrange Apples) (= realOrange Mixes))) ; pick an apple from the "orange" crate
(check-sat)
(get-model)
(assert (not (and (and
        (= realApple Oranges)
        (= realMixed Apples))
        (= realOrange Mixes))
))
(check-sat)
(get-model)
```

1.4 TEST AND RESULTS

All codes exist in 'puzzle' folder. Problem1 code named as "prob1.z3", problem2 as "prob2.z3" and last problem as "prob3.z3". You can execute these anytime with 'z3' command. The results of each problem are as shown:

1. sat
   ```
   (model
     (define-fun realApple () Content
       Oranges)
     (define-fun realMixed () Content
       Apples)
     (define-fun realOrange () Content
       Mixes)
   )
   ```
2. unsat
3. sat
   ```
   (model
     (define-fun realApple () Content
       Oranges)
     (define-fun realMixed () Content
       Apples)
     (define-fun realOrange () Content
       Mixes)
   )
   sat
   (model
     (define-fun realApple () Content
       Mixes)
     (define-fun realMixed () Content
       Oranges)
     (define-fun realOrange () Content
       Apples) )
   ```

PROBLEM 1 As a result, content of the crate labeled "Apple" are oranges, "Mixed" are apples and "Orange" are mixes. This is credible result, so we can conclude that the formula which is in 'Approach' section works well.

PROBLEM 2 The result is 'unsat'. Hence, statement "there is only one possible assignment for ..." is proven.

PROBLEM 3 The first model of result is one solution for given condition and second model also. Therefore, there are at least two solutions for given picking restriction, so the uniqueness is denied.

2 PROGRAM VERIFICATION(VERIFIER)

2.1 DEVELOPMENT ENVIRONMENT

- OS - *OS X 10.11.5(El Capitan)*
- Shell - *GNU bash, 3.2.57 (x86_64-apple-darwin15)*
- Language - *Python 2.7.11*
- Tools - *javac(version-7), soot, z3 SMT Solver*

You can also run this program in ubuntu or other linux environment.

2.2 HOW TO APPROACH THE PROBLEM

I summarized whole solving process step by step. The steps are as follows:

1. Read java file and transform to jimple format
2. Read jimple file and make z3 script that check whether assertion is valid
3. Run z3 script and make output

There is a special thing we should take care of. In second task on this problem, the second and third steps are combined. This is because there exist many program flows to make assertion invalid in case of second task(but only one in first task) and we must consider all these flows whether it generates exception. To deal with them, i used **recursion** method which is also called **backtracking**. Using this method, we can get and see all flows. And next job is making z3 script, running it and checking model simultaneously whenever it reaches exit condition of recursion. This causes big running time but i have no any different idea... Fortunately, in my testcases the program consumes a little time to generate z3 script, so i thought it has no problem unless test file have the huge number of branches. And now, let me explain details of each summarized step.

TRANSFORM JAVA TO JIMPLE First, we compile java using *javac* which makes class file. The result class file will be input for *soot*. Soot is a tool for analyzing, optimizing and visualizing Java and Android applications. This given material simply outputs *jimple* file and we can just use it. These whole process can be done with basic python command.

TRANSFORM JIMPLE TO Z3 AND RUN Z3 Parsing is first task we need. Jimple has simple instruction format, so we can easily parse that line by line. In each line, we classify each instruction and execute them (but not real 'execute', for example '*a* = *b* + *1*' means in future just substitute variable '*a*' with '*b* + *1*'). We make z3 assertion when meets *throw* statement because in jimple, *throw* statement stands for 'assertion invalid'. So in summary, we read jimple file, execute all possible program flows(as mentioned earlier, there are many possible flows that meets *throw* statement in second task, but only one in first task) and print invalid assertion model when we get 'sat' on running z3. (also exit program if print model)

MAKE OUTPUT During previous step(run z3), the invalid assertion model(assignment) must be found if it exists. Hence, if recursive function ends up and flow gets here, the assertion in java file is valid! So just print 'VALID' and we are done.

2.3 IMPLEMENTATION

The python source code of verifier is in 'verifier' folder. In this section, i will just describe pseudocodes of each step and provide explanation a bit.

TRANSFORM JAVA TO JIMPLE See below pseudocode.

Algorithm 1: GenerateJimple

```
Data: Java file
Result: Jimple file
begin
    if file name argument is not valid then
        return;
    end
    change directory to 'soot' and copy new java code;
    remove previous java, class, z3 files named same;
    compile the java source code and output class file;
    execute soot and make jimple file;
end
```

Soot is in *'verifer/soot'* folder, it needs directory change command. In python, bash command can be offered by *'os.system'* api.

TRANSFORM JIMPLE TO Z3 AND RUN Z3 See below pseudocode.

Algorithm 2: GenerateZ3andRun

```
Data: Jimple file, current label index
Result: print model if founded, otherwise none
begin
    for Instruction in Jimples[CurrentIndex] do
        if THROW then
            write stacked assertion to z3 file;
            Run Z3;
            if result is SAT then
                Get model of SAT and Print;
                Exit program;                           /* exit condition */
            end
            return ;
        else if GOTO then
            GenerateZ3andRun(NextIndex)
        else if ASSIGN then
            decode instruction;
            update variables;
        else if IF then
            decode instruction;
            save current state and update assert condition;
            GenerateZ3andRun(NextBranchIndex) ;
            restore state;
            continue ;
        else
            continue ;
        end
    end
    GenerateZ3andRun(CurrentIndex+1)
end
```

Variable *'Inst'* denotes instructions and *'Jimples'* stores all meaningful instructions(add, subtract, if, throw, ...) indexed with label number. For example, *Jimples[2]* stores meaningful instructions in label2. Since jimple number labels in sequence(1, 2, ...), i could use this weird indexed data structure.

2.4 Test and Results

Using command "python verifier.py [java-file-path]", you can execute this program anytime for test. I tested with 4 testcases which are given from teaching assistant. The results of each case are as follows:

```
1. (model
     (define-fun l3 () Int
       (- 6))
     (define-fun l2 () Int
       (- 7))
     (define-fun l1 () Int
       (- 8))
   )

2. (model
   )

3. VALID

4. (model
     (define-fun l3 () Int
       1)
     (define-fun l1 () Int
       1)
     (define-fun l2 () Int
       1)
   )
```

Let me explain results. If program judges assertion is valid the result is 'VALID', otherwise model of assertion. In model, variable named 'l1' stands for '1'st parameter. So 'l3' stands for third parameter of test function. If certain variable does not print assignment, it means that the assertion at this assignment is always invalid regardless of its value.

I confirmed this results are all correct. You can test more complicated cases. But it can take more bigger time to generate results and be killed by OS since the method is **exhaustive search**. So, i am willing to try to make time-efficient solver in future.