Test strategy - Version 1

Sofia Boutahar
David Courtinot

Reference : model-checking.test-strategy
February 16th, 2015

Signatures

Project manager - Zohour Abouakil :
Quality responsible - David Courtinot :
Customers - David Doose - Julien Brunel :

Contents

I Overview **2**

II Testing AST conversion to CFG **3**
II.1 Test strategy 3
II.2 Testing the parser 3
II.3 Testing results 5

III Testing the model checking algorithm **6**
III.1 Automated testing 6
III.2 Testing process 6
III.3 Results 7

Part I

Overview

The purpose of a test strategy is to clarify the major tasks and challenges of the test project, define what we want to accomplish and how we are going to achieve it. Moreover, it helps us figure out if there are missing requirements in the project and have a clear state of it at any point. Our project is divided into two parts: The conversion of the AST to the CFG and the model checking using CTL. The first part can be divided into two subparts : Parsing the file generated by the Clang compiler and converting the AST to the CFG.

Part II

Testing AST conversion to CFG

II.1 Test strategy

When we identify a particular problem in our code, we try to debug it and the fix the issues. To make sure that the fix works, we test our program again to see if it works. Therefore, it is not sufficient to validate the test if the tested program works, we should make sure that our fixes don't create some other problems in any other parts of our projects due to dependencies. So, a set of related test cases may have to be repeated again, to make sure that nothing else is affected our fixes. Basically, whenever there is a fix in one unit, we repeat all unit test cases for that unit in order to achieve a higher level of abstraction and quality.

II.2 Testing the parser

The clang compiler takes a C++ source code as input and returns a file describing an Abstract Syntax Tree. The purpose of creating a parser is to extract from the Clang file all the useful information and build a tree that contains it.

As the number of possible tests for our parser is practically infinite, we created several tests starting from the easiest to the most complicated one. The aim was to ensure that our program is working for the minimum number of tests needed to get the coverage we want.

```
TranslationUnitDecl 0x1028254d0 <<invalid sloc>>
|-TypedefDecl 0x102825a10 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x102825a70 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x102825e30 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x102825ed0 <workspace/Projet-merou/ModelChecker/unitary_tests/while/doWhile.cpp:1:1, line:11:1> main 'int (void)'
  `-CompoundStmt 0x102871fb0 <line:1:12, line:11:1>
    |-DeclStmt 0x102826058 <line:2:2, col:11>
    | `-VarDecl 0x102825fe0 <col:2, col:10> a 'int'
    |   `-IntegerLiteral 0x102826038 <col:10> 'int' 5
    |-DoStmt 0x102871ee8 <line:3:2, line:9:16>
    | |-CompoundStmt 0x102871e40 <line:3:5, line:9:2>
    | | `-IfStmt 0x102871e10 <line:4:3, line:8:3>
    | |   |-<<<NULL>>>
    | |   |-BinaryOperator 0x1028260d0 <line:4:7, col:11> '_Bool' '>'
    | |   | |-ImplicitCastExpr 0x1028260b8 <col:7> 'int' <LValueToRValue>
    | |   | | `-DeclRefExpr 0x102826070 <col:7> 'int' lvalue Var 0x102825fe0 'a' 'int'
    | |   | `-IntegerLiteral 0x102826098 <col:11> 'int' 6
    | |   |-BinaryOperator 0x102826140 <line:5:4, col:8> 'int' lvalue '='
    | |   | |-DeclRefExpr 0x1028260f8 <col:4> 'int' lvalue Var 0x102825fe0 'a' 'int'
    | |   | `-IntegerLiteral 0x102826120 <col:8> 'int' 5
    | |   `-CompoundStmt 0x1028261d8 <line:6:8, line:8:3>
    | |     `-BinaryOperator 0x1028261b0 <line:7:4, col:8> 'int' lvalue '='
    | |       |-DeclRefExpr 0x102826168 <col:4> 'int' lvalue Var 0x102825fe0 'a' 'int'
    | |       `-IntegerLiteral 0x102826190 <col:8> 'int' 3
    | `-BinaryOperator 0x102871ec0 <line:9:11, col:15> '_Bool' '>'
    |   |-ImplicitCastExpr 0x102871ea8 <col:11> 'int' <LValueToRValue>
    |   | `-DeclRefExpr 0x102871e60 <col:11> 'int' lvalue Var 0x102825fe0 'a' 'int'
    |   `-IntegerLiteral 0x102871e88 <col:15> 'int' 5
    `-DeclStmt 0x102871f98 <line:10:2, col:12>
      `-VarDecl 0x102871f20 <col:2, col:10> b 'int'
        `-IntegerLiteral 0x102871f78 <col:10> 'int' 17
```

Figure II.1 - Example of an input AST (generated automatically thanks to Clang by our test program) of a do-while statement

```
ModelChecker/unitary_tests/while/doWhile.txt
Labels :
OtherASTNode(-1,)
  ConcreteASTNode(1,FunctionDecl,0x10282b8d0,(11,1:1),FunctionDecl 0x10282b8d0 <ModelChecker/unitary_tests/while/doWhile.cpp:1:1, line:11:1> mai
    ConcreteASTNode(2,CompoundStmt,0x103036fb0,(1:11,12:1),CompoundStmt 0x103036fb0 <line:1:12, line:11:1>)
      ConcreteASTNode(3,DeclStmt,0x10282ba58,(2,2:11),DeclStmt 0x10282ba58 <line:2:2, col:11>)
        ConcreteASTNode(4,VarDecl,0x10282b9e0,(2,2:10),VarDecl 0x10282b9e0 <col:2, col:10> a 'int')
          ConcreteASTNode(5,IntegerLiteral,0x10282ba38,(2,10:10),IntegerLiteral 0x10282ba38 <col:10> 'int' 5)
      ConcreteASTNode(3,DoStmt,0x103036ee8,(3:9,2:16),DoStmt 0x103036ee8 <line:3:2, line:9:16>)
        ConcreteASTNode(4,CompoundStmt,0x103036e40,(3:9,5:2),CompoundStmt 0x103036e40 <line:3:5, line:9:2>)
          ConcreteASTNode(5,IfStmt,0x103036e10,(4:8,3:3),IfStmt 0x103036e10 <line:4:3, line:8:3>)
            NullASTNode(6)
            ConcreteASTNode(6,BinaryOperator,0x10282bad0,(4,7:11),BinaryOperator 0x10282bad0 <line:4:7, col:11> '_Bool' '>')
              ConcreteASTNode(7,ImplicitCastExpr,0x10282bab8,(4,7:7),ImplicitCastExpr 0x10282bab8 <col:7> 'int' <LValueToRValue>)
                ConcreteASTNode(8,DeclRefExpr,0x10282ba70,(4,7:7),DeclRefExpr 0x10282ba70 <col:7> 'int' lvalue Var 0x10282b9e0 'a' 'int')
              ConcreteASTNode(7,IntegerLiteral,0x10282ba98,(4,11:11),IntegerLiteral 0x10282ba98 <col:11> 'int' 6)
            ConcreteASTNode(6,BinaryOperator,0x10282bb40,(5,4:8),BinaryOperator 0x10282bb40 <line:5:4, col:8> 'int' lvalue '=')
              ConcreteASTNode(7,DeclRefExpr,0x10282baf8,(5,4:4),DeclRefExpr 0x10282baf8 <col:4> 'int' lvalue Var 0x10282b9e0 'a' 'int')
              ConcreteASTNode(7,IntegerLiteral,0x10282bb20,(5,8:8),IntegerLiteral 0x10282bb20 <col:8> 'int' 5)
            ConcreteASTNode(6,CompoundStmt,0x10282bbd8,(6:8,8:3),CompoundStmt 0x10282bbd8 <line:6:8, line:8:3>)
              ConcreteASTNode(7,BinaryOperator,0x10282bbb0,(7,4:8),BinaryOperator 0x10282bbb0 <line:7:4, col:8> 'int' lvalue '=')
                ConcreteASTNode(8,DeclRefExpr,0x10282bb68,(7,4:4),DeclRefExpr 0x10282bb68 <col:4> 'int' lvalue Var 0x10282b9e0 'a' 'int')
                ConcreteASTNode(8,IntegerLiteral,0x10282bb90,(7,8:8),IntegerLiteral 0x10282bb90 <col:8> 'int' 3)
        ConcreteASTNode(4,BinaryOperator,0x103036ec0,(9,11:15),BinaryOperator 0x103036ec0 <line:9:11, col:15> '_Bool' '>')
          ConcreteASTNode(5,ImplicitCastExpr,0x103036ea8,(9,11:11),ImplicitCastExpr 0x103036ea8 <col:11> 'int' <LValueToRValue>)
            ConcreteASTNode(6,DeclRefExpr,0x103036e60,(9,11:11),DeclRefExpr 0x103036e60 <col:11> 'int' lvalue Var 0x10282b9e0 'a' 'int')
          ConcreteASTNode(5,IntegerLiteral,0x103036e88,(9,15:15),IntegerLiteral 0x103036e88 <col:15> 'int' 5)
      ConcreteASTNode(3,DeclStmt,0x103036f98,(10,2:12),DeclStmt 0x103036f98 <line:10:2, col:12>)
        ConcreteASTNode(4,VarDecl,0x103036f20,(10,2:10),VarDecl 0x103036f20 <col:2, col:10> b 'int')
          ConcreteASTNode(5,IntegerLiteral,0x103036f78,(10,10:10),IntegerLiteral 0x103036f78 <col:10> 'int' 17)
```

Figure II.2 - Corresponding output (ASTNode tree data-structure), as the test program prints it

II.3 Testing results

We

Part III

Testing the model checking algorithm

Unlike the CFG part, there is a lot of room here for automated tests because there are many elementary operations that must return a specific result. We have written some generic methods that enable to automatically check the output rather than performing a manual check. This constitutes a very simple testing framework that we will present here. We will then present our process and results for the first iteration.

III.1 Automated testing

All the test functions rely on generic methods such as **assertEquals** or **assertTrue** that print the result of the test with its number. This way, it is fairly easy to generate a log file and get the result of the tests in the clearest way. You can find below an example of such test functions :

```
1 def printMsg(failed: Boolean) = {
2       val msg = "\tTest %d %s".format(i,if (failed) "failed" else "passed")
3       if (failed) Console.err.println(msg)
4       else println(msg)
5       i += 1
6 }
7 def assertEquals[T](t0: T, t1: T) = printMsg(t0 != t1)
8 def assertTrue(b: Boolean) = printMsg(!b)
9 def compareEnv[T](envT1: Env[T], envT2: Env[T], expected: Env[T]) = assertEquals(envT1 interEnv envT2,
    def testNeg[T](env: Env[T], envs: Env[T]*) = assertEquals(!env,Set(envs: _*))
```

III.2 Testing process

For this part, we have made a lot of unitary tests. We first tested every operation defined on the Environment subclasses because it constitutes the base of almost all computations performed by the algorithm. Once we were assured that these operations were correctly implemented, we tested the operations of the ModelChecker, starting with the elementary operations to the compound operations. We still have not tested the highest level functions.

III.3 Results

We detected and fixed some minor bugs in our elementary operations. Globally, all the functions were implemented correctly. Note that we had to rewrite those tests when we changed our conception for the CTL part. This was fairly easy thanks to the good structure of the code and we soon obtained similar results. Just as an example, here is an example of the log generated by the automated tests.

```
        Test 0 failed
Testing environments...
-----------------------
Testing compareEnv...
        Test 1 passed
        Test 2 passed
        Test 3 passed
        Test 4 passed

Testing the negation of an environment...
        Test 5 passed
        Test 6 passed
        Test 7 passed
        Test 8 passed

Testing '-' operation...
        Test 9 passed

Testing model checker...
------------------------
Testing shift...
        Test 10 passed
```

Figure III.1 - Example of a test log file