Clang Static Analyzer

Clang Static Analyzer is a bug-finding tool upon Clang (http://clang.llvm.org/) and LLVM (http://llvm.org/). It has a bunch of built-in checkers which statically analyze source code and reports bugs. Currently the default checkers are listed here (http://clang-analyzer.llvm.org/available_checks.html#default_checkers). Developers are also allowed to build their own checker.

In order to find bugs, this analyzer performs symbolic execution of the given program. Symbolic execution is a powerful program analysis technique that explores all possible pathes in a program. It doesn't actually run the program. Instead, it solves the constraints (e.g. if condition and loop condition) and binds the solution with symbolic values. In each possible path, clang static analyzer applies a bunch of checkers to evaluate all expressions and statements in that path to find bugs. Generally, each checker checks for a single kind of bug by analyzing the symbolic values of the expression.

Here is a simple example.

```
1    int calculator( char op, int left, int right){
2     int result;
3
4     if(op == '+'){
5         result = left + right;
6     }else if(op == '-'){
7         result = left - right;
8     }else if(op == '*'){
9         result = left * right;
10    }else if(op == '/'){
11        result = left / right; // here is a division-by-zero bug
12    }else{
13        exit(-1);
14    }
15
16    return result;
17 }
```

Let's take a look at line 11. For the purpose of presentation, the symbolic value for each variable or parameter starts with $. To reach this line, the constraint " op == '/' " must be satisfied. Symbolic execution solves this constraint and binds '/' with $op. Because left and right are function parameters passed by the caller, the analyzer doesn't know the concrete values in static analysis. So it simply binds these two parameters with $left and $right. The the value of variable result will be "$left / $right".

Every time the analyzer encounters a division expression, a specific checker is triggered to detect division-by-zero bugs. The checker examines the symbolic value of the divisor: if the divisor is binded with any integers, there is a division-by-zero bug.

Installation

Mac OS X

Pre-built binaries of Clang Static Analyzer are available on Mac OS X (10.5 or later). Since Xcode 3.2, users have been able to run the Clang Static Analyzer directly within Xcode. But you are always recommended to check out the latest build.

1. Download the latest build checker-276.tar.bz2 (http://clang-analyzer.llvm.org/downloads/checker-276.tar.bz2).

2. Unpack the package anywhere.

```
$  tar -jxvf checker-276.tar.bz2 -C /any/where/you/wish/
```

3. There are two tools, **scan-build** and **scan-view** at the top of the **checker-276** directory.

- **scan-build**: scan-build is the high-level command line utility for running the analyzer
- **scan-view**: scan-view is used to view analysis results generated by scan-build

4. To run scan-build, you are suggested to add the checker-276 directory to your path.

```
$ sudo nano /etc/paths
// add the checker-276 directory to the existing path
$ echo $PATH
```

Otherwise, you have to specify a complete path for scan-build in the command.

Ubuntu

Clang Static Analyzer can also be easily installed using Ubuntu Package Manager.

```
$ sudo add-apt-repository ppa:kxstudio-team/builds
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install llvm-3.2 clang-3.2
```

If you install with apt-get, the path to **scan-build** has already been included in **$PATH** and you can directly run it without specifying its path in command line.

Clang Analyzer documentation (http://clang-analyzer.llvm.org/installation.html) asks linux users to manually build clang and llvm. You are not recommended to do that since it's super hard to build them.

Windows

Install ubuntu or Mac OS X on your virtual machine.

Getting Started

Basic Usage

Basic usage of scan-build is designed to be simple: just type 'scan-build' in front of your build command in the command line.

```
$ scan-build gcc foo.c
$ scan-build make
```

The first command analyzes the the code in foo.c compiled with scan-build and the second on analyzes a project built with make.

Example 1: a Single C program

Here is a buggy C program (you can find more buggy code examples here (examples.zip)).

```
struct S {
  int x;
};

void f(struct S s){
}

void test() {
  struct S s;
  f(s); // warn: passed-by-value arg contain uninitialized data
}

int main(){
  test();
}
```

scan-build reports detected bugs in the command line.

```
$ scan-build gcc 1.c
1.c:10:3: warning: Passed-by-value struct argument contains uninitialized data (e.g., field
  f(s); // warn: passed-by-value arg contain uninitialized data
  ^ ~
1 warning generated
```

In this report, 1.c:10:3 can be interpreted as "file: 1.c, line 10, column 3". And you can find the description of the bug as "warning: Passed-by-value struct argument contains uninitialized data".

scan-build also generates a html report, index.html, as shown below. By default, you can find it in the directory /tmp.

buggy code - scan-build results

User:	troy@troy-Lenovo-B4306
Working Directory:	/home/troy/2015Winter/Data Science in Software Engineering/clang/buggy code
Command Line:	gcc 1.c
Date:	Tue Jan 27 13:17:28 2015

Bug Summary

Bug Type	Quantity	Display?
All Bugs	**1**	☑
Logic error		
Function call argument is an uninitialized value	1	☑

Reports

Bug Group	Bug Type ▾	File	Line	Path Length	
Logic error	Function call argument is an uninitialized value	1.c	10	1	View Report

You are also allowed to specify where to store the generated html report using option, **-o**.

```
$ scan-build -o /target/directory/path/ gcc 1.c
```

Example 2: a C++ Project

Here is a buggy example C++ project (sam.zip) for solving a Sudoku puzzle with driver, header, implementation, and input files. The project builds the executable file and associated dependencies using a Makefile and with the commend 'make' which is supported by scan-build.

```
--root--
  |- easy.txt
  |- Makefile
  |- main.cpp
  |- medium.txt
  |- Puzzle.cpp
  |- Puzzle.h
```

Now let's run clang static analyzer on this project (you may want to clean the project by 'make clean' first).

```
$ scan-build make
g++ -c main.cpp
main.cpp: In function ‘int main(int, char**)’:
main.cpp:15:16: warning: division by zero [-Wdiv-by-zero]
g++ -c Puzzle.cpp
Puzzle.cpp: In member function ‘void Puzzle::print3()’:
Puzzle.cpp:232:8: warning: division by zero [-Wdiv-by-zero]
g++ main.o Puzzle.o -o main
```

Clang static analyzer reports two division-by-zero bugs in main.cpp and puzzle.cpp.

Bug Metrics Retrieval

As you can see, the bug report contains many valuable information, like bug type, bug description and bug numbers. You may want to retrieve all these data and use it in your project. For example, one interesting topic is "What kind of bug appears more often in your program?" and you can investigate the correlation between bug type and bug numbers using the data retrieved from the bug report, as what they did in the relative code churn paper (http://research.microsoft.com/pubs/69126/icse05churn.pdf).

Here is a code example.

```
import sys
import csv
import bs4

with open(sys.argv[1], 'r') as myfile:
        s=myfile.read()

#just get just the table summarizing the bugs
s=s.split('<h2>Bug Summary</h2>',1)[1].split('<h2>Reports</h2>')[0]

page    = s
soup    = bs4.BeautifulSoup(page)
csvout  = csv.writer(sys.stdout)
csvout2 = csv.writer("metrics.csv");

for table in soup.findAll('table'):
    for row in table.findAll('tr'):
        csvout.writerow([tr.text for tr in row.findAll('td')])
        csvout2.writerow([tr.text for tr in row.findAll('td')])
```

This script here traverses the DOM tree in the bug report and gathers data for five metrics, bug group, bug type, file, line number.

To run it, make sure you have installed python bs4 package (http://www.crummy.com/software/BeautifulSoup/) on your laptop. This package allows you to programmatically parse, navigate and search a html document.

```
$ python parseErrors.py /tmp/scan-build-2015-01-27-1/index.html
Bug Group,Bug Type,File,Line,Path Length,
Logic error,Function call argument is an uninitialized value,1.c,10,1,View Report
```

It also writes all data to a csv file, bugSummary.csv in current directory.

Implementing Your Own Static Analyzer

In this section, I will illustrate how to perform source code analysis with a simple example of counting the number of methods. Basically, there are three high-level steps.

1. Parse the source code and generate the AST.

2. Implement an ASTVisitor to traverse the AST generated in step 1.

3. Every time the visitor encounters an AST node that declares a method, it increments the counter.

#1 Extending Clang Static Analyzer

Developer can define their own checker and register them in the clang framework. After registration, the self-defined checker will be called along with built-in checkers when evaluating expressions in the program.

In order to count the number of methods, our checker will be simply implemented with AST Visitors (http://clang-analyzer.llvm.org/checker_dev_manual.html#ast) and we don't even need to leverage the powerful symbolic execution.

1. Check out the source code of clang and llvm, follow steps 1-4 of the Clang Getting Started (http://clang.llvm.org/get_started.html) page.

2. Create a cpp file named SimpleMethodChecker.cpp in the directory lib/StaticAnalyzer/Checkers. This class should inherit from Checker (http://clang.llvm.org/doxygen/classclang_1_1ento_1_1Checker.html) template class; the template parameter(s) describe the type of events that the checker is interested in processing. Since our checker only needs to count the number of methods, we basically need to implement a counter and everytime we encouter function declartion in the AST.

```
class SimpleMethodChecker : public Checker<check::ASTDecl<FunctionDecl>> {
public:

  int counter;

  SimpleMethodChecker():counter(0){}

  void checkASTDecl(const FunctionDecl *D, AnalysisManager &Mgr, BugReporter &BR) const {
     counter ++;
  }
};
```

3. The following code should also be added to SimpleMethodChecker.cpp in order to register our own checker in Clang framework.

```
void ento::registerSimpleStreamChecker(CheckerManager &mgr) {
  mgr.registerChecker();
}
```

4. All checkers are defined in a table, lib/StaticAnalyzer/Checkers/Checkers.td so we also need to add our checker in the table.

```
let ParentPackage = Core in {
...
def SimpleMethodChecker : Checker<"SimpleMethodChecker">,
  HelpText<"Counting the number of methods in cpp files">,
  DescFile<"SimpleMethodChecker.cpp">;
```

5. Build the clang project

6. After adding a new checker to the analyzer, you can verify that the new checker was successfully added by seeing if it appears in the list of available checkers.

```
$ clang -cc1 -analyzer-checker-help
```

#2 Using a Stand-alone C/C++ Parser

Alternatively, you can use stand-alone C/C++ parsers to generate the AST and then implement AST visitors to analyze source code. In this case, I used a C/C++ parser called pycparser (https://github.com/eliben/pycparser) and write a simple python script (method_counter.py) to count the number of methods.

```
CPPPATH = '../pycparser/utils/cpp.exe' if sys.platform == 'win32' else 'cpp'

class FuncDefVisitor(c_ast.NodeVisitor):
    counter = 0
    def visit_FuncDef(self, node):
        self.counter = self.counter + 1

def count_methods(filename):
    # Note that cpp is used. Provide a path to your own cpp or
    # make sure one exists in PATH.
    #
    try:
        ast = parse_file(filename, use_cpp=True, cpp_path=CPPPATH, cpp_args=r'-I../pycparse
    except:
        return

    v = FuncDefVisitor()
    v.visit(ast)
    print(v.counter)
```

#3 Using JDT ASTParser

The ASTParser in Java Development Toolkit (JDT) provides good support for program analysis If you are working with Java programs.

1. Create a java project and add JDT jar files to the build path.

2. Write code to parse java files and implemennt an ASTVistor to traverse the AST. Everytime it visits a MethodDeclaration node, increment the counter.

```
public class MethodCounter {

            int counter = 0;

                public void count(String str) {

                        // call ASPParser to generate the AST
                        ASTParser parser = ASTParser.newParser(AST.JLS3);
                        parser.setSource(str.toCharArray());
                        parser.setKind(ASTParser.K_COMPILATION_UNIT);

                        final CompilationUnit cu = (CompilationUnit) parser.createAST(null)

                        // traversing the AST and count the number of MethodDeclaration nod
                        cu.accept(new ASTVisitor() {

                                public boolean visit(MethodDeclaration node) {
                                        counter ++;
                                        return true;
                                }
                        });
                }

                //read file content into a string
                public String readFileToString(String filePath) throws IOException {
                        StringBuilder fileData = new StringBuilder();
                        BufferedReader reader = new BufferedReader(new FileReader(filePath)

                        char[] buf = new char[10];
                        int numRead = 0;
                        while ((numRead = reader.read(buf)) != -1) {
                                String readData = String.valueOf(buf, 0, numRead);
                                fileData.append(readData);
                                buf = new char[1024];
                        }

                        reader.close();

                        return  fileData.toString();
                }

                public static void main(String[] args) throws IOException{
                        MethodCounter mc = new MethodCounter();
                        String code = mc.readFileToString("test.java");
                        mc.count(code);
                        System.out.println(mc.counter);
                }
}
```

The source code is available here (MethodCounter.zip). You just need to import it into your Eclipse workspace.