本 科 实 验 报 告

实验名称：vscode 插件开发

学　　员：马铁诚 刘思睿 韩启华 高雪原

学　　号：201902001031 201902001038 201902001045 201902001048

培养类型：无军籍本科　　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：教授

实 验 室：306-705　　实验日期：2022.7.12

国防科学技术大学训练部制

一、实验要求：

1. 四人一组，自行挑选针对 C/C++或 Java 程序的静态或动态测试工具，将其作为服务端完成扩展到 vscode 插件中。静态测试工具参考：CppCheck、KLEE、Clang-tidy 等；动态测试工具参考：AFL、LibFuzzer 等

2. 代码架构

采用客户端服务器架构，推荐使用 gRPC 进行客户端与服务端之间的通信

3. 客户端展示内容要求

针对不同类型的测试工具，应该能够对测试结果有直观的展示

例如，对于静态代码分析工具插件，如 C/C++的静态代码分析工具 CppCheck 的 vscode 插件 Cppcheck plug-in，在分析结束后应该能够在控制台 Output 中输出较为详细的分析结果；对于符号执行工具插件，如 KLEE 的 vscode 插件，应该能够在分析结束后生成并直观的展示测试用例；对于动态测试工具插件，如 AFL 的 vscode 插件，应该能够在测试结束后展示出特殊的测试用例（让程序崩溃的测试用例等）以及覆盖率信息。

二、实验环境：

1.Ubuntu 系统（18.04 或 20.04）。

2.vscode

3.nodejs（JavaScript 的运行环境）

4.npm（nodejs 的包管理工具）

三、测试工具选择：

1. 选择 CppCheck 作为服务端工具

2. 插件前端选择使用 JavaScript 编写，后端使用 python 编写，使用 gRPC 进行通讯

四、实验内容：

1.配置插件开发项目所需要的环境，包括下载并更新 vscode，nodejs，npm 等，然后在命令行输入 `Npm install @grpc/grpc-js`

`Npm install @grpc/proto-loader`

安装所需要的包扩展。

然后执行命令 `npm install -g yo generator-code`，安装脚手架，接着进入工作目录，使用脚手架创建工程，其中 language 选择 JavaScript，并设施好插件名称。

2. package.json 内容详解

本次插件实验的插件名设置为“plugin”，其在显示在插件市场的名称也是“plugin”，插件版本号为“0.0.1”，如图所示：

```
{
    "name": "plugin",
    "displayName": "plugin",
    "description": "a homework",
    "version": "0.0.1",
    "engines": {
        "vscode": "^1.69.0"
    },
    "categories": [
        "Other"
    ],
```

插件的激活数组设置为识别到使用语言为 C 语言或者 cpp 时激活，对应于集成在后端的 cppcheck 工具。如图所示：

```
    "activationEvents": [
        "onLanguage:c",
        "onLanguage:cpp"
    ],
```

在 contributes 配置项中声明所开发插件使用的命令，包括两个，分别是 plugin.useCppCheck 使用 cppcheck 工具，和 plugin.highlight 将运行 cppcheck 工具产生的结果使用高亮在源代码中展示出来。如图所示：

```
    "contributes": {
        "commands": [
            {
                "command": "plugin.useCppCheck",
                "title": "Use CppCheck"
            },
            {
                "command": "plugin.highLight",
                "title": "HighLight CppCheck Result"
            }
        ],
```

然后定义菜单，首先将上述两个命令加入编辑器右键菜单中从而实现在代码中单机右键激活菜单后能显示并选择配置项中的两个命令，并设置为当源文件为 C 语言或者 cpp 编写时才在菜单中出现，如图所示：

```
    "menus": {
        "editor/context": [
            {
                "when": "resourceLangId == c ||  resourceLangId == cpp",
                "command": "plugin.useCppCheck",
                "group": "navigation@1"
            },
            {
                "when": "resourceLangId == c ||  resourceLangId == cpp",
                "command": "plugin.highLight",
                "group": "navigation@2"
            }
        ],
```

同时，将 plugin.useCppCheck 命令添加到资源管理器右键菜单，如图所示：

```
        "explorer/context": [
            {
                "when": "resourceLangId == c || resourceLangId == cpp || explorerResourceIsFolder",
                "command": "plugin.useCppCheck",
                "group": "navigation"
            }
        ]
```

3. extension.js 内容详解

设置 activate()激活函数，分别注册并实现 package.json 中的两个命令。

首先引入写好的 client 文件，其内部实现了传输数据的具体函数，将其命名为 client。

```
const client = require('./client/client')
```

对于 plugin.useCppCheck 命令，设置可选参数 uri，当从编辑器右键菜单执行时将当前打开文件路径 uri 传进去。因此在函数内部判断是否得到当前文件的路径 uri，如果得到，则将其作为参数传入 client 文件中实现的 cppcheck 函数。如图所示：

```
context.subscriptions.push(vscode.commands.registerCommand('plugin.useCppCheck',(uri)=>{
    var currentFilePath;
    if(uri){
        currentFilePath = uri.fsPath;
        var fileName = {
            name: currentFilePath
        };
        console.log(client);
        client.cppCheck(fileName);
    }
}));
```

对于 plugin.highLight 命令，与上述命令类似，不同点为将得到的当前路径作为参数传入 client 文件的 highLight 函数。如图所示：

```
context.subscriptions.push(vscode.commands.registerCommand('plugin.highLight',(uri)=>{
    var currentFilePath;
    if(uri){
        currentFilePath = uri.fsPath;
        var fileName = {
            name: currentFilePath
        };
        console.log(client);
        client.highLight(fileName);
    }
}));
```

4. 安装插件： pip3 install grpcio grpcio-tools，sudo apt install cppcheck

5. 修改 test.proto，指定包、函数、返回值、值的特性。（1,2 代表次序）

```
syntax = "proto3";

package kokaze;

service pluginService{

    rpc cppCheck(Filename) returns (CppCheckContent) {}
    rpc highLight(Filename) returns (TestCaseContent){}

}


message Filename{
    string name = 1;
}
message TestCaseNumber{
    int32 number = 1;
}
message TestCaseName{
    string name = 1;
}
message TestCaseContent{
    string title = 1;
    repeated string content = 2;
}
message CppCheckContent{
    string content = 1;
    string output = 2;
}
```

6. Client.js：实现 extension.js 中调用的相关函数的具体情况，

```
function cppCheck(filename) {
    client.cppCheck(filename,(err,response)=>{
        if(err){
            vscode.window.showErrorMessage(err);
        }else{
            vscode.window.showInformationMessage(response.content);
            var output_channal = vscode.window.createOutputChannel('plugin');
            output_channal.clear();
            output_channal.append(response.output);
            output_channal.show();
        }
    });
}
```

```
function highLight(filename) {
    client.highLight(filename,async (err,response)=>{
            let decorationType = vscode.window.createTextEditorDecorationType({
                backgroundColor: '#bbd0353d'
            });
            let editor = vscode.window.activeTextEditor;
                //const path = '/home/gao/main/test.c';
            var lines = [];
            for(var i=0;i<response.content.length;i++){

                var line = response.content[i]-1;
                lines.push(new vscode.Range(line, 0, line, 100));
            }
            console.log(lines)
            editor.setDecorations(decorationType, lines);

            vscode.window.showInformationMessage(response.title);

        });
}
```

上述的 cppcheck 的函数，具体情况是：先监听后端 client 的情况，若有返回查看是报错还是信息，如果是报错就展示错误信息，否则将其返回值（对用户的发出提示框，对具体情况输出至 output）

上述的 highlight 的函数使得在编辑器内对后端返回的报错的相应的行数标黄。

7. 生成辅助 py 文件：

python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=. ./test.proto

```
# Generated by the gRPC Python protocol compiler plugin. DO NOT EDIT!
"""Client and server classes corresponding to protobuf-defined services."""
import grpc

import test_pb2 as test__pb2


class pluginServiceStub(object):
    """Missing associated documentation comment in .proto file."""

    def __init__(self, channel):
        """Constructor.

        Args:
            channel: A grpc.Channel.
        """
        self.cppCheck = channel.unary_unary(
                '/kokaze.pluginService/cppCheck',
                request_serializer=test__pb2.Filename.SerializeToString,
                response_deserializer=test__pb2.CppCheckContent.FromString,
                )
        self.highLight = channel.unary_unary(
                '/kokaze.pluginService/highLight',
                request_serializer=test__pb2.Filename.SerializeToString,
                response_deserializer=test__pb2.TestCaseContent.FromString,
                )
```

8. 编写后端代码 cppcheck.py：这里要做的是首先连接服务器：（与 client.js 握手）

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    test_pb2_grpc.add_pluginServiceServicer_to_server(PluginServiceGeneration, server)
    server.add_insecure_port('[::]:50000')
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    logging.basicConfig()
    serve()
```

然后等待前端做出相应动作。等前端做出动作后，后端监听到动作的发生，就进行处理。

后端 cppcheck 函数原理是：将传入的路径按照文件和目录进行区分，分别调用 cppcheck 的相应命令生成.txt 和.xml 的报错文件，放入集合 error 文件夹。然后将 txt 文件的内容和给用户显示的信息打包传回前端。

```
class PluginServiceGeneration(test_pb2_grpc.pluginServiceServicer):
    def cppCheck(self, callback):
        print('INFO: starting')
        filePath = self.name
        print('INFO: checking '+filePath)
        '''
        error_path = os.path.join(filePath,'error')
        error_xml_path = os.path.join(filePath,'error_xml')
        print('INFO: error_path: '+error_path)
        print('INFO: error_xml_path: '+error_xml_path)
        os.system('mkdir -p '+error_path)
        os.system('mkdir -p '+error_xml_path)
        '''
        filename = filePath.split('/')[-1]
        realPath = os.path.dirname(filePath)
        print('INFO: realPath: '+realPath)
        if(os.path.isdir(filePath)):
            os.system('mkdir -p '+os.path.join(filePath,'error'))
            error_ins = 'cppcheck --enable=all --std=posix '+filePath+' 2> '+filePath+'/error/dir_err.txt'

            error_path_ins = 'cppcheck --enable=all --std=posix --xml '+filePath+' 2> '+filePath+'/error/dir_err.xml'
            os.system(error_ins)
            os.system(error_path_ins)
            with open(filePath+'/error/dir_err.txt') as f:
                output = f.read()
        else:
            os.system('mkdir -p '+os.path.join(realPath,'error'))
            error_ins = 'cppcheck --enable=all --std=posix '+filePath+' 2> '+realPath+'/error/'+filename+'_error.txt'
            error_path_ins = 'cppcheck --enable=all --std=posix --xml '+filePath+' 2> '+realPath+'/error/'+filename+'_error.xml'
            os.system(error_ins)
            os.system(error_path_ins)
            with open(realPath+'/error/'+filename+'_error.txt') as f:
                output = f.read()
        print('INFO: error_path: '+error_ins)
        print('INFO: error_path_xml: '+error_ins)
        print('INFO: finished '+filePath)
        return test_pb2.CppCheckContent(content="Checking finished. Result is stored." output=output)
```

后端 highlight 函数原理是：找到集合 error 文件夹，然后分析 xml 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
    <cppcheck version="1.82"/>
    <errors>
        <error id="ignoredReturnValue" severity="warning" msg="Return value of function malloc() is not used." verbose="Return value of
            <location file="/home/ma/test/test.c" line="5"/>
        </error>
        <error id="leakReturnValNotUsed" severity="error" msg="Return value of allocation function &apos;malloc&apos; is not stored." v
            <location file="/home/ma/test/test.c" line="5"/>
        </error>
        <error id="ignoredReturnValue" severity="warning" msg="Return value of function malloc() is not used." verbose="Return value of
            <location file="/home/ma/test/test_1.c" line="5"/>
        </error>
        <error id="leakReturnValNotUsed" severity="error" msg="Return value of allocation function &apos;malloc&apos; is not stored." v
            <location file="/home/ma/test/test_1.c" line="5"/>
        </error>
        <error id="missingIncludeSystem" severity="information" msg="Cppcheck cannot find all the include files (use --check-config for
    </errors>
</results>
```

找到其中的报错信息行数，把它和给用户显示的信息打包传回前端。

```
    def highLight(self, callback):
        import xml.etree.ElementTree as ET
        from xml.etree.ElementTree import Element
        print('INFO: starting')
        filePath = self.name
        realPath = os.path.dirname(filePath)
        filename = filePath.split('/')[-1]
        cont = []
        if(not(os.path.exists(os.path.join(realPath,'error')))):
            return test_pb2.TestCaseContent(title='You have not use Cppcheck yet.',content='')
        if(not(os.path.exists(os.path.join(realPath,'error/dir_err.xml')))):
            if(not(os.path.exists(os.path.join(realPath,'error/'+filename+'_error.xml')))):
                return test_pb2.TestCaseContent(title='You have not use Cppcheck yet.',content='')
            else:
                xml_name = os.path.join(realPath,'error/'+filename+'_error.xml')
                tree = ET.parse(xml_name)
                root = tree.getroot().getchildren()[1]
                for err in root:
                    print(err)
                    if(err.attrib['severity'] == 'warning' or err.attrib['severity'] == 'error'):
                        if(err[0].attrib['file'] == filePath):
                            cont.append(err[0].attrib['line'])
        else:
            xml_name = os.path.join(realPath,'error/dir_err.xml')
            tree = ET.parse(xml_name)
            root = tree.getroot().getchildren()[1]
            for err in root:
                print(err.attrib['severity'])
                if(err.attrib['severity'] == 'warning' or err.attrib['severity'] == 'error'):
                    if(err[0].attrib['file'] == filePath):
                        cont.append(err[0].attrib['line'])
        return test_pb2.TestCaseContent(title='CppCheck HighLight has prepared.',content=cont)
```

后端处理后将按照 test.proto 规定的输出格式输出至前端接收。

```
            with open(realPath+'/error/'+filename+'_error.txt') as f:
                output = f.read()
        print('INFO: error_path: '+error_ins)
        print('INFO: error_path_xml: '+error_ins)


        print('INFO: finished '+filePath)

        return test_pb2.CppCheckContent(content="Checking finished. Result is stored.",output=output)
```

最后前端作最后的处理。

前端 cppcheck 函数原理是：先将给用户显示的信息显示出来，然后将返回的.txt 报错文件打印到 output 里面。

```
function cppCheck(filename) {
    client.cppCheck(filename,(err,response)=>{
        if(err){
            vscode.window.showErrorMessage(err);
        }else{
            vscode.window.showInformationMessage(response.content);
            var output_channal = vscode.window.createOutputChannel('plugin');
            output_channal.clear();
            output_channal.append(response.output);
            output_channal.show();
        }
    });
}
```

前端 highlight 函数原理是：highlight 函数对报错的代码函数实现高亮，方便用户找到报错的代码行数。先创建 decorationType,设定背景色等。然后获取报错的行数，将需要高亮的代码范围加入列表。最后通过 setDecorations 对所选范围进行高亮。

```
function highLight(filename) {
    client.highLight(filename,async (err,response)=>{
            let decorationType = vscode.window.createTextEditorDecorationType({
                backgroundColor: '#bbd0353d'
            });
            let editor = vscode.window.activeTextEditor;
                //const path = '/home/gao/main/test.c';
            var lines = [];
            for(var i=0;i<response.content.length;i++){

                var line = response.content[i]-1;
                lines.push(new vscode.Range(line, 0, line, 100));
            }
            console.log(lines)
            editor.setDecorations(decorationType, lines);

            vscode.window.showInformationMessage(response.title);

        });
}
```

9. xml 转换为 html 进行展示：

9.1 引入相关依赖

查阅网络资料[1]得知，将 XML 文件转为 HTML 文件需要引入 sys 和 lxml 包中的 etree 模块，因此，分别引入该两部分即可：

```
import sys
from lxml import etree
```

9.2. 确定 XML 文件路径

首先，获取开发目录的绝对路径，使用 proto 传到后端的数据的 name 就是开发目录的绝对路径：

```
filePath = self.name
```

然后，获取生成的 XML 文件的路径，XML 文件存放在开发目录下的/error 目录下，所以直接使用字符串拼接即可：

```
xml_path = filePath + '/error/dir_err.xml'
```

9.3. 生成 XML DOM

使用 etree 的 parse()方法，解析该 XML 文件，生成 XML 树，再使用 etree 的 tostring()方法将 XML 树转换为字符串，最后，使用 etree 的 XML()方法将该字符串转换为 XML DOM：

```
xml_tree = etree.parse(xml_path)
xml_file_content = etree.tostring(xml_tree, encoding = 'UTF-8', method = 'xml')
xml_dom = etree.XML(xml_file_content)
```

9.4. 生成 HTML 字符串

首先，使用 etree 的 XSLT()方法，将之前生成的 XSL DOM 转换为对象 transform，使用 transform 将 XML DOM 转换为 HTML，最后将 HTML 强制转换为字符串类型：

```
xml_dom = etree.XML(xml_file_content)
transform = etree.XSLT(xsl_dom)
html_result = transform(xml_dom)
```

9.5. 生成 HTML 文件

以写的方式打开目标 HTML 文件，由于不存在该文件，会自动创建该文件，然后将 HTML 字符串写入 HTML 文件，即生成了符合要求的 HTML 文件：

```
with open(filePath + '/error/htmlResult.html') as fhtml:
    fhtml.write(str_html)
    f.close()
```

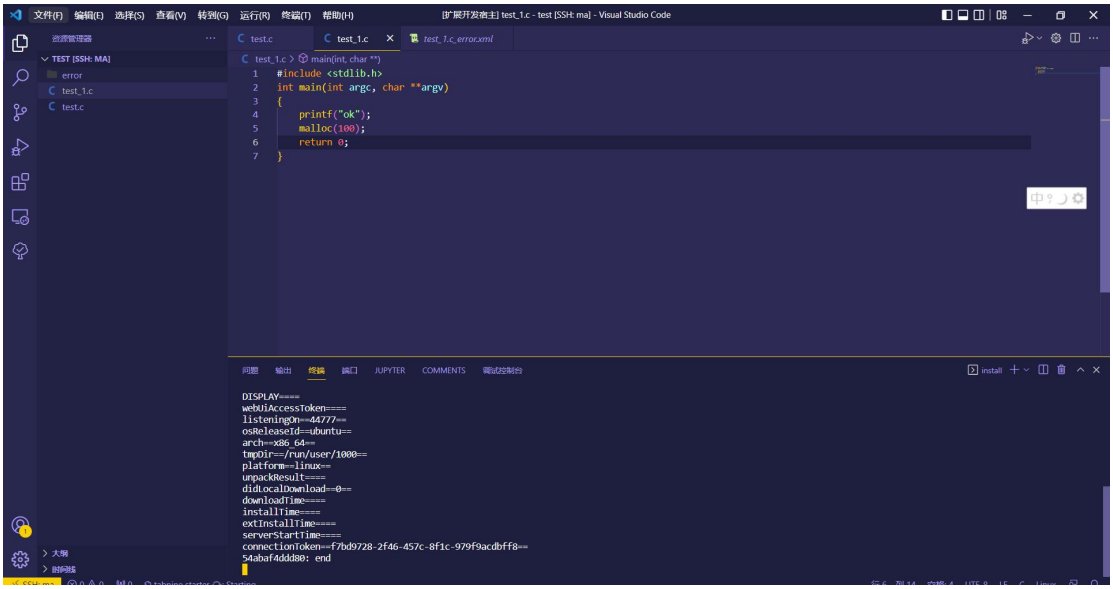
9.6 结果

```
def to_html(filePath):
    import sys
    from lxml import etree
    xml_path = filePath + '/error/dir_err.xml'
    xml_tree = etree.parse(xml_path)
    xml_file_content = etree.tostring(xml_tree, encoding = 'UTF-8', method = 'xml')
    xml_dom = etree.XML(xml_file_content)
    transform = etree.XSLT(xml_dom)
    html_result = transform(xml_dom)
    print(html_result)
    str_html = str(html_result)
    with open(filePath + '/error/htmlResult.html') as fhtml:
        fhtml.write(str_html)
        fhtml.close()
```

10. 无误后运行。运行步骤：

(1) 在 client 文件夹内输入命令 python3 cppcheck.py

(2) F5 调试



(3) 执行动作，观察结果。

五、实验结果：

实验结果在视频内展示。