Master Thesis

Implementation and Validation for a Continuous Testing Method in Automotive Software Development

Abstract

In the past few years, the importance of software unit testing has increased drastically in the automotive field with the increasing complexity of systems used for driver assistance.

Unit testing and integration testing are commonly used in automotive software development processes. And the testing enables decrease errors and gains efficiency during the development phase. All of these advantages can only be achieved when the entire organization has a clear strategy to follow, with established rules and guidelines, as well as clear process definitions.

The purpose of this thesis is to validate a continuous testing method in automotive software development and this concept is implemented internally to the department Engineering Application Ultrasonic at Robert Bosch GmbH.

As a result of this thesis, the testing process was defined in accordance with international norms and many work products (e.g. templates and scripts) were developed. The implementation of continuous unit testing was analysed and the application and of the concept of continuous integration testing was implemented.

Introduction

Continuous testing is a key step that cannot be avoided in automotive software development. In terms of the continuous testing method, the V-mode development process is the most important one. And the computer-aided control system design, which is not only the design and offline simulation of control schemes but also a real-time rapid control prototype, a product code generation and hardware-in-the-loop testing, is widely used in the development process. Bosch's park pilot project department expected to conduct preliminary continuous testing in internal of the department to detect defects in the design as early as possible. Thus, a continuous testing method is needed.

This thesis dissertating manual testing which including unit testing and integration testing and analysed the validation of continuous testing method of both as well. Manual testing of these two tests was fulfilled and got good test results in the early stage of the thesis internship. And in the second stage, the automation feasibility of the integration test was analysed and partial automation was realized. The parts that cannot be automated are mainly due to technical limitations, unsupportive of hardware and time constraints.

Comparing to integration testing, test cases must be written for each specific unit when unit testing. The test tool only can be started when the corresponding test cases were completed. For integration testing, to set up and debug the bench to guarantee good operation would be the first step. In addition, manually set some signals before the test to adjust the ECU to the state that meets the test requirements was obligatory and un-skippable. The ultimate goal of this thesis is to set these manually setting parts as scripts and save them as a new configuration to enable automatic testing by simply launching the configuration, and even launching tests remotely periodically via Jenkins.

Working Process Design

The general approach is the same for both tests. Figure 1 shows the overall design process.

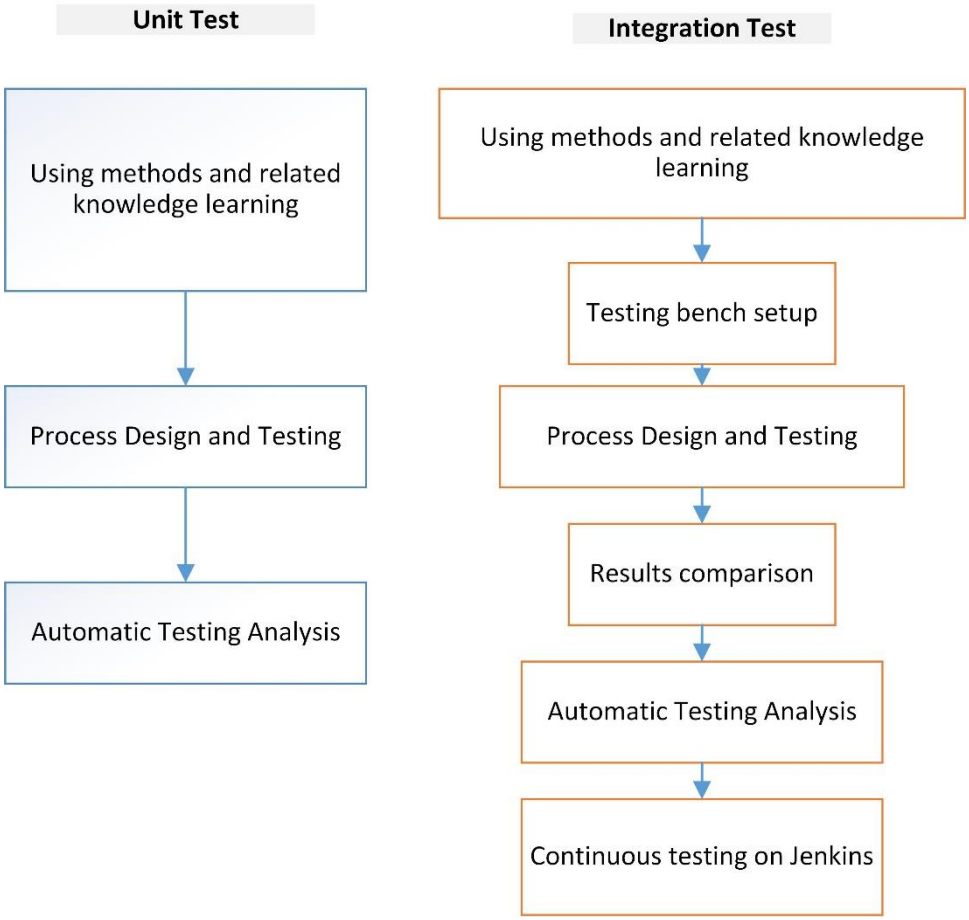

Figure 1 Plan of Thesis

Test Bench for Integration Test

This part is a description of the hardware required for integration testing. The Bench's configuration can be described in Figure 2(the wirings of power supply are omitted). The test bench are composed of the following components: iC5000, JTAG, park pilot ECU, Tischbox, Echo simulator, Vector interface.

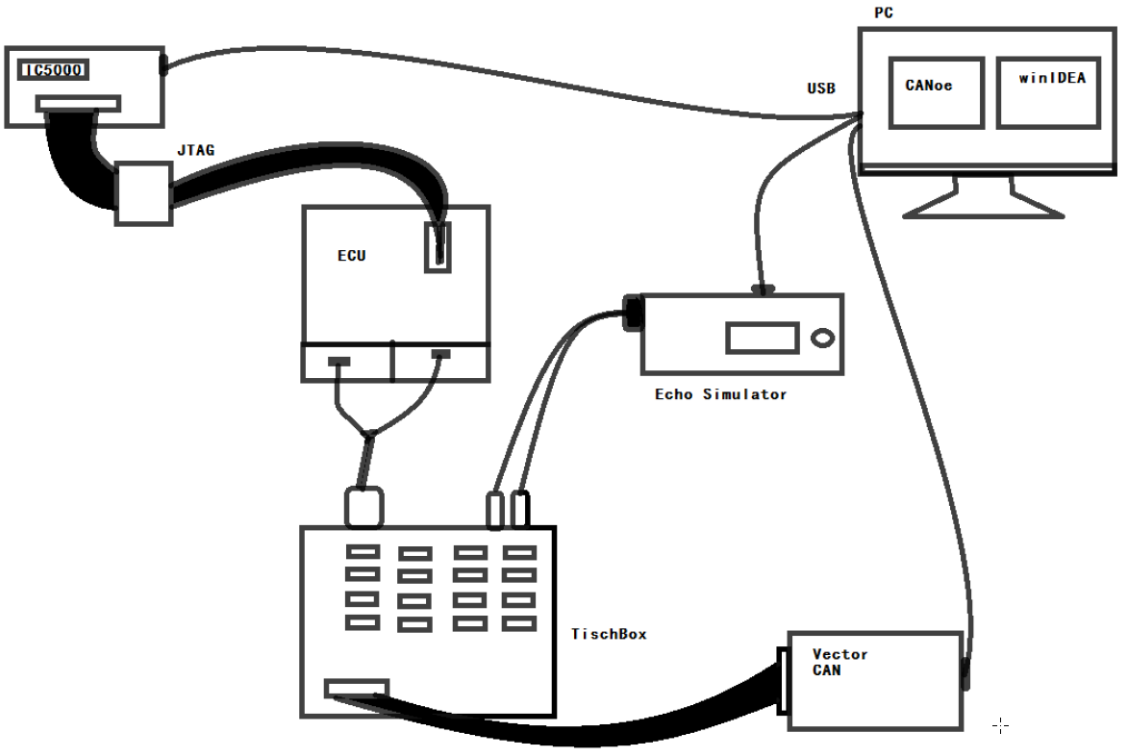

Figure 2 The test bench's configuration

Instances

Automatic parking System is in active mode when the vehicle is in reverse gear or below a speed of 15 kmh. In the hardware part required for the automatic parking system, narrow beam ultrasonic technology was mainly used. The ultrasonic sensors are distributed at the positions of the front and rear bumpers.

The sensors send ultrasonic signals to search for available parking spaces and send this information to the Signal Acquisition Module (SAM) continuously. Regardless of whether the active parking switch on or off, the sensor is working as long as the vehicle is moving. This sensor is the same as the sensor principle used in the reversing radar, but it is not interchangeable.

There are three zones duly measured from rear bumper within which the sensor could detect and prompt user to action.

1) 60 cm to 120cm zone (slow beeping)

2) 30 cm to 60 cm zone (fast beeping)

3) 0 cm to 30 cm zone (continuous beeping)

Automotive SPICE and ISO26262

Two primary norms are relevant at parking and manoeuvring assistance system: Automotive SPICE and ISO 26262.

It stands for automotive software process improvement and capability determination and consists of two elements, a process reference model (PRM) and a process assessment model (PAM).

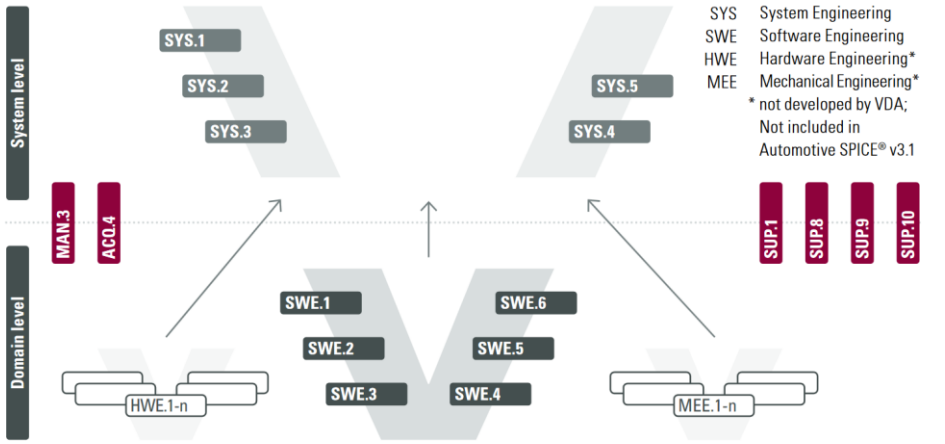

Figure 3 V-model *[13]*

ISO 26262 provides an abstract classification of safety risk in an automotive system or element called Automotive Safety Integrity Levels (ASIL). ASIL classification are used within ISO 2626 to express the level of risk reduction required to prevent a hazard, with ASIL A representing the lowest and ASIL D the highest. Requirements and methods are applicable and selected based on the ASIL classification.

Table 1 Methods for software unit testing

	Methods	ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

Process of Unit Test

More specifically, TPT will connect to MATLAB and SIMULINK and perform pre-test configuration. This process is usually around five minutes. Then the user only needs to select the use case that needs to be tested and click on it. Two test reports will be obtained after the operation is completed. One is from the TPT, mainly about whether the results of the operation are successful. Another report came from MATLAB, mainly about test coverage.

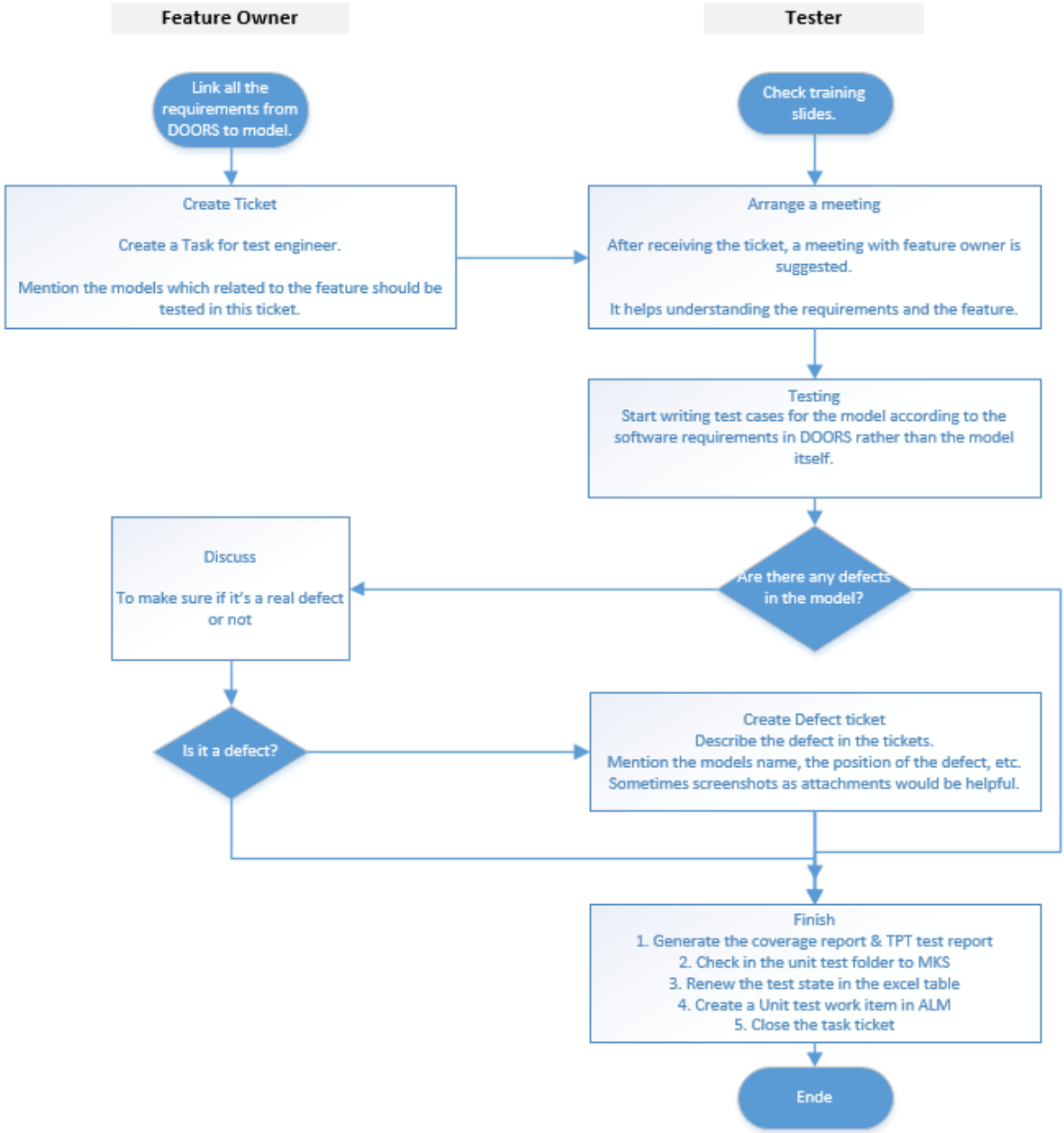

Figure 4 Test process of unit test

The Table 2 is methods for deriving test cases for software unit testing from ISO26262. Supposing there is a variable named 'Number_A' that can take a value from 1 to 100.

The 1a method Analysis of requirements representing the requirement itself, which means the value inside must be obtained in software unit testing. Such as 20, 30 and 50.

The 1b method Generation and analysis of equivalence classes representing that, if value of 30 be take in previous test cases, then it can be considered the value 29 or 31 is also representative of the test case that 'Number_A' valued 30.

The 1c method Analysis of boundary values representing that the value of the boundary should also have associated test cases, such as 1 or 100 of 'Number_A'.

The 1d method Error guessing representing that values which should not exist should also have associated test cases, such as -1 or 1000 of 'Number_A'.

In the above test cases, the methods for deriving test cases for software unit testing are satisfied by adding more combinations of requirements and conditions that do not exist in requirements.

Table 2 Methods for deriving test cases for software unit testing [14]

	Methods	ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.

Automatic Testing Analysis

For unit testing, the basic use of the software (TPT) has been mastered by the department colleagues. Although there are still many new problems that need to be consulted to support engineers of Pike Tec GmbH during the learning process. There are two main things have been completed:

- Designed and standardized the process of unit testing in the park pilot department of Bosch.
- Analysed the possibility of TPT performing continuous testing in the Jenkins based on existing basis.

Pike Tec provided a special TPT Plugin for Jenkins. This plugin allows users to execute tests model in TPT via Jenkins. XML file can be generated in JUnit format for the reporting of test Results. And test report in Jenkins can be display as well.

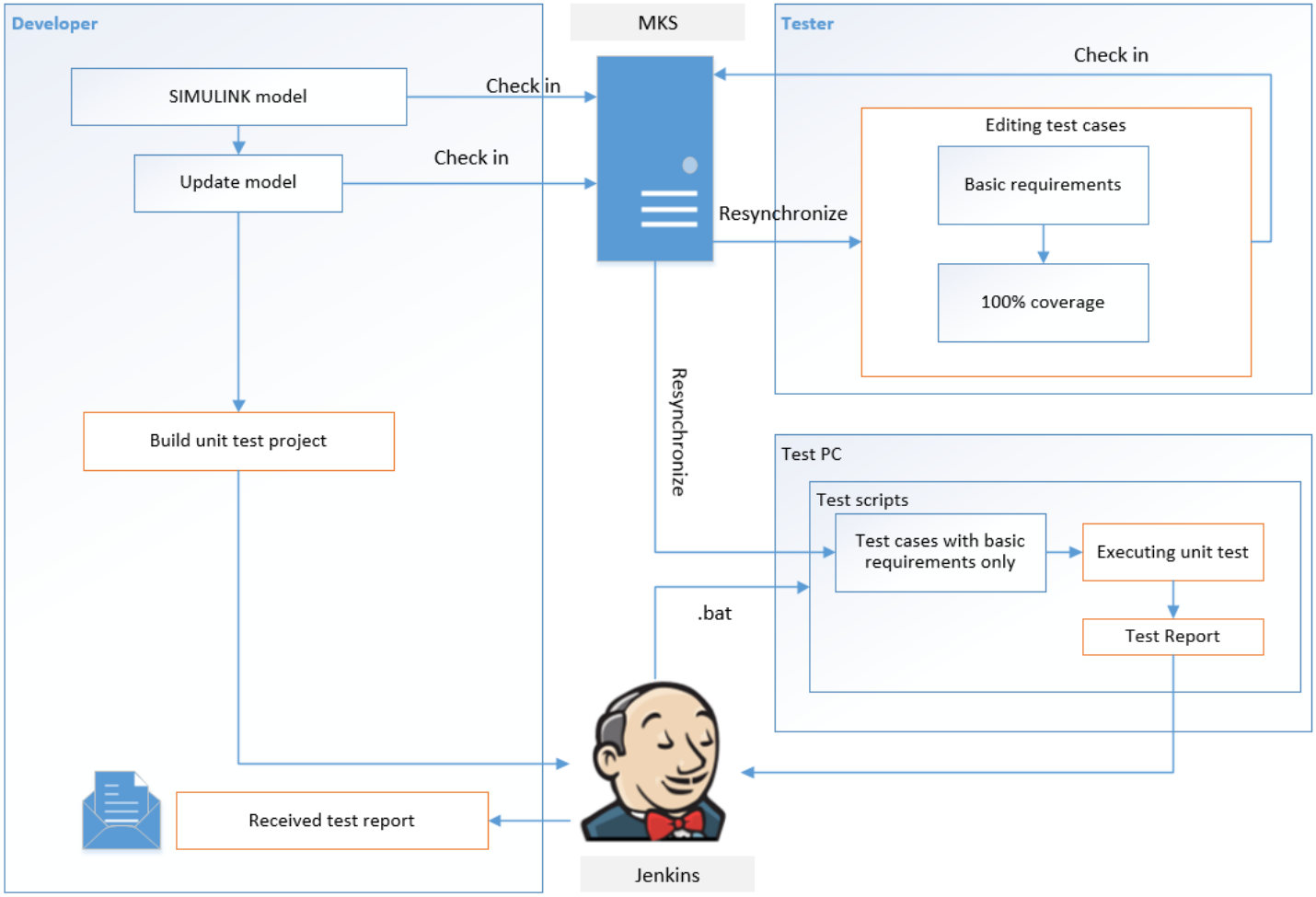

Figure 5 Automatic testing process of unit test

Thus, the automatic testing process of unit test as shown in Figure 5. In this flow chart, the unit test process been partially automated. Surely, the test ticket and defect tickets can only create manually. But it is possible for developer to execute the unit test in test pc via Jenkins without the test software installation. And it is also possible for developer to check the test result. There is no extra work be bring to the tester in this process except checking in the test scripts twice after finishing test cases modification. And the test script which only covered the basic requirement usually contained ten to twenty test cases or even less.

This process is limited for the improvement of unit testing. Because the length of time spent in the unit test portion is determined by the complexity of the test case and the unit being tested. And typically takes between one hour

and eight hours. Once the program in the unit has been modified or added, the test case should be updated. Otherwise it is impossible to achieve 100% coverage. This poses a challenge to the automation of unit testing, because writing test cases takes much longer than eight hours compared to up to eight hours of testing time. It may take only three or five days, but up to one month to write test cases when the model is complex is also possible. Unit testing can be executed at the end of the day and the test results will be available the next morning. Therefore, automated unit testing does not increase productivity significantly.

Process of Integration Test

The process of integration test is shown in Figure 6:

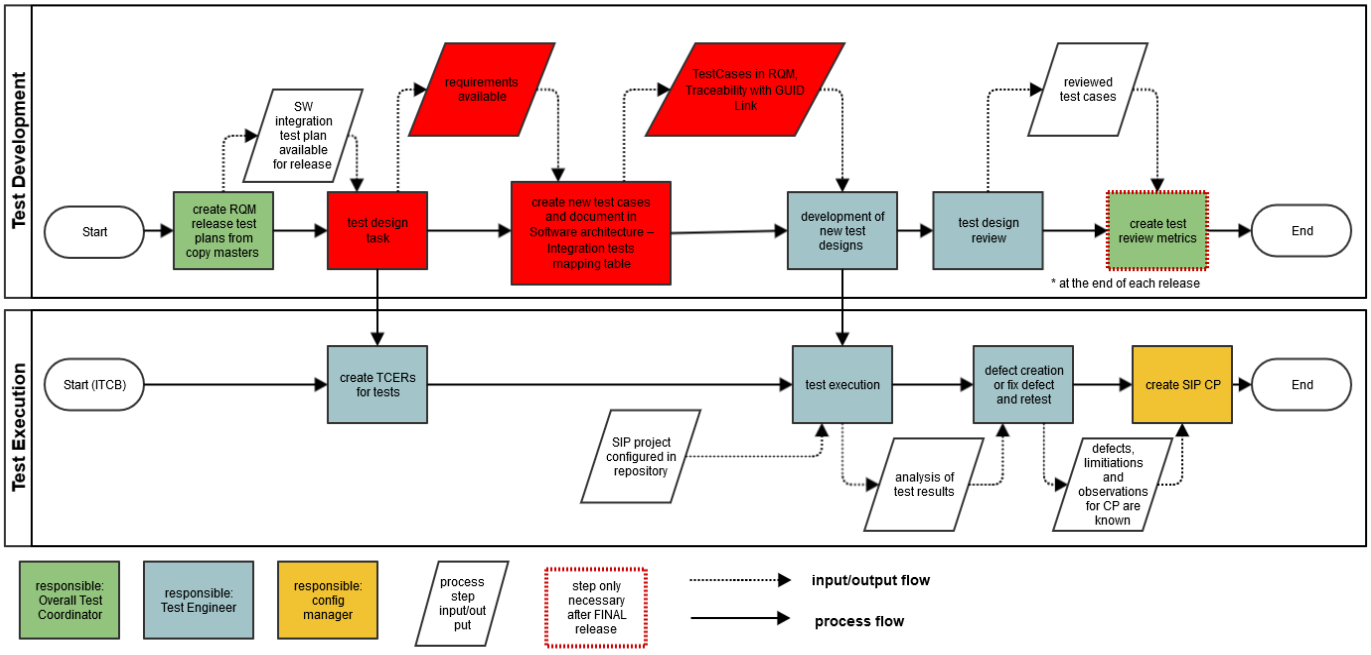

Figure 6 Software integration test process flow chart

Manually Integration Test

Manually integration testing consists of three steps: resynchronizing files from MKS, flashing ECU and setting CANoe configuration. The manually integration test does not refer to full manual test. There are also parts that can be automatically configured, such as the execution of testing. The CAN signals can modified in CANoe rest bus simulation (at most time statically) which means

these signals have to set manually by clicking and write signals values before the continuous test method proposed.

The simulated obstacle will display in CANoe as shown in Figure 7. The echo simulator will simulate the ultrasonic response and sent it to park pilot ECU for analyse.

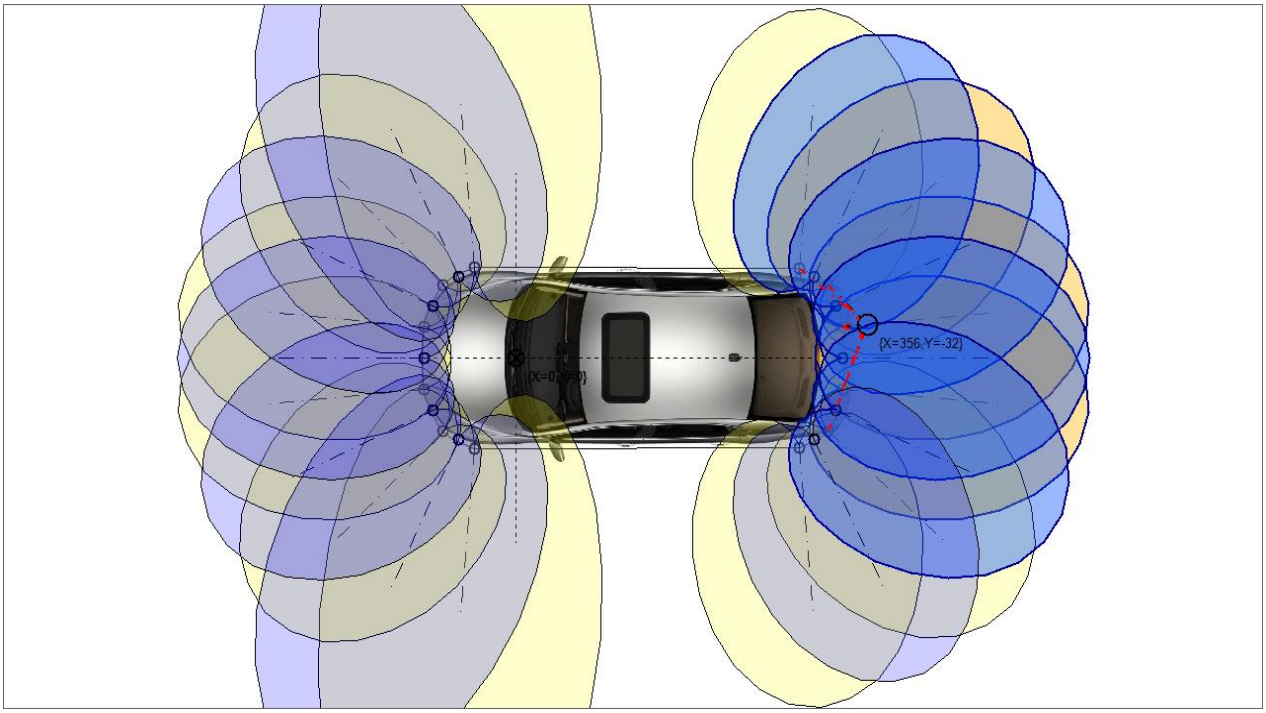

Figure 7 Obstacle simulation on rear

Automatic Testing Analysis

For integration testing, there are more challenges than unit testing. The first is because the integration test is a HIL test, then some testing bench must be connected to the computer. These hardware devices must be debugged to work together in concert to ensure that the test is accurate. The second point is that the pre-test preparation for integration testing is cumbersome. Because of the good test tool TPT, unit tests only need to open the TPT and run the test case. The integration test needs to flash the hex file of the current project into the ECU being tested and perform manual signal configuration. The third point is that the test cases in the integration test are more complicated than the unit tests and all the functions need to be verified here.

Because integration testing is a higher-order test than unit testing, test cases for integration testing do not need to be written for each unit. Thus, the use cases for integration testing will hardly change much. Regardless of the

improvements made to the units in the system, as long as there is no major adjustment to the customer's needs, it should pass the integration test.

The process of automatic testing is shown in Figure 8.

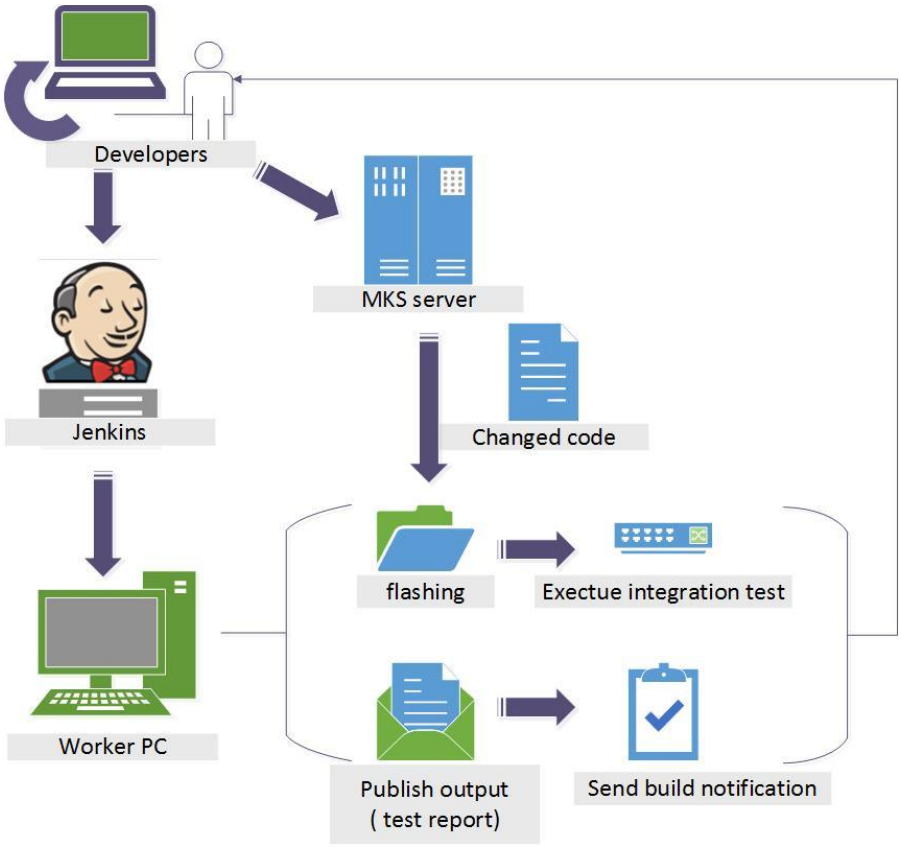

Figure 8 Contiuous testing process

Continuous Testing

Continuous testing for integration testing involves several steps: test cases modification, winIDEA workspace configuration, scripts implementation, automatically execution and continuous deployment. And these steps must be performed in sequence.

Test Cases Modification and winIDEA Workspace Configuration

Test Cases Modification

Rebuilding test cases is necessary during the deployment of continuous testing. Test cases are designed by our colleagues in India and at present, there are 70 test cases for complete integration test used by park pilot department.

There are three types of obstacle checking. They are:

- Obstacle Detection. In this mode, only the presence or absence of obstacles is detected.
- Acoustic Detection. The obstacles are detected by checking APS_target_state which includes sound, sound direction, volume and break length.
- Optical Detection. This is mainly about HMI.

winIDEA Workspace Configuration

…. [Hide]

Scripts of Flashing

There are four scripts to implementing automatic flash: Winideaprepare.cs, WinideaFlash.cs, ActiveRun.cs and WinideaKeepRun.cs.

The first one is Winideaprepare.cs. The structure of the script is shown in Figure 9:

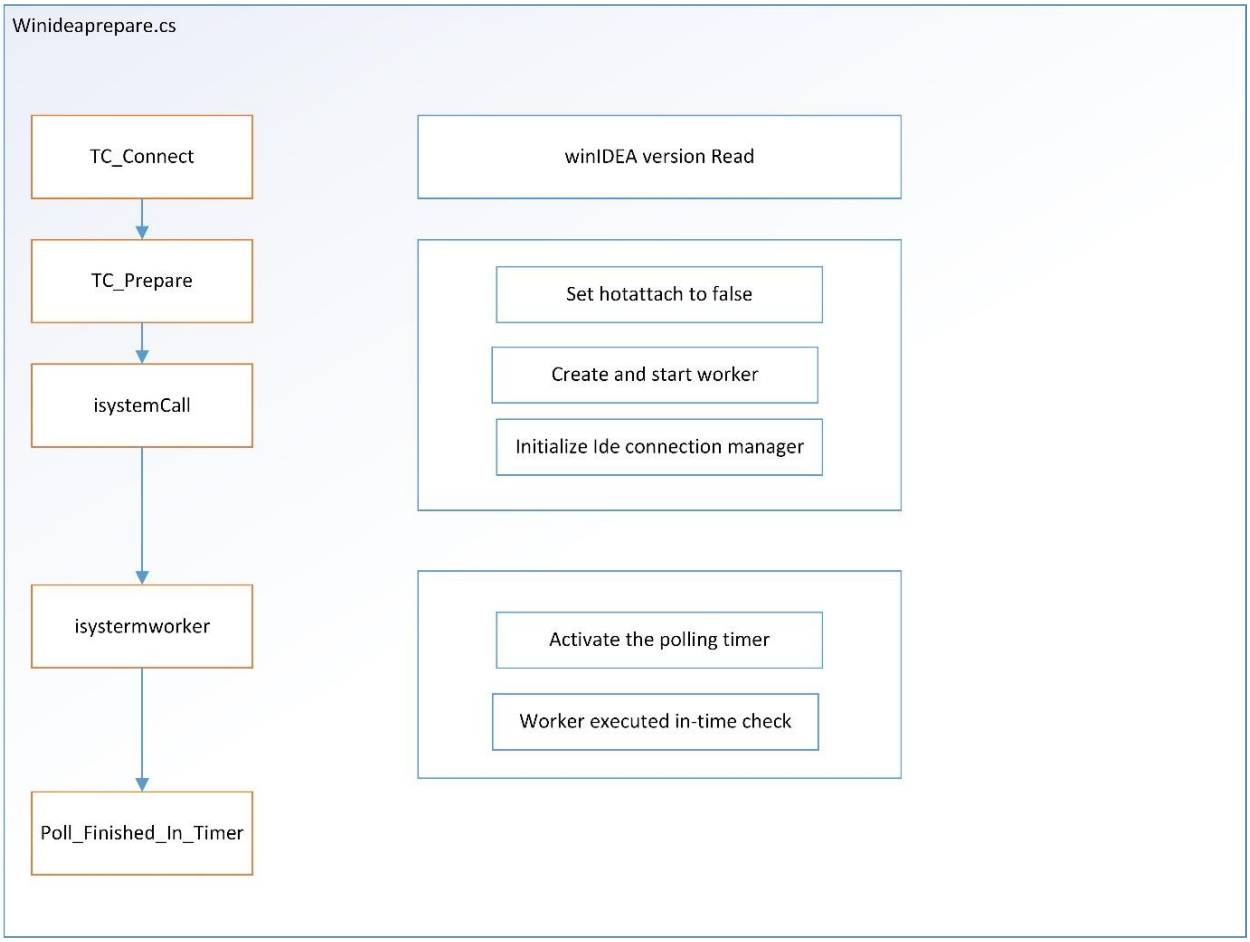

Figure 9 Script structure of Winideaprepare.cs

The second one is WinideaFlash.cs. The structure of the script is shown as Figure 10:

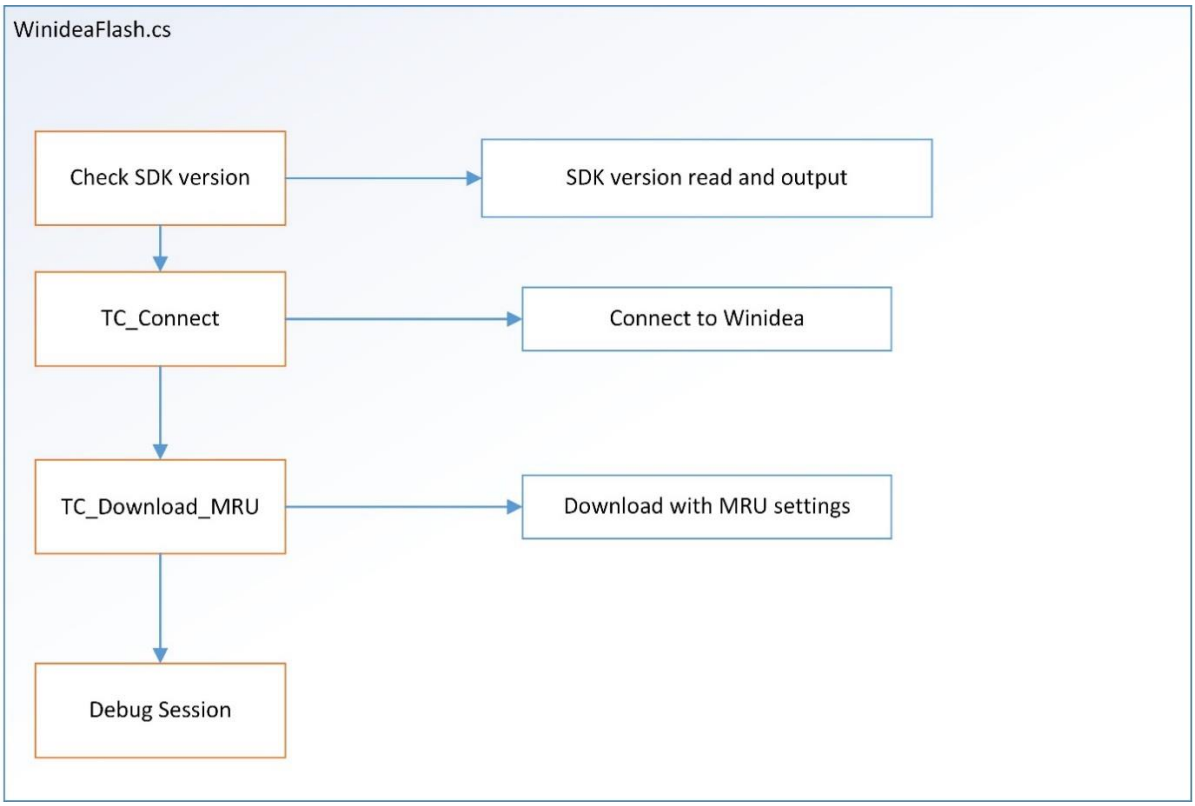

Figure 10 Script structure of WinideaFlash.cs

Connects to the most recently used winIDEA and opens the given workspace.

The third one is ActiveRun.cs. The structure of the script is shown in Figure 11:

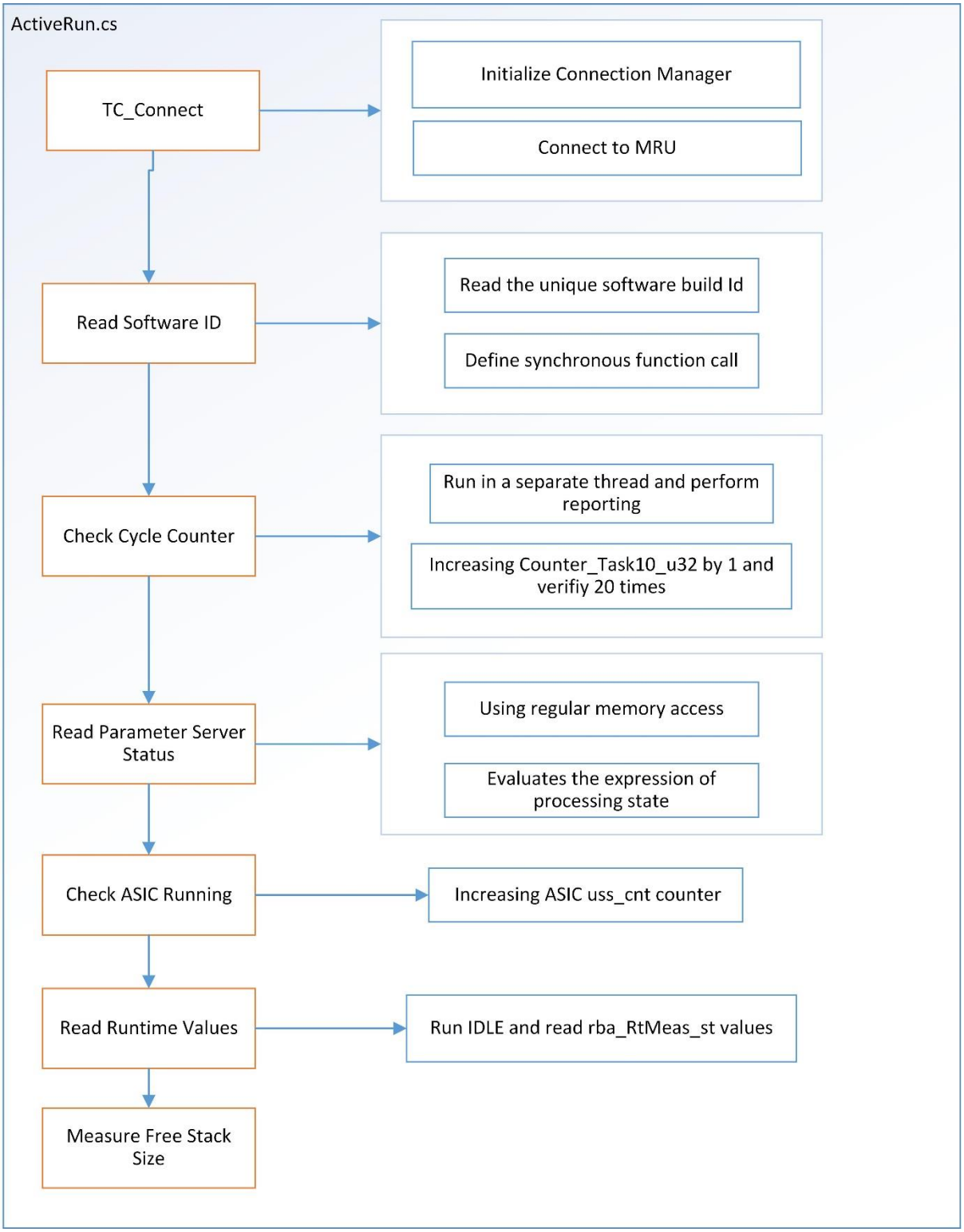

Figure 11 Script structure of ActiveRun.cs

The fourth one is WinideaKeepRun.cs. The structure of the script is shown in Figure 12:

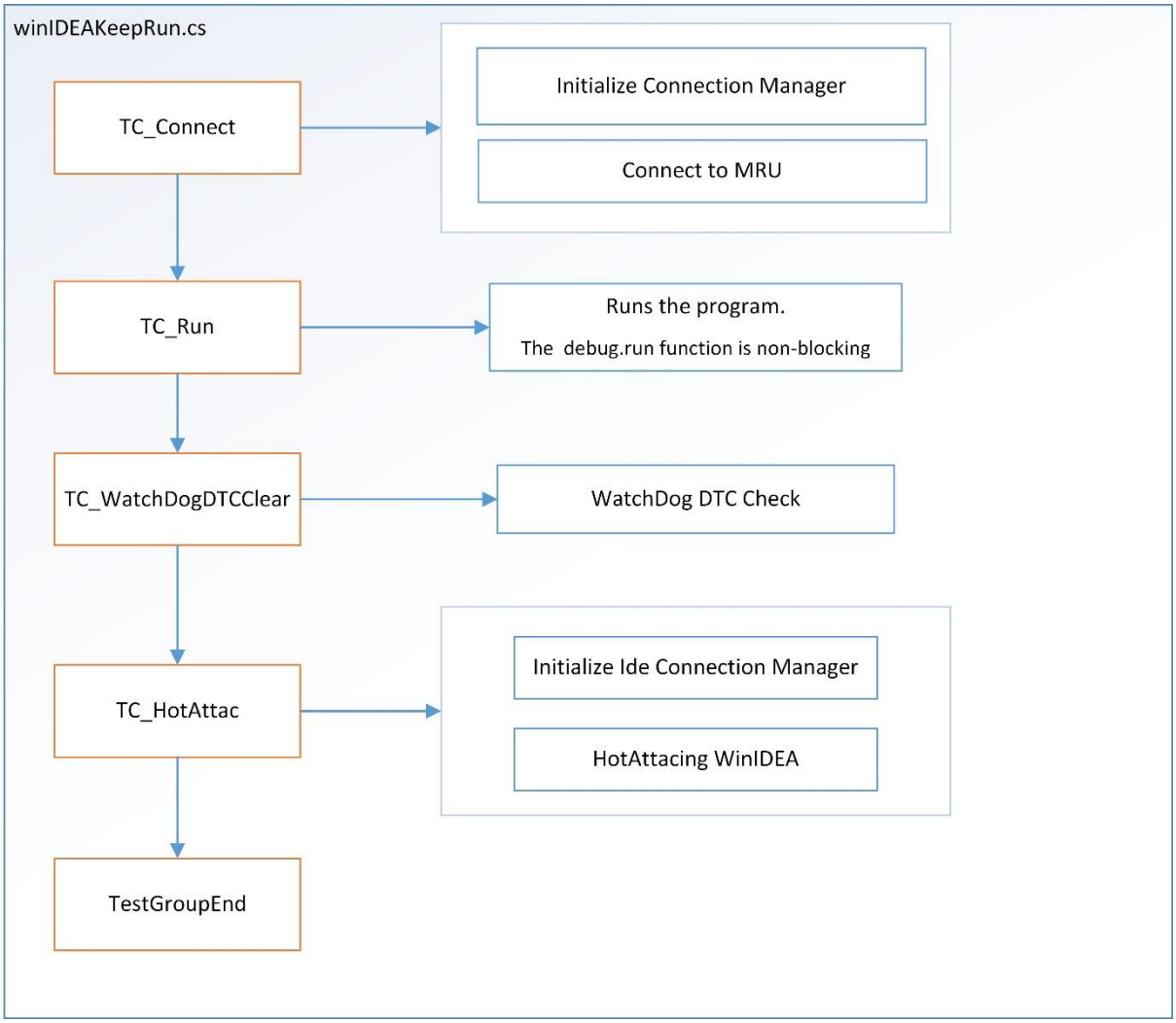

Figure 12 Script structure of WinideaKeepRun.cs

Automatically Execution

There are several possibilities for writing this script. One is to create a new *.vtuexe* file and convert these scripts into a CAPL form written in vTESTstudio. This is equivalent to having another test case, but this test case is not used to verify the software functionality. Thus, no extra steps are required except executing directly. The second one is writing another script and packaged it into executable file. The specified CANoe configuration file will started by launching and the scripts of this set up will be loaded and run.

The structure of this script shown as Figure 13:

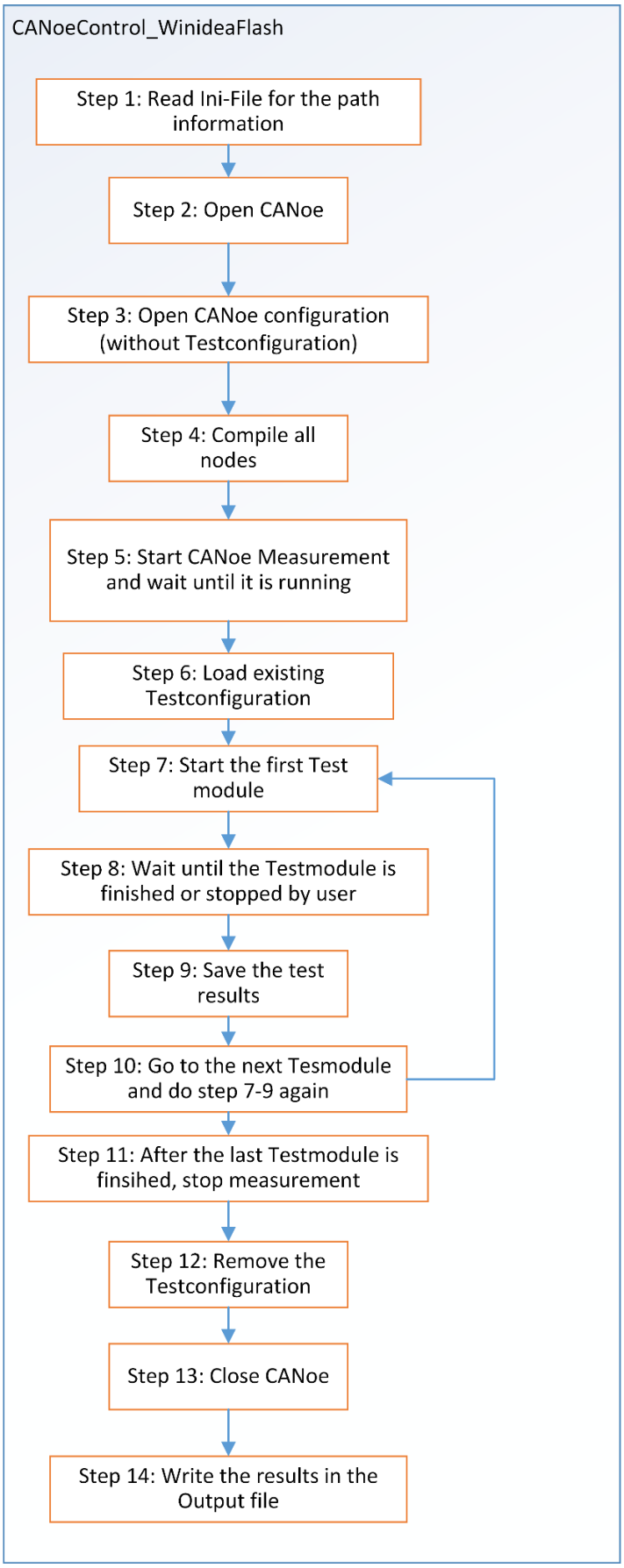

Figure 13 Script structure of CANoeControl_WinideaFlash

The structure of CANoeControl_Integration test shown as Figure 14:

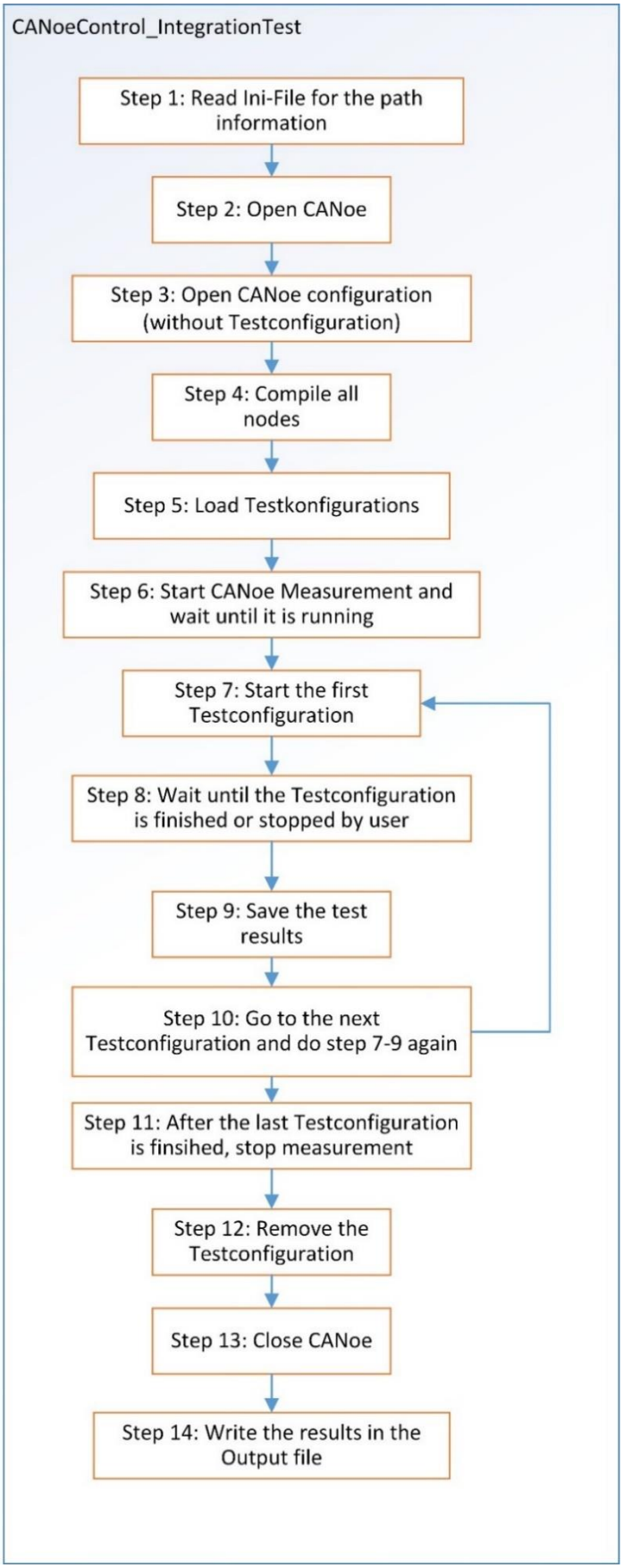

Figure 14 Script structure of CANoeControl_IntegrationTest

The configuration settings of this two scripts are saved in the same path as the scripts located. As shown below respectively:

Configuration settings of Winidea Flash:

CanoeControl.ini

// Path and name of Canoe Konfiguration file .cfg - CANoe v9.0 SP4

D:\MKS\RestbusSimulation\PPtest_Jenkins.cfg

*// Path and name of Testconfiguration file *.tse*

D:\MKS\TestEnv\Winidea_Tests\smoke_test\SmokeTest.tse

Configuration settings of Integration Test:

CanoeControl.ini

// Path and name of Canoe Konfiguration file .cfg - CANoe v8.5

D:\Sandbox\CANoeHIL_VTS\TestEnvironment\RestbusSimulation\ Full_8.5.cfg

*// Path and name of Testconfiguration file *.vtuexe - up to 17 *.vtuexe are possible - each line one vtuexe*

D:\MKS__test5\scripts\TestUnit_Busoff.vtuexe

Therefore, modify the above path is the only step that is needed when updating the integration test script.

1.1.1. Jenkins pipeline configuration

The relationship between the stages are shown in Figure 15. It can be observed in the stage view of Jenkins pipeline. The stages will execute from left to right in sequence. Obviously, there is a vacancy in this figure which is the integration test. The continuous test method will fill it.

Stage View

	Preparation	MKS Checkout	Build Premium	Build Comfort 8CH	Build Comfort 4CH	Integration-Test	MKS Checkpoint	Publish	Debug
Average stage times: (Average full run time: ~37min 27s)	548ms	7min 25s	11min 1s	8min 46s	8min 55s	NaNy NaNd	NaNy NaNd	1min 15s	NaNy NaNd
#320 Jan 05 22:17 No Changes	522ms	10min 38s	11min 55s	8min 50s	8min 57s			1min 15s	
#319 Jan 04 22:17 No Changes	501ms	6min 20s	10min 56s	8min 49s	8min 56s			1min 14s	
#318 Jan 03 22:17 No Changes	513ms	6min 42s	10min 57s	8min 48s	8min 56s			1min 15s	
#317 Jan 02 22:17 No Changes	461ms	6min 22s	10min 44s	8min 44s	9min 1s			1min 16s	
#316 Jan 01 22:17 No Changes	489ms	6min 12s	10min 55s	8min 43s	8min 56s			1min 14s	

Figure 15 Stages of Jenkins pipeline

There are two different ways to configure batch files on Jenkins. The first one is using *'execute windows batch command'* in Jenkins as shown in Figure 16.

The Second one is using Jenkins pipeline.

Figure 16 Execute windows batch command in Jenkins configuration

However, the Jenkins pipeline was already used in park pilot department so the method of executing windows batch command directly cannot be implement. For configurations on Jenkins, three batch files are in need. The Integration Test can only be generated when the Boolean *'Integration_Test'* is true. The file that composed this function is as shown in Figure 17.

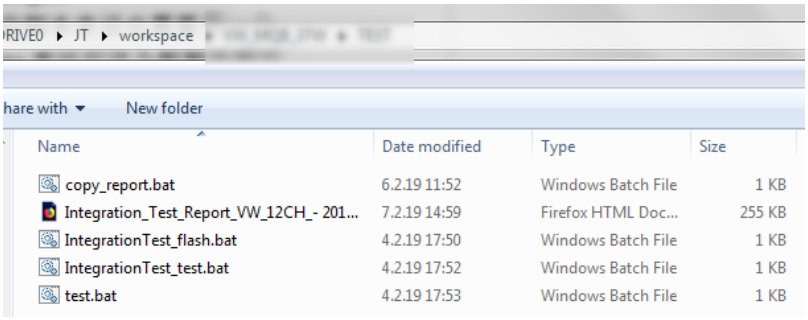

Figure 17 batch files for Jenkins configuration

And the contents of the batch files are given as follows:

IntegrationTest_flash.bat

```
echo open CANoe and execute test cases
rem execute .NET test cases
cd D:\MKS\test5\Jenkins_config\CANoeControl_WinideaFlash
CANoeControl.exe
```

IntegrationTest_test.bat

```
echo open CANoe and execute test cases
rem execute .NET test cases
cd D:\MKS\test5\Jenkins_config\CANoeControl_Integration_Test
```

```
CANoeControl.exe
```

copy_report.bat

```
@echo off
for /f "delims=" %%a in ('wmic OS Get localdatetime  ^| find "."') do set
dt=%%a
set YYYY=%dt:~0,4%
set MM=%dt:~4,2%
set DD=%dt:~6,2%
set HH=%dt:~8,2%
set Min=%dt:~10,2%
set Sec=%dt:~12,2%

set stamp=%YYYY%-%MM%-%DD%_%HH%-%Min%-%Sec%
if exist **.html (
      del /q **.html
)
copy
"D:\Sandbox\CANoeHIL\TestEnvironment\RestbusSimulation\Report_TestConfigu
ration.html" "D:\JT\workspace\ TEST\Integration_Test_Report_- %stamp%..html"
```

Obviously the first and second batch files are used to load the specific workspace path to execute the exe file.

The *copy_report* batch file is used to copy the test report which automatically generated by CANoe when test finished to the current Jenkins workspace path and rename it with timestamp. This is in preparation for sending the test report via email.

In the Jenkins pipeline, the following script was added:

```
// ###################################################################
//     INTEGRATION TEST
// ###################################################################
stage('Integration-Test'){
when {
      expression {
            return "${DEBUG_STAGE_ONLY}" == "false" &&
"${INTEGRATION_TEST}" == "true" ;
      }
 }
```

```
    steps {
        echo "########## INTEGRATION TEST - winIDEA Flash ##########"
        script {
            try {
                bat '%WORKSPACE%\\IntegrationTest_flash.bat'
                echo "########## INTEGRATION TEST - Testing ##########"
                bat '%WORKSPACE%\\IntegrationTest_test.bat'
                echo    "########## INTEGRATION TEST - report mail
##########"
                    emailext attachmentsPattern: '**.html', body:
"""<b>Integration Test</b><p>Finished : Job '${env.JOB_NAME}
[${env.BUILD_NUMBER}]' passed :</p><p>Please check console output at "<a
href="${env.BUILD_URL}">${env.JOB_NAME}
[${env.BUILD_NUMBER}]</a>"</p>""", charset:'UTF-8', mimeType:
'text/html',subject: """PASS: Project -> '${env.JOB_NAME}'""",to: 'fixed-
term.Yu.Zhong@de.bosch.com'
            }
            catch(err){
                echo    "########## INTEGRATION TEST - report mail
##########"
                bat '%WORKSPACE%\\copy_report.bat'
                emailext attachmentsPattern: '**.html', body:
"""<b>Integration Test</b><p>Finished : Job '${env.JOB_NAME}
[${env.BUILD_NUMBER}]' failed :</p><p>Please check console output at "<a
href="${env.BUILD_URL}">${env.JOB_NAME}
[${env.BUILD_NUMBER}]</a>"</p>""", charset:'UTF-8', mimeType:
'text/html',subject: """FAIL: Project -> '${env.JOB_NAME}'""",to: 'fixed-
term.Yu.Zhong@de.bosch.com'
            }
        }
    }
}
```

Assuredly, the process of changing the configuration which mentioned in section 5.3 can also be changed in Jenkins rather than change the local files in test worker PC.

```
echo open CANoe and execute test cases
rem execute .NET test cases
cd . \JENKINS\Bench_IntegrationTest\CANoeControl_WinideaFlash
```

```
    if exist CanoeControl.ini (
        del /q CanoeControl.ini
        echo File [CanoeControl.ini] Deleted Successfully
    )
    rem write CanoeControl.ini newly for Jenkins Worker
    echo // Path and name of Canoe Konfiguration file    .cfg - CANoe v9.0 SP4>>
CanoeControl.ini
    echo D:\MKS\RestbusSimulation\PPtest_Jenkins.cfg>> CanoeControl.ini
    echo // Path and name of Testconfiguration file *.tse>> CanoeControl.ini
    echo D:\MKS\TestEnv\Winidea_Tests\smoke_test\SmokeTest.tse
    >> CanoeControl.ini
    rem CANoeControl.exe


    echo open CANoe and execute test cases
    rem execute .NET test cases
    cd .\PP_PF_SIP6\RBP\GEN6\TOOLS\RB_SIP\JENKINS\Bench_IntegrationTest\C
ANoeControl_Integration_Test

    if exist CanoeControl.ini (
        del /q CanoeControl.ini
        echo File [CanoeControl.ini] Deleted Successfully
    )
    rem write CanoeControl.ini newly for Jenkins Worker
    echo // Path and name of Canoe Konfiguration file    .cfg - CANoe v9.0 SP4>>
CanoeControl.ini
    echo D:\Sandbox\CANoeHIL_VTS\TestEnvironment\RestbusSimulation\
Full_8.5.cfg>> CanoeControl.ini

    echo //path and name of the basic test configuration *.vtuexe>>
CanoeControl.ini
    echo D:\MKS\__test5\scripts\TestUnit_Busoff.vtuexe >> CanoeControl.ini
    rem CANoeControl.exe
```

1.2.Continuous Integration Process

After the above chapters, the process of integration testing on Jenkins has been given by Figure 18：

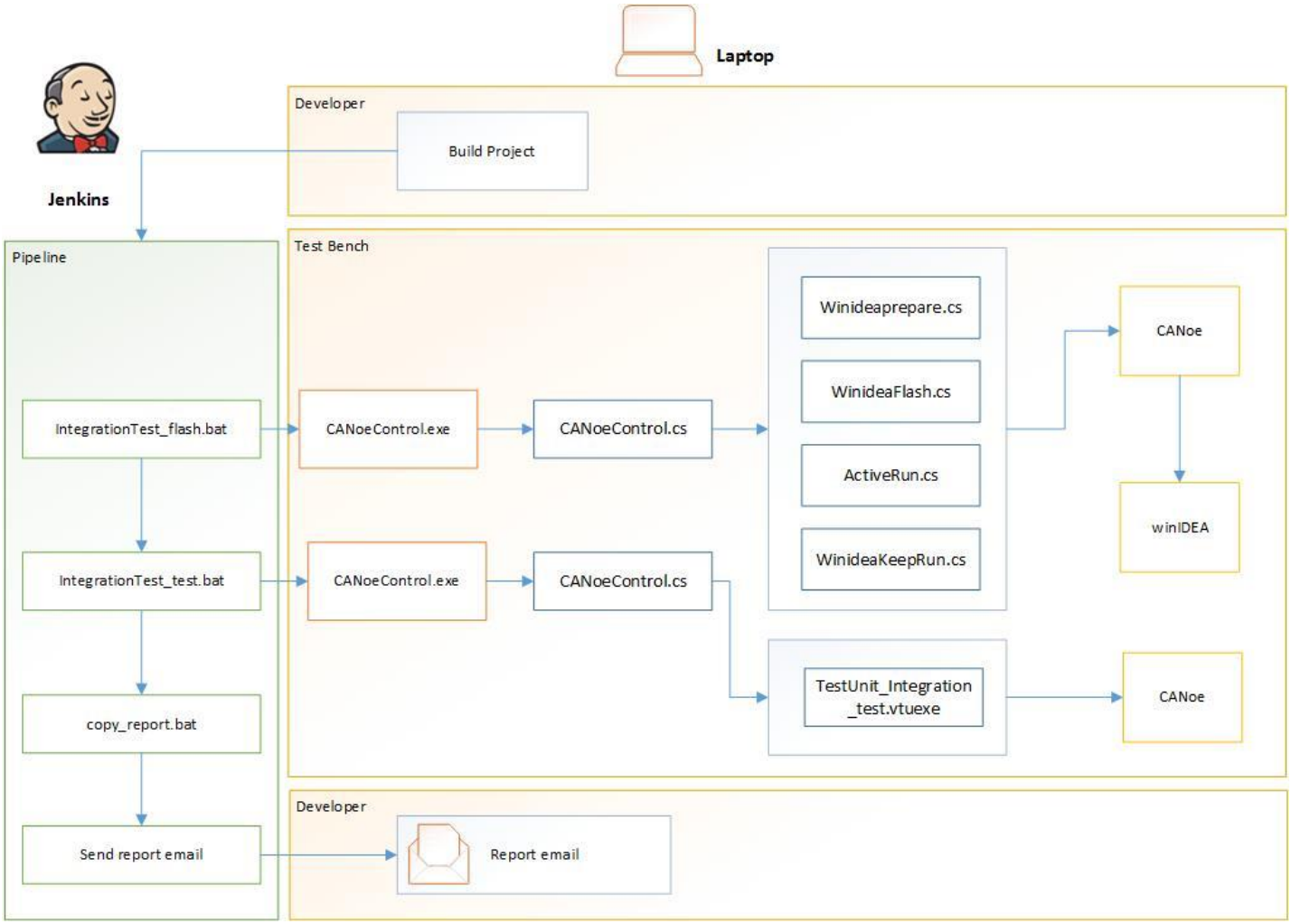

Figure 18 Jenkins pipeline structure

Obviously, Jenkins and test bench take over all the testing work in this process. The only thing that need developer to do is trigger integration test project with correct test node. The developer can continue their own development work during waiting the test report arrived. Thus, the developer can focus more on development rather than repeating complex tests. Thereby, the efficiency of the team is improving.

The mail obtained from Jenkins is shown in Figure 19. The attachment of the mail is a test report copied that from the CANoe test directory and renamed with an accurate timestamp which avoids the duplicate name of the test report.

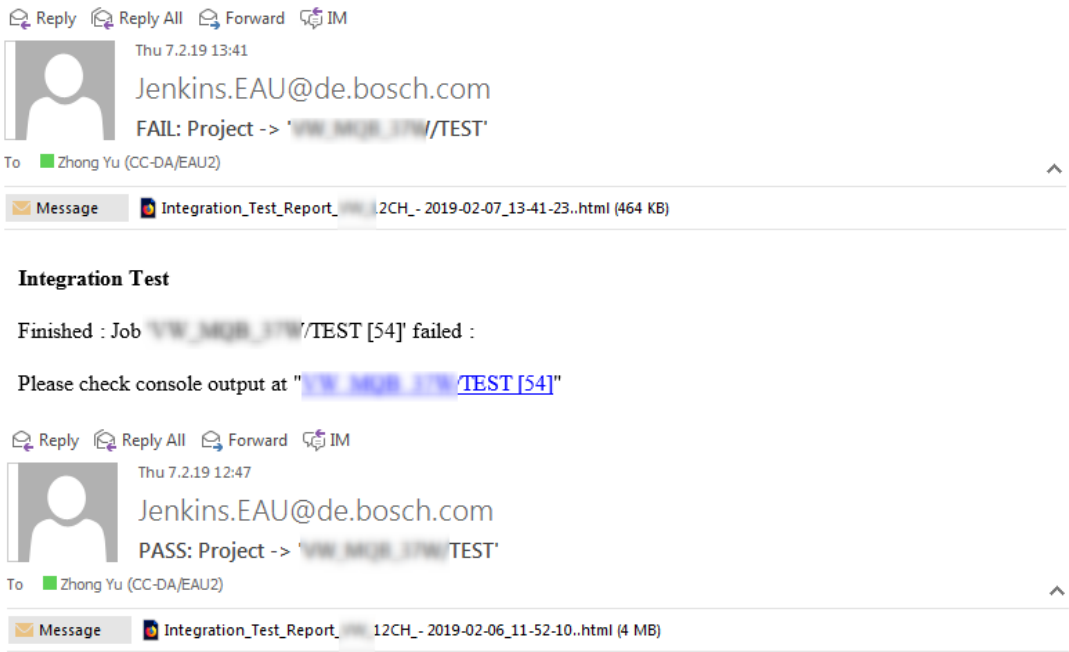

Figure 19 Test result mail