CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Test Automation for Automotive Embedded Systems

Master's thesis in Embedded Electronic System Design

DINGYUAN ZHENG, SHUYUE ZHANG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Test Automation for Automotive Embedded Systems

DINGYUAN ZHENG, SHUYUE ZHANG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Embedded Electronic System Design
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2017

Test Automation for Automotive Embedded Systems
DINGYUAN ZHENG, SHUYUE ZHANG

Supervisor: Jan Jonsson, Chalmers University of Technology
Examiner: Per Larsson-Edefors, Chalmers University of Technology

Master's Thesis 2017:01
Department of Computer Science and Engineering
Embedded Electronic System Design
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Test Automation for Automotive Embedded Systems
DINGYUAN ZHENG, SHUYUE ZHANG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With an increasing software complexity, it becomes a huge challenge to guarantee quality and reliability in automotive systems. However, the extensive use of manual testing is time consuming, costly and error prone, which leads to lack of product quality. Also it has a limited coverage due to its countable combinations. Thus, test automation for automotive systems has started to play a more important role nowadays. The main objective of this thesis work is to find out a automatic test framework for automotive systems, including information retrieval, test activity execution, result analysis and making decisions for next test phase. Interfaces that connect the above parts in the framework are built up to have communications and data exchange between isolated software systems. We show that this automated testing framework can realize different kinds of test case combinations and functionalities with dummy test cases.

Keywords: Test automation, software testing, automotive systems

Acknowledgements

The thesis becomes the reality with the support, encouragement and support by many individuals.
Firstly, we want to express my sincere gratitude to our supervisor Prof. Jan Jonsson in Chalmers for his unwavering support, patience and enthusiasm during the thesis project. He provides us so many valuable advices on how to polish our writing.
Besides, we would like to extend my gratitude to our supervisor in Volvo Car Group, Mattias Hillhammar and Ehsan Zaeimzadeh for offering us the opportunities to work with this topic and leading us on the project when we get stuck with their patience.

DINGYUAN ZHENG, SHUYUE ZHANG, Gothenburg, 01 2017

Contents

List of Figures

1

Introduction

With an increasing software complexity it has become important to guarantee quality and reliability in automotive systems. This has led to the need for systematic software testing. In most automotive industries, software testing is implemented manually, which is time-consuming, costly and error-prone. Also, by manual testing, the combination of test cases that can be executed is very limited, causing warranty and maintenance problems. Therefore, test automation has started to play an important role in improving systematic software testing in industries.

1.1 Background

Volvo Cars Corporation(VCC) is a car manufacturer developing luxury cars and human-centric car technology. The section Switches and Comfort Electronics is responsible for the control systems in a car, such as Climate Control, Parking Climate, Seat-, Window-, Mirror- and Roof-Control, User Input and Interior Lights. The group Analysis and Verification is a part of this section and is mainly responsible for functional testing, acceptance testing and system verification of embedded systems or subsystems with the different technical aspects mentioned above. While manual testing is currently used for these systems, fully automated testing is necessary for reducing or eliminating the shortcomings of manual testing to achieve time- and cost-efficiency and high performance of testing process.

This master thesis project aims to create an automated testing framework for the control systems mentioned above. The framework should include fairly unlimited combinations of test cases in order to reduce warranty and maintenance problems. Some algorithms should be designed to take decisions for next testing phase according to previous testing results and should support different test types, such as smoke tests, regression tests, long-term tests and stress tests. Another purpose of this project is to create or provide different tools that could automatically review test scripts and analyze test results.

1.2 Motivation

In test automation, a software framework is designed to provide an optimized verification flow to control the execution of test activities and to compare actual outcomes with predicted outcomes. The software framework should be designed to automatically perform repetitive, but necessary, tasks or test activities that would be difficult to achieve in manual testing [1]. Large parts of existing test processes provide automated test execution and monitoring in practical software development. In contrast, most test cases are still generated by labor-intensive manual tasks [2]. Therefore, the automated generation of test cases is one major challenge. One approach to solve this problem is to utilize model-based testing using models to represent the desired behavior of a system under test (SUT) for test case generation [3]. In this master thesis project, several test types(such as smoke tests, regression tests, long-term tests and stress tests) will be generated using test automation. One direction of our focus is to find appropriate physical models suitable for the aforementioned test cases within automotive embedded systems.

Usually, the decision regarding how to test the system in a subsequent phase is made by a test engineer's experience, which leads to several problems: Firstly, limited test case combinations cannot give a comprehensive testing result, which could make test engineers take inappropriate decisions in some cases. Secondly, new test cases will be generated to test newly developed software; in this case, experience is not sufficient to make decision for future testing phase. Thirdly, entirely relying on human experience will lead to human errors, which could cost time and money. Therefore, this project should focus on generating unlimited combinations of test cases and designing algorithms that can make a decision according to previous testing results. A further complication is that testing tools used at VCC are mutually isolated, making automatic testing more difficult. Building communication interfaces between different tools is therefore another key point that we should focus on.

Embedded systems similar to the ones in this master thesis project have a complex construction that requires the co-design of software and hardware components in a testing development. Hence, it would be advantageous if flaws in the development or verification process could be revealed in earlier phases in order to reduce costs in the automotive industry [1]. Additionally, automated testing should provide the same, or even better, accuracy compared to the manual testing approach. The approaches should focus on providing an optimized work flow to detect problems in earlier verification stages and on achieving high accuracy and performance of test automation.

A testing system should also integrate function libraries, test data sources, object details and reusable modules. Many existing frameworks, such as Framework for Integrated Test, provide support for automated software testing [4]. Thus, the test automation framework/tool in this project is required to give an integrated testing system that could provide different test specifications based on test cases.

1.3 Project Goals

In this master thesis project, the following two overall goals should be achieved: (i) to create a test automation framework including unlimited test case combinations and algorithms that can make decision for the next testing phase, and (ii) to create a tool chain for executing test cases and analysing the test results in a systematic way automatically. The detailed goals of the project are as follows:

- Design an automatic testing framework and a tool chain including the following procedures: managing test requirements, review test cases, automatic testing process and test reports analysis.
- Add functionalities to make the testing framework more intelligent compared with manual testing. The functionalities to be added are: test loops, available time, random combinations.
- Design algorithms that support smoke test, regression test, long-term test and stress test. The algorithms should also provide suggestions for the next testing phase based on previous test results. (Optional)

1.4 Automatic Testing Framework Overview

The overview of the testing framework is shown in Figure 1.1. The framework should be able to perform within Hardware-in-Loop (HIL) environment [5].

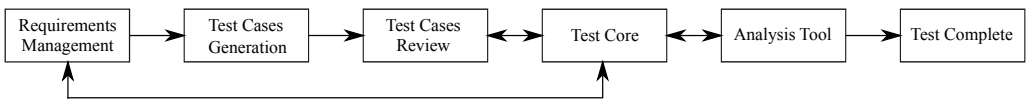


Figure 1.1: The overview of automatic testing framework

In *Requirements Management*, test specifications and parameters should be obtained from a database management system. *Elektra* is a software system commonly used at VCC to provide data management. This software system is an application for product life-cycle management, based on the *Vector Informatik GmbH*[1] product "eASEE Automotive Solution", and is developed by VCC with the cooperation of *Vector* [6]. *Elektra* provides a web application programming interface (API) by which web services could be implemented to acquire all the information about test specifications, test cases and test parameters from a VCC server.

After requirements are managed, test engineers generate test cases by creating sets of source code and storing them into a version and revision control system (*Test Cases Generation* step in Figure 1.1). Before executing a test case, the corresponding

[1]Vector Informatik GmbH is a company providing software tools, services and components for networking of embedded electronic systems. `https://www.vector.com`

source code should be reviewed automatically to check if it meets standard templates and manually to check whether the logic in the code is correct or not (*Test Cases Review* step in Figure 1.1).

When *Test Cases Review* is finished, those test cases will be executed in *Test Core*, the core part in this framework. Test Core should be able to select test cases, decide execution sequence order, execute test cases and generate test reports. In addition, some functionalities, such as random combination, test loops, available time and etc., should be added into Test Core to make the framework more intelligent. Moreover, the Test Core should be compatible with automotive systems.

After test reports are generated, an *Analysis Tool* should analyze the reports and generate a result or conclusion automatically. In this step, some algorithms should be designed to help a user understand the result or conclusion and make a decision for the next testing phase.

Finally, when the entire testing event is complete, a final report should be generated and sent to Team Center, which is the step *Test Complete* in Figure 1.1.

1.5 Challenges

In accordance to the introduction from previous sections, several challenges will be addressed in this project.

Firstly, although some industries have made some progress on test automation, fully automatic testing with artificial intelligence is still a difficult field. Hence, how to build such test framework is one of challenges in this project.

Secondly, with an increase in number of test cases, random combinations could lead to an intractable number of tests. How to control the number of combinations in order to avoid breaking the HIL rig is therefore another challenge.

Finally, since the algorithms need to substitute human experience, how to design algorithms to provide high accuracy is also a challenge.

1.6 Delimitation

The HIL Software/Hardware framework and the version control system SVN are currently tools and equipment used by the group. This should be considered as fixed infrastructures during the thesis work, which means no changes can be made in these parts.

1.7 Overview

The rest of the report is organized as follows: Chapter 2 introduces the principles that will be applied in the project. Next, Chapter 3 compares different methodologies that have been considered for the entire framework as well as for each part of the framework. In Chapter 4, detailed implementations of the framework are described. Chapter 5 shows some preliminary results of the basic framework with simple functionalities implemented. Chapter 6 discusses the features and the merits of the automatic testing framework and the improvements that could be made in the future development. Finally, conclusions are given in Chapter 7.

2

Theory

In order to understand how the goals mentioned in Chapter 1 are fulfilled, it is necessary to get familiar with the principles of software testing. The project aims to improve software testing, thus the principles of this domain are essential to study. The basics of software testing is described in Section 2.1. Apart from software testing, other concepts are also inevitable to be studied. For example, in Requirement Management, Elektra is used to manage test specifications and offers Web API. Web service can be utilized to get test specifications by making requests to the VCC server via the defined API. Web service will be introduced in Section 2.2. After Test Cases Generation, a software version and revision control system is required to store the created test cases and to perform Test Cases Review. These parts are introduced in Section 2.3. Finally, the framework should be compatible with the automotive systems. Hence, electronic control units (ECUs) and bus systems are fundamentals that will be described in Section 2.4.

2.1 Software Testing

Different definitions of existing software testing highlight various aspects of such testing. Firstly, software testing is used to evaluate attributes of the system and check whether the designed system meets specific requirements. This kind of software testing is deemed as positive testing, which primarily pays attention to the requirement [7]. Secondly, software testing is defined as the process of finding defects by executing the system. This kind of software testing is considered as negative testing, which aims at looking for defects besides the discrepancy of requirements [8]. Thirdly, software testing is the process where we could check the status of benefits and the risk associated with the release of software systems. This kind of software testing is used to reduce or migrate the risk of failure of the system, keeping system more robust and reliable [9]. In practice, software testing usually combines positive and negative testing. That is to say, software testing is used to not only check whether a system fulfills all requirements but also find errors which will reduce the robustness of the system [10].

Software testing is quite important in the industry since the price of not testing may be a failure of the system. Consequently, customers may lose confidence in the

company, which will lead to further loss of profit [10].

Four properties could be used to judge the quality of test cases which are deemed as a critical element to show how good this testing is. These four properties are its defect detection effectiveness, cost, maintenance effort and exemplariness. Specifically, defect detection effectiveness means whether it is likely to find defects or not. Cost here shows how economical the test case is to perform, analyze and debug. Maintenance effort illustrates how much effort is needed to keep executing testing, especially when the software changes. Exemplariness shows the coverage of one test case. Basically, an exemplary test case could test more than one thing, thus reducing the total number of test cases required. This will help further reduce the cost in the testing. Different levels of software testing and its corresponding testing techniques are described in the following subsection.

2.1.1 Different levels of software testing

Unit testing is used to identify errors in the program logic. It is a process of testing individual units. The unit here could be a predefined reusable component, such as sub-programmes or sub-routine [8]. The aim of unit testing is to check whether expected results for each state could be achieved after testing. *Integration testing* is used to check the interface among separately test units and to show the structural relationship of the system with respect to its units. Three basic integration structures of functional tree; top-down, bottom-up and sandwich can be used to describe the order how units are integrated [11]. For unit and integration testing, *functional testing* is used as testing technique to check the functionality.

System testing aims to evaluate whether the outcome of a product or system fulfills our expectation, that is, to demonstrate correct behaviors. In this master thesis project, *System Testing* is focused on designing an automatic testing framework. For *System Testing*, *Model-based testing (MBT)* is an appropriate technique for test automation on system level [11]. It will be described in details in the next subsection.

2.1.2 Model-Based Testing

Model-based testing (MBT) is a relatively new approach to software testing that extends test automation from test execution to design testing using automatic test case generation from models [12]. GOTCHA-TCBeans, mbt, MOTES, TestOptimal, AGEDIS, ParTeG, Qtronic, Test Designer and Spec Explorer are examples of commonly-used tools for MBT [13, 14, 15]. In the following, we will compare the pros and cons of these tools and assess how well suited they are for test automation. The tool Spec Explorer does not provide GUI support to select test cases, which seems not so flexible when connecting to other tools, increasing the risk of unreliable integration. For GOTCHA-TCBeans, input values could only be selected manually by testers to create the adapter. Usually, this kind of adapter is used to generate

pieces of code automatically [13]. Thus, GOTCHA-TCBeans is not suitable to work as a tool in test automation.

Moreover, most tools, including GOTCHA-TCBeans, mbt, MOTES and ParTeG, only have partial support for model creation. That is to say, in order for these MBT tools to be extended to other tools, newly-created models have to be imported manually, which increases the difficulty of test automation. Hence, such interface between MBT tool and other tools may introduce problems, which will reduce the reliability and robustness of the MBT. With respect to model verification, GOTCHA-TCBeans, AGEDIS and Spec Explorer do not support test case debugging [16]. Thus, some other software should be introduced to reduce the risk of not finding defects of MBT tools.

With respect to model verdicts, ParTeG and Test Designer are not good choices for test automation, since they do not provide the verdict of executing test cases. Verdict here means the result of test cases, pass, warning or fail. This involves a difficulty in analyzing test results and deciding the test cycle for the next phase [13]. Qtronic does not support offline testing, which means that the tool does not generate test cases as human-readable assets that can later assist in manual testing. Meanwhile, online testing of Qtronic requires an extended test execution engine through a DLL plug-in [17]. This increases the risk of test failure since it will increase the difficulty of integration associated with different tools.

In summary, we have found that most MBT tools in the industry cannot be used in a fully automated testing system because they lack support for extracting information, executing test cases, analyzing test results or determining how to test for the next phase.

2.1.3 Test Automation

Test automation is a special form of testing. It has a special software, which is considered as the control to the test case execution, the comparison of actual outcomes with predicted ones, and test analysis. The advantage to use the test automation is to reduce the cost in the long term, get larger coverage during the same or shorter time and reuse the resource more efficiently. Moreover, whether the test automation is good or bad relies a lot on the selection of test cases. As described before, four properties are used to identify good test cases. Cost and maintenance effort are two main features that should be focused on when selecting test cases in test automation. This will help users to decide when test automation could be a good choice. Usually test automation is not suitable for tasks with little repetition such as late development verification. It is more likely to apply test automation for repetitive tasks such as unit testing or regression testing considering the fact that it is quite expensive to build up or maintain a test automation framework. Thus, it is much better to consider long term use for test automation in advance.

2.2 Web Service

Before introducing Web service technology, we should define the concept of service-oriented architecture (SOA). SOA is an abstract architectural style that is able to build software systems with loosely coupled and dynamically bound services. The basic principle of SOA is centered around three aspects: 1) abstract definitions of services should be provided, including the detailed appropriate approach to binding the service dynamically, 2) details of services need to be published in order to allow users to understand how they can obtain the desired information, 3) users need to have some approaches to finding what services are available and can meet their needs [18]. Loose coupling is the main feature of SOA. The definition of loose coupling is that services or components are within a relationship with minimum dependencies and retain an awareness of each other [19]. Also, the services should be described in an uniform way and it should be possible for the services to be discovered and composed.

Web service is a virtualized technology that offers services and is therefore an important approach for realizing an SOA [18]. The World Wide Web Consortium (W3C) defines Web services as systems designed to support interoperability via machine-to-machine interaction over a network. It also has an interface described in a machine-processable format. Other systems can interact with the Web service by conveying messages using transport protocols with machine-readable languages in conjunction with other Web-related standards. Web service technology is basically composed of several loosely coupled components, including transport services, messaging services, service description, discovery services, service security, reliable messaging, transactions, service composition and payload format [20].

In this report, two of the most prevalent Web service technologies, "Big" and Representation State Transfer (REST), will be introduced. "Big" Web service is a traditional solution that is frequently used currently, and is a combination of different techniques for the loosely coupled components aforementioned. REST is defined as an architectural style for building large-scale distributed network systems [21]. It is currently the recommended method of Web service. The two Web services are further described in the following sections.

2.2.1 "Big" Web Services

The "Big" Web services technology is a widely used solution for designing applications over a network. The components that are elementary to compose an operative Web service will be introduced in the following sections.

2.2.1.1 Transport Protocol

For "Big" Web services, message transport technologies constitute the foundation of an interoperable messaging architecture that are the basic abstract architecture of Web services. Normally, HyperText Transport Protocol (HTTP) and Secure HTTP (HTTPS) can cover most support for transport protocols in World Wide Web, but other transport protocols can also be used to transmit messages. The supported protocols in "Big" Web services are Transmission Control Protocol (TCP), Simple Mail Transfer Protocol (SMTP), Java Message Service (JMS), IBM Message Queue (MQ), Blocks Extensible Exchange Protocol (BEEP) and Internet Inter-Orb Protocol (IIOP) [22]. As HTTP and HTTPS are generally employed in the majority of Web service applications or systems, other communication protocols will not be described. Transport protocols are fundamental for acquiring a larger scope of interoperability, but details behind those protocols are hidden from the design of Web services [18].

2.2.1.2 Messaging services

Messaging services is another foundation component in forming "Big" Web services specifications and technologies. For most of Web service applications, Simple Object Access Protocol (SOAP) is the ubiquitous usage for messaging. SOAP gives a simple mechanism for exchanging structured information between components or services. It can reduce the complexity of integrating applications or systems built on different platforms. In other words, SOAP messaging can create a communication channel for transmitting information between services on different platforms [23]. A SOAP message is represented in the eXtensible Markup Language (XML) format which is a metalanguage for defining new languages, which is platform-independent and does not use Unicode for definition [24]. XML Schema is another data representation to define specification and documents developed by W3C. It uses XML syntax and is capable of elaborating data types and give a preferable structure. XML or XML Schema is commonly known as payload format in Web services[25].

A SOAP message is an XML document containing three elements: an envelope, a header and a body. An envelope is the root element that consists of a header and a mandatory body. A header can extend features for SOAP [23].

2.2.1.3 Service Identification

The exchanging messages need to be identified both by senders and receivers. Two techniques, uniform resource identifiers (URIs) and Web service addressing (WS-Addressing), are widely applied in message identification.

A URI is a string of characters for identifying an abstract or concrete resource (message). It can enable interaction with representations of a resource over a network

via specific protocols. Uniform Resource Locators (URLs) are a universal form of URIs and are typically used for referring to a web address.

WS-addressing is an interoperable and transport-independent solution for identifying resources or messages. It segregates the address information from a transport protocol but allows Web service applications to communicate with address information directly, which identifies Web services endpoints and secures end-to-end endpoint identification in messages. This ultimately enables messaging systems to support message transmission through networks [18]. WS-addressing is a structure for conveying a reference to a Web service endpoint and a set of message addressing properties combining addressing information with a specific message. Thus, WS-addressing normalizes the corresponding information into a uniform format handled independently in Web service applications [26].

2.2.1.4 Service Description

Metadata is normally defined to fully describe the characteristics of services deployed on a network, and is essential to manage loose coupling. Web Service Description Language (WSDL) is the most widespread technique of metadata. WSDL offers the possibility for developers to depict functional features of a Web service [18].

WSDL uses the XML format to define a set of endpoints operating either document-oriented or procedure-oriented messages. A WSDL document normally contains two parts: abstract definitions and concrete descriptions. The first part defines SOAP messages with a language- and platform-independent manner, whereas the second part can provide serialization [27].

WSDL offers a standard, language-independent view of services for clients, and therefore constitutes a future-oriented method for applications and services. Moreover, interoperability across diverse programming paradigms is allowed [28].

2.2.2 REST

REST was initially defined as an architectural style for networking with several constraints, such as client-server communication, statelessness, cacheability, uniform interface, layered system and code-on-demand [29, 30]. A resource and the representation of this resource is the most important foundation of REST. A resource could be a product, a document, a homepage or anything that is an abstraction of information in a server. The resource can be represented by a document with a specific format, a library image and so forth, which is defined as the representation of the resource [31]. Resources can be identified by URIs over two transport protocols: HTTP and HTTPS. In other word, a REST Web services can be simply designed by combining HTTP or HTTPS protocols and URIs.

A representation is a sequence of bytes and the format of the metadata describing

those bytes [29]. Several formats are supported in REST to represent resources. Apart from XML, REST also supports JavaScript Object Notation (JSON), Multipurpose Internet Mail Exchange (MIME) and YAML [21]. This allows developers to have more choices in order to select a format suitable to a particular application.

REST has two main features. One feature is *statelessness*, meaning that the server does not need to care about which state a client is in, and vice versa. This improves reliability and scalability because the recovery process in case of failures is simplified and the memory consumption can be reduced without persisting state [18]. The other feature is *uniform interface*. This interface basically supports create, retrieve, update and delete methods [18, 21, 30]. These methods correspond to the HTTP methods: `GET`, `POST`, `PUT`, `DELETE`, which greatly simplifies the interface design. RESTful Web service has been currently supported by various programming language like Python and Java, and is considered to be a simple but powerful approach to Web services design [21].

2.3 Software Version and Revision Control System

A version control system is a software package which can help trace file documentations after every change by using different tags. Moreover, such tags could be organized to identify into different levels of changes, providing opportunity to revisit such tagged stages when needed. Rather than simply overwriting a changed file or storing different versions of the same file under different file names, version control system can simplify the work flow of changing file documentation. Tracing back to the old version becomes much easier and time-efficient in this system.

A version control system can be classified based on its operation mode – local version control system, centralized version control system (CVCS) and distributed version control system (DVCS) [32].

2.3.1 Centralized version control system

CVCS holds a central server connected with different clients, which means that all files that are stored in that specific server can be shared. More specifically, each client in such control system can request the latest or any specific version from the server and make a working copy on their local machine through the command "`checkout`". Moreover, each client could push their latest changes to create a new version on the server through the command "`checkin`" [33]. The advantage of CVCS is that it is very easy to trace the history of changes on a central server, which leads to a direct top-down management of teamwork and a straight-forward back-up of data, especially for a large repository with huge amounts of files. In addition, CVCS allows partial checkout of a sub-tree from the repository tree which

provides flexibility. Moreover, CVCS is highly integrated with other tools and quite mature, thus different software can be used with CVCS. However, CVCS has also its disadvantages. For example, there is always a risk to get in trouble when the centralized server is corrupted, meaning that no information stored in the server can be accessed. Moreover, even though some software versions of CVCS have applied a copy-modify-merge model instead of lock-modify-unlock model to reduce the risk of overwriting, it still becomes more complicated to complete merge operation compared to the automatic merges used in DVCS.

Subversion (SVN), as a centralized system for collaborative editing and sharing of information, is one type of version control software. A repository is the core of SVN, which works as the central store of data with a typical hierarchy of files and directories. Any client who connects to this repository is able to read or write data. One of the advantages of SVN is that it is possible to view a previous version of the file-system if a specific parameter (e.g. revision number, date, author) is provided [34]. Since SVN can track all changes in the data stored over time, it is very efficient when comparisons among different version are needed.

Moreover, due to the copy-modify-merge model of CVCS, the risk of overwriting the latest change by another client occasionally is reduced. In this model, every client should create his/her own working copy. Private changes in the working copy are then merged into a new and final version in SVN repository. The advantage of this model is that it can ensure that different clients can work in parallel, without waiting for one another [34].

2.3.2 Distributed version control system

DVCS, being a hybrid system of local version control system and centralized version control system, has its own advantage. Not only can it avoid the high risk of losing the entire history of files caused by corrupted server in centralized version control system, but can also provide opportunity to work collaboratively on the same project since it does not rely on a user's individual machine as is done in a local version control system [33]. Moreover, DVCS provides a higher reliability since partial operations will never occur in DVCS, which will reduce the risk of data loss. Additionally, DVCS has a better performance due to its special mechanics where it takes snapshots of entire sets of files instead of storing difference of each file, which will increase the speed of DVCS system. However, DVCS still has some disadvantages. For example, it stores the entire repository in each user machine locally and shares information among different users connected to that server by transferring local changes. This approach of storing the entire repository may lead to a problem of time inefficiency when the repository is huge. It is particularly inconvenient for a large company to use DVCS if too much information is stored in the repository. Moreover, DVCS does not use revision numbers, which means that traces of the latest file become difficult to generate [35].

Git is a typical DCVS software with properties of high speed and good support

for distributed, non-linear work flow [33]. Non-linear work flow provides improved support for merging and branching, which will encourage local users to keep their branches up-to-date with the mainline, and reduce the risk of their branches becoming out-of-date. Moreover, such non-linear work flow provides improvement on merging, further reducing the burden of space.

2.4 Automotive Embedded System

An automotive embedded system is a very complicated system, yet tremendous progress has been achieved in vehicle industries in recent decades. Not only has hardware been improved, but software has been applied in a vehicle to create many functionalities that are beneficial for the human user and that helps control electronics components such as sensors and actuators. Hence, software development has played an important role in vehicle design.

All software are executed by ECUs. An ECU is an embedded electronic device that controls various electronic systems or subsystems in a vehicle. The control device and its electronic components are located on the hardware that contains a micro-controller and memories. The control software, stored in memories and performed in the micro-controller, is lower-level programming code that can read signals from sensors, evaluate signals and react on a set of events based on received signals in actuators. Consequently, the ECUs act as digital computers in vehicles [36, 37].

There are a variety of types of ECU, including Engine Control Module (ECM), Electronic Brake Control Module (EBCM), Vehicle Control Module (VCM), Body Control Module (BCM), Powertrain Control Module (PCM), Transmission Control Module (TCM), Central Control Module (CCM), Central Timing Module (CTM), General Electronic Module (GEM), etc [36]. In modern vehicles, more than 100 ECUs are distributed in the different parts of the vehicle and the complexity and sophistication of software in ECUs is expected to increase [38].

2.4.1 Bus System

As ECUs are distributed over an entire vehicle, a network should be used to enable communications among different ECUs. Several network topologies are widely used nowadays, for example, bus topology, star topology, ring topology, mesh topology and hybrid topology. Among these, the bus topology is one of the most widespread approach in the automotive industry.

The core element of the bus topology is a single and linear bus that connects with different nodes via short cables. Then nodes connect with distributed subsystems or components within an entire system. Nodes receive and send messages across the network. If one node fails, the messages on this node will not available to

other nodes anymore. However, the remaining nodes can still exchange information. Nevertheless, if the core bus fails, the entire bus system will not work [37].

The Controller Area Network (CAN) bus is widely employed in the vehicle industry. Other bus systems, such as Local Interconnect Network (LIN) bus, Media Oriented systems Transport (MOST) bus and FlexRay bus, are also commonly used in modern automotive systems [37].

Currently, various tools simulating bus systems have been developed to allow for the design and verification of vehicle systems. CANoe and CANalyzer, both developed by Vector, are two of the most prevalent tools as they can simulate most types of bus systems [39]. They both use CAPL programming language to generate software or test cases. CAPL programming is similar to C programming but supports limited functions compared to C or C++.

3

Methodology

Based on the theory described in Chapter 2, methodologies for the whole system associated with detailed sub-systems will be introduced in this chapter. We start by discussing how to choose the programming language which is used to design the whole system.

Based on the pre-study phase performed early in the project, Python programming and CAPL programming are two alternatives that can be used to build a framework. Python is a high-level, interactive, interpreted, object-oriented programming language. It is simple and powerful and there are many third-party modules as open source that can provide helpful functions [40, 41]. CAPL is a programming language, similar to C programming, that has been developed by Vector. This language is commonly used in bus system applications such as CANoe and CANalyzer and provides many functions specific for automotive system testing [42]. These two programming languages are both suitable for this project. Three approaches can now be considered, as shown in Figure 3.1.

The first approach (Figure 3.1a) is to use CAPL programming to design the framework. CAPL is powerful at CAN networking and provides many functions for message communication between different buses, which is beneficial for automotive systems. Moreover, CANoe or CANalyzer are tools that are commonly used at VCC, thus no third party tools need to be learned by test engineers. However, there are many functions that are supported in C programming but not supported in CAPL programming, which makes other parts of framework difficult to design.

The second approach (Figure 3.1b) is to use Python programming to design the entire framework. COM server could be applied to develop an automatic testing framework using Python programming. Hence, we could produce our own application to perform test automation, which will give the possibility to limit the maintenance cost. Nevertheless, using COM server means that the controlling of test cases should be programmed in Python. In this case, all source code of test cases also need to be modified, meaning no reuse of generated test cases.

The third approach (Figure 3.1c) is to combine CAPL and Python to design the framework, that is, using CAPL to control test cases in Test Core and using Python to achieve other functionalities, such as *Analysis Tool*, Web Service for *Requirements Management*. This approach could both reuse the generated test cases and make

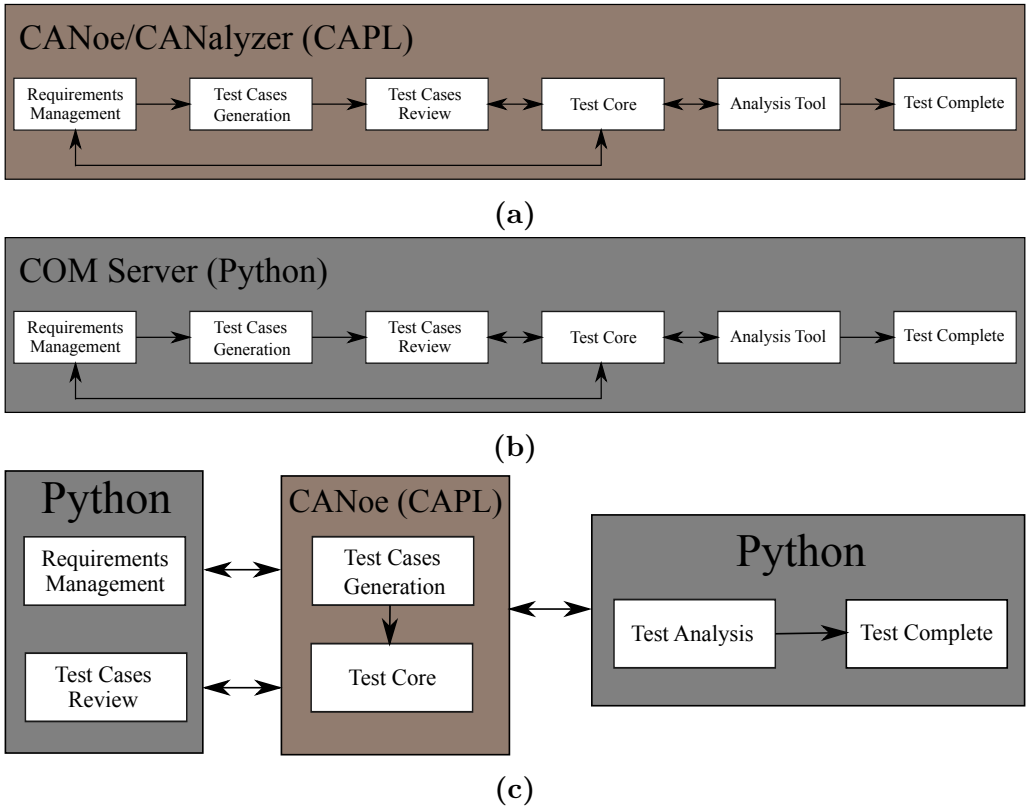


Figure 3.1: Three approaches of designing automatic testing framework

the design of other parts simpler. Accordingly, constructing an interface to connect Python and CAPL would be a goal in this project.

When this project started, a large amount of test cases had already been generated. Hence, it would be advantageous to reuse the existing test cases. Meanwhile, there are both people who are good at Python programming and engineers who have much experience of programming CAPL. We could have good support from this group. Therefore, we chose to use the third methodology in the end. The methodologies of each block in Figure 3.1c will be discussed in the following sections.

3.1 Requirements Management

As introduced in Section 2.2, "Big" and REST web services technologies are two common approaches for web services. "Big" web service can support more comprehensive functions and WSDL can model a flexible and custom-defined interface, which can make SOAP commendable for complex web service implementations. Compared with "Big" Web service, REST provides less functions. For instance, "Big" Web service can support several transport protocols such as HTTP, HTTPS, TCP, SMTP, JMS, MQ, BEEP and IIOP, while REST only supports HTTP and HTTPS. Hence, less options are available when using REST. Moreover, due to the uniform interface in REST, the interface design is restricted. The developers have

to manually write code to assemble resources and encode or decode the exchanged resource representations.

However, REST still have some advantages that are appropriate for many web service implementations. First of all, REST is simple to implement, which is very suitable for beginners. Also, the uniform interface implies that developers do not need to make decisions of interface design if the Web APIs are already defined. Furthermore, REST supports several format of resource representations such as XML, JSON, YAML and MIME, which means that there are more choices for developers based on different requirements. Although REST only provides HTTP transport protocols, most of web services are based on HTTP or HTTPS. Therefore, REST can have the guarantees of these two basic protocols.

In our project, HTTPS protocol is required for Web service to Elektra. REST is adequate for HTTP request operations: `PUT`, `GET`, `POST` and `DELETE`. The "`requests`" module provided by Python can be used to perform these operations. Also, a Web API was already defined in Elektra and can be used directly. The API is formed by URLs constructed with the parameters given in Elektra. Additionally, Elektra supports resource representation with JSON format. JSON-formatted data has the same data structure as the dictionary data structure in Python, meaning that information from Elektra can be easily processed in Python. According the aforementioned reasons, Restful web service was chosen to perform the *Requirements Management* block. It is designed in three steps: 1) constructing URLs based on parameters in Elektra; 2) making web requests by using the module "`requests`" and 3) processing the information acquired from Elektra.

3.2 Test Cases Review

According to Section 2.3, CVCS and DVCS are good choices for storing different version of files and information sharing. Considering the low cost of using, SVN and Git, as free open-source software, they will be a good choice. SVN is much more suitable for the big company with a big repository due to its flexibility of partial checkout. In this case, less space inefficiency will occur in each local user since they could only choose the needed sub-tree files from the repository.

On the other hand, Git is much more flexible for smaller teams with smaller repositories. It provides more flexible communication and commitment among developers since each developer could have its own repository that they can save incrementally. Moreover, Git enhances the ability of non-commitment on experimental change before submitting, which provides the possibility to revert with uncertain outcomes, improving the reliability and the performance of the work.

Since the repository of VCC has become quite large over the years, SVN seems like a better choice. Moreover, the reality is that the Test Group has already created a repository to store all test case scripts in SVN, thus it needs a huge effort to

transform scripts from SVN to Gits. Consequently, SVN will be used to store scripts of test cases. Python programming will be used to perform *Test Cases Review* block in SVN.

When the project started, a Python script that can review source code of test cases had already been provided. In this Python script, a golden template provided by VCC is used as a standard to check the structures of test case scripts. The implementation of *Test Cases Review* could be divided into three steps. Firstly, by using SVN commands, log files of changed test case scripts in SVN should be checked due to different revision numbers or dates. After this, the needed scripts can be selected according to checked log files. The `"Subprocess"` module in Python offers functions for executing SVN commands. Since the log files are represented in XML format, the `"xml.etree.ElementTree"` module in Python will be applied to parse information (authors, file names) from those log files. Secondly, the selected test case scripts should be reviewed via the Python script mentioned above. Thirdly, different actions shall be taken to inform the authors of test case scripts according to different results we get from the previous step.

3.3 Test Core

Test Core will mainly be designed using four stages. The first stage is to create a framework that automatically performs a testing process that includes selecting test cases, deciding sequence of test cases and executing test suites. Initially, dummy test cases could be used to verify this testing process. The second stage is to add simple functionalities, such as test loops, available time and so forth. Next, real test cases with different control systems, such as seat control, light control, etc, will be applied to verify the designed framework. Finally, more advanced functionalities are considered to add into the framework to make it more intelligent.

3.4 Analysis Tool

For *Analysis Tool*, two alternatives were considered at the earlier phase in the project. One alternative is to analyze the testing report while Test Core is running. The main idea is that once each test case finishes its execution and generates a testing report, the Analysis Tool will analyze the generated testing report. However this method might affect the Test Core and cause a burden on the hardware in HIL experimental environment.

The other method is to collect all generated test reports after Test Core finishes running. The tool can then analyze all reports and give a conclusion that can be used to design algorithms that can make decisions for the subsequent testing phase. The advantage of this method is that it can separate the analysis process, which will not impact the implementation of Test Core. It also provides simplicity of design of

Analysis Tool and will not lead to a burden on the hardware. Therefore, we decided to use the second method.

The Analysis Tool will be designed using Python. It is divided into four steps: 1) reading testing reports and extracting the useful information in reports; 2) sorting the obtained information; 3) analyzing the sorted information and calculate pass rate, failed test case and etc; 4) generating a refined report with analyzed result.

3.5 Decision

For *Decision*, the main idea in this block is to decide a new test cases list for next testing phase. A basic approach is that if the failure rate of one test case is over 10%, the test case should be executed more times. This test case would be inserted after each element except itself in the original list. If the number of the test case with failure rate over 10% is more than one, this process will be repeatedly. To be specific, if executed test cases are test2, test4 and test5, and the failure rate of test2 and test5 has a failure rate over 10 %, then the new test case list would be test4, test2, test5, test2, test2, test5, test4 and test5. If not any test case with failure rate over 10% is found, the test cases will be implemented in another different sequence order.

The Decision part will also be designed using Python. It is divided into two steps: 1) checking original sequence and whether any test case with failure rate over 10% is available and 2)giving a new test case sequence by inserting the failed test cases after each passed test case.

3.6 Interface

As introduced in the third approach (Figure 3.1c), interfaces should be designed to connect Python programming and CANoe. From pre-study phase, we discovered that CANoe support CAPL dynamic-link library (DLL) to perform complex tasks or calculations using other programming languages if the functions in CAPL cannot support such calculations or tasks [42]. DLLs are executable files that act as shared libraries containing functions and resources in Microsoft Windows. By a DLL, calling functions and using resources are enabled from a separate file loaded to the application. Hence, in this project, the components, such as Requirement Management, Test Cases Review, Analysis Tool and Decision, can be implemented by Python programming and connected to CANoe via a DLL file. CAPL export table is used to export the functions created by other languages to CAPL programming. CAPL DLLs are usually implemented using C/C++ programming, thus a solution should be applied to embed Python into C application. Python programming provide APIs that gives C and C++ programmers access to the Python interpreter at a variety of levels.

Using the approaches described above, the interface will be designed in the following stages: 1) creating a static function to embed Python programming into C application using Python/C APIs, 2) creating a function that can call the static function and return the output to CAPL, 3) export the built function with CAPL export table and 4) building the DLL files using Visual Studio.

4

Implementation

Based on the methodology chosen in Chapter 3, the detailed design of automatic testing framework is shown in Figure 4.1.

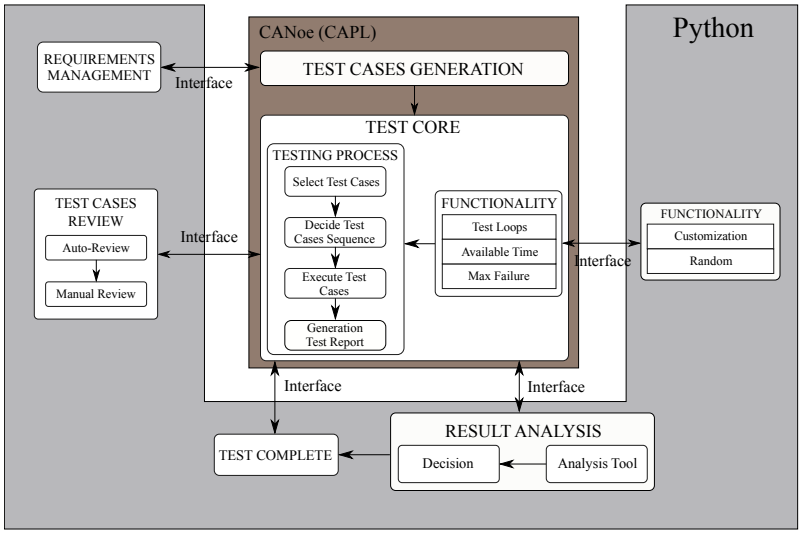


Figure 4.1: The detailed design of the automatic testing framework.

Test Core is designed in CANoe. The *Test Core* should automate the testing process including test case selection, deciding sequence and executing test cases using the selected execution order. Also, the functionalities, such as test loops, available time and max failure, are added into the testing process. The testing process and functionalities are implemented in CAPL programming. Some complex functionalities, such as random combinations and customization are implemented in Python and connected to CANoe by an interface. Test cases are also generated in CANoe.

Other blocks are implemented using Python. In *Requirements Management*, a Web service is used to obtain test specifications from Elektra to *Test Core*. In *Test Cases Review*, a review tool is designed to review the test scripts stored in SVN.

After *Test Core* is complete, a *Result Analysis* is performed. It has two elements. *Analysis Tool* is created to analyze test reports and gives a conclusion to *Decision* element. Algorithms are designed in *Decision* to make a decision for next testing phase, like give a new test cases list to CANoe. After each test cycle, a result report and a log file need to be generated so that both can be viewed by test engineers (in

Test Complete). Finally, several interfaces are created to connect these parts with *Test Core*. The implementations of each part in the framework will be introduced in the following sections.

4.1 Requirements Management

As mentioned in Section 3.1, the `requests` module is used to make requests to Elektra and URLs are used as the Web API. The `requests` module provides two functions `requests.get` and `requests.post` to implement GET and POST HTTP methods respectively. The parameters used to produce URLs can be viewed directly from Elektra.

The detailed implementation of *Requirements Management* is shown in Figure 4.2.

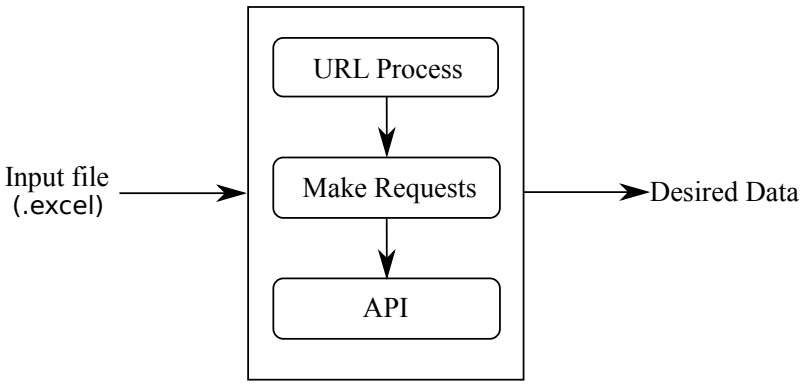


Figure 4.2: The implementation of *Requirements Management*.

Firstly, test engineers write the desired URL parameters in an Excel file as an input. A function "URL process" will be performed to produce a URL based on the parameters in the file. Figure 4.3 shows the details about this function. The file is firstly read by the function `csv.reader()` provided in the `csv` module in Python. A dictionary data structure is then created to store all the parameters' names and values read from the file. If the parameters' values exist, they will be used to build the URL. The result of "URL process" function is a URL string.

After "URL process", a request will be made to get data from Elektra or post data to Elektra. Two functions mentioned at the beginning of this section will be used to perform GET and POST methods. If a GET method is required, the code `r = requests.get(url, auth=('username', 'password'))` needs to be performed and username and password are necessary to get access to Elektra. `r.json()` can be used to get the JSON-formatted data. If a POST method is required, the code `r = requests.post(url, auth=('username', 'password'), data)` needs to be performed. The `data` is the information that needs to be posted and should also be a JSON-formatted data.

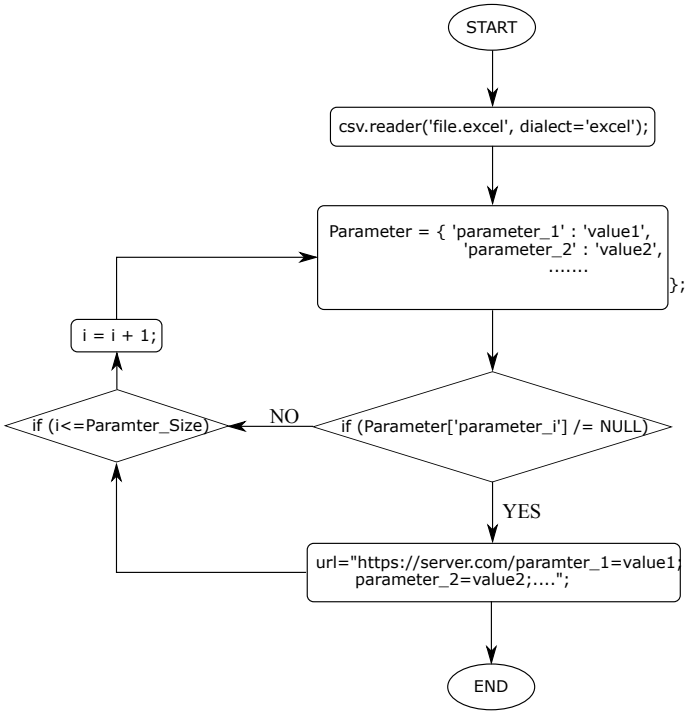


Figure 4.3: The flow chart showing how "URL process" function works.

Finally, if a GET method is performed, the data obtained will be sent to *Test Core*. For POST methods, the data will be sent to Elektra.

4.2 Test Case Auto-Review

Test cases are created in CANoe with CAPL programming and generated with CAN files (shown as `.can` files). The *Test Case Auto-Review* block is implemented in three steps. Figure 4.4 shows the whole work flow of this process.

Firstly, path information and basic information such as revision number and author used to control the range of log data should be checked to ensure which corresponding SVN commands should be applied. `"SVN log"` is applied to go through all previous specific version of files and directories in given path. Through this command, log messages with date, author information and the path where changes occur could be checked. Moreover, since XML file is much easier to extract information by python module `"xml.etree.ElementTree"`, `"SVN log -xml"` is used to create a XML format of log data. Date and revision number, as the possible input parameter, are defined to control the range of files we check, increasing the efficiency to find information needed. For example, `"-r"` could not only help us to check files with revision in a particular order, but also decide the date range of changed files we will check. In this step, we need to make sure which input parameter is taken as the control of range. If a revision number is given, then all files from this revision number to the latest one could be checked automatically to make sure whether it is

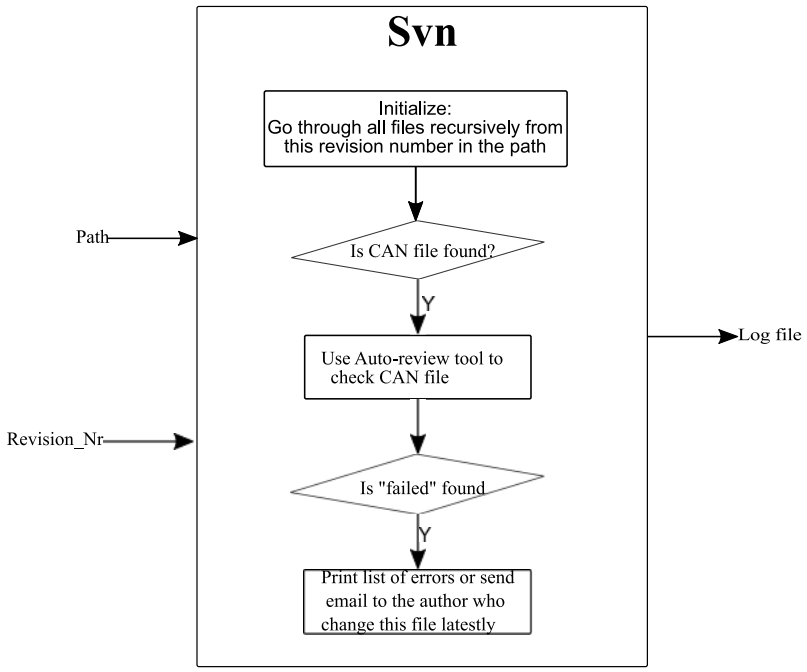


Figure 4.4: The flow chart of test case Auto-Review.

a test case script file. Also, this revision number shall be updated to avoid checking the unchanged test case script files repeatedly. Moreover, if a date is given, then all files changed or created after this date should be checked automatically.

Secondly, those selected CAN files shall be reviewed by a Python script which aims to check whether CAN files accord with the golden template whose target is to give a standard structure to show how indentation, header, tag and comment works in this situation. The result of code review will be stored in a CSV file and could be dealt with different actions. During this part, the data, such as revision, path and author, which we extract from generated XML log files should be used to help us execute the actions to deal with errors found in CSV file after code review process.

The third step is to take different actions according to different results we get from previous step. If no error occurs in the result file, nothing special need to be done. If error occurs in the result file, two actions shall be implemented. We could either store this file as an attachment and email the programmer who changed this CAN file latest, or list log results directly. What's more, if error is caused only due to indentation, direct changes could be made in that CAN file.

4.3 Test Core

The *Test Core* is designed in CANoe and three basic functionalities, that are Max Failure, Test Loops and Available Time, are achieved in CAPL programming. The other functionalities, Random Combination and Customization, are achieved in Python programming and connected to CANoe using interfaces.

In *Test Core*, a control panel is firstly designed to control testing process, shown in Figure 4.5. At the top of the panel, a path dialog is used to import the Excel file containing all URL parameters. Two click buttons are designed to achieve `GET` and `POST` HTTP methods in order to manage requirements. Then the panel for *Test Cases Review* is achieved by having a path dialog to import the test scripts and a button "Check" for triggering the reviews.

The entire testing process is controlled by two click buttons: Start and Stop buttons. The third button "Get Exe Time" is given to get the execution time of every test cases. The functionalities are set in *Test Settings*, which are used to give the test parameters before test activities start. All vehicle control systems are set in the botton of the panel. There are three columns. In the first column, the check boxes are used to select the wanted test cases. The second column is used to start or stop the test cases. The third column is given to indicate the status of test cases.

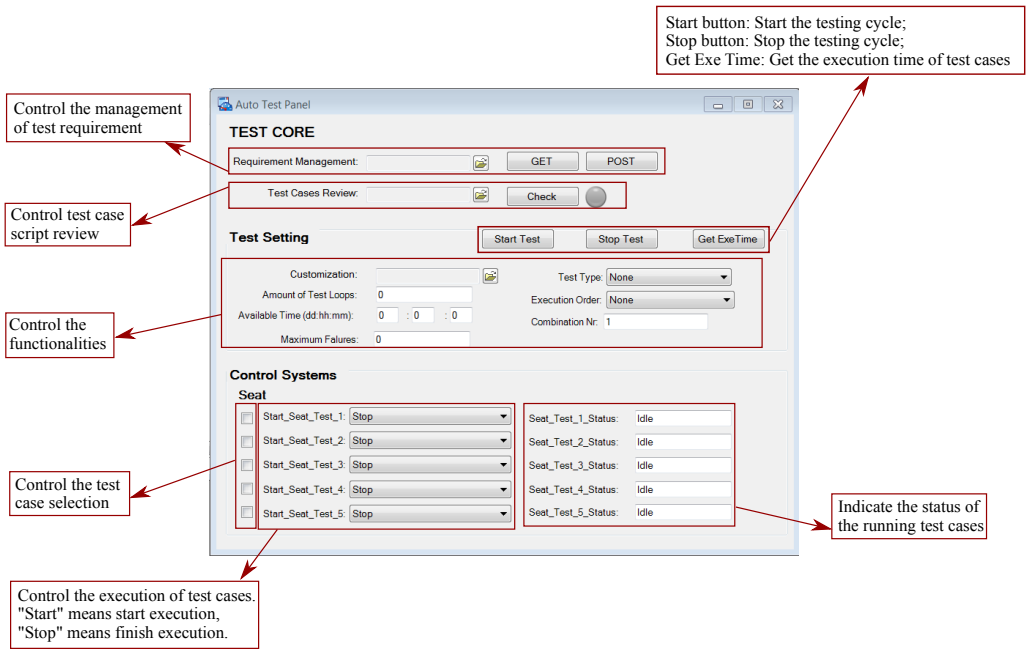


Figure 4.5: The overview of the panel

.

Each setting on the control panel designed above is controlled by a system variable in CANoe. Those system variables can be utilized in CAPL for achieving automatic testing process. The detailed implementation is shown in Figure 4.6.

Three memories are created to store system variable names. `"Selection_Mem"` is the memory of storing system variable names for selecting test cases, the memory `"Execution_Mem"` is used to store the system variable names for executing test cases, and the memory `"Status_Mem"` is used to store the system variable names for test case status. These memories share the same pointer. Besides, two buffers are also created to store the selected system variables: one buffer (`"Exe_buff"`) for storing execution system variables and the other (`"Status_buff"`) for storing status system variables. Two buffers also share the same pointer.

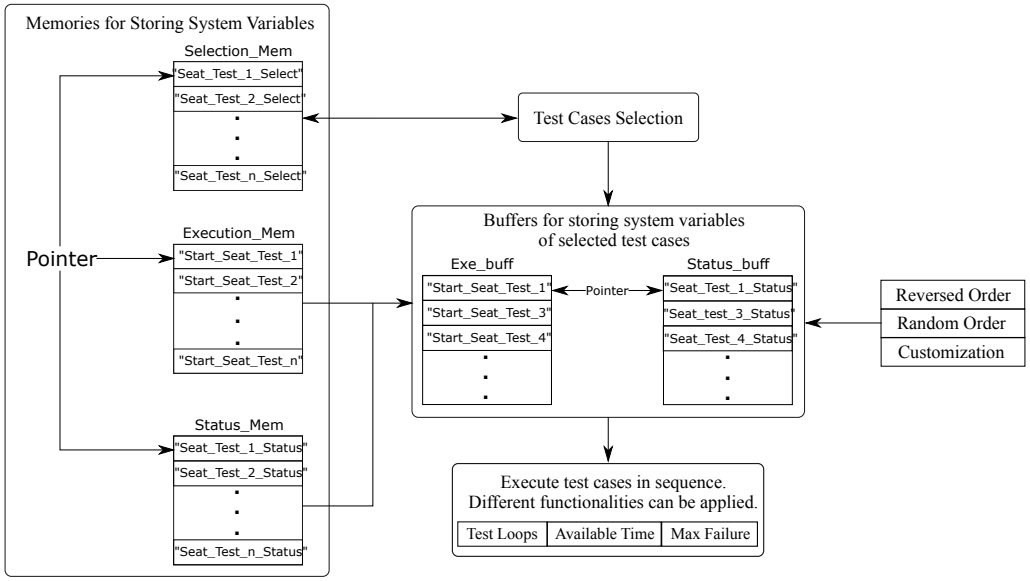


Figure 4.6: The flow char of *Testing Process* in the framework.

Firstly, the system variables in the memory "`Selection_Mem`" are checked by the function defined for *Test Case Selection*. If the test cases are selected, the corresponding variables for test case execution and test case status are stored in the buffers. Different execution orders and "Customization" functionality are also implemented by re-ordering the variables in buffers. Finally, the selected test cases stored in the buffers are executed in sequence with desired functionalities.

4.3.1 Test Cases Selection

A function is created to select wanted test cases. System variables in "`Selection_Mem`" are checked in this function. A system variable for selection has two values. The value 1 means the test case is selected, and the value 0 means the test case is unselected. CAPL also provides two functions that can get values from system variables and set values to system variables, for example, "`sysGetVariableInt()`" and "`sysSetVariableInt()`". Figure 4.7 shows the implementation of *Test Case Selection*.

To begin with, the pointers, both for memories and buffers, are initialized. The value of each system variable in the memory is obtained and checked. If a test case is selected, the variables for test case execution and test case status are put into the buffers. After this, the pointer moves to the next element in memories and buffers. If the test case is not selected, the pointer will move to the next element directly. When all test cases are inspected, *Test Case Selection* is finished.

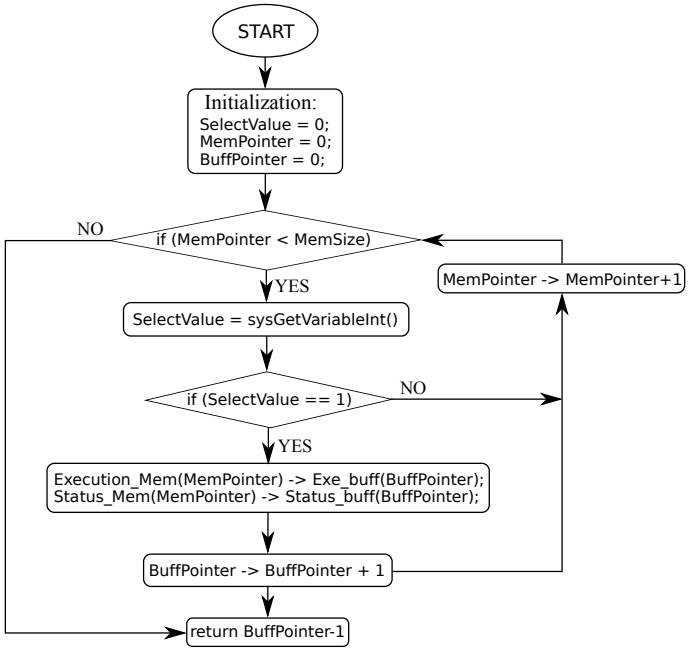


Figure 4.7: The flow chart of automatic execution selection.

4.3.2 Test Cases Execution Sequence

When the selection is finished, the test cases in the buffers are executed with required execution orders. To execute a test case, the system variables for test case execution and test case status are used, for example, "`Start_Seat_Test_1`" and "`Seat_Test_1_Status`" in Figure 4.6. The system variables for test case execution have two values defined: 1 means "START" and 0 means "STOP". The system variables for test case status have five status defined: "Idle", "RunningPassed", "RunningFailed", "FinishedPassed" and "FinishedFailed". The corresponding values for status are 0,1,2,3 and 4.

In CANoe, test cases can be executed in sequence manually, whereas CAPL programming provides timers to trigger a sequential execution automatically. A timer is triggered by the function `setTimer(TimerName, time)`. Implementing a periodic timer can be achieved by recalling this function and be created in the following steps [42]:

- Declare the timer as a variable in a CAPL code.
- Set the timer at the beginning of the testing event by calling the function `setTimer(TimerName, time)`.
- Define the event that needs to be executed in the timer. The timer event is triggered by code "`on Timer TimerName`".
- Reset the timer by re-calling the same function at the end of the task.
- Re-calling the timer functions to achieve period events.

According to the thesis requirements, three different execution orders should be implemented. They are sequential order, reversed order and random combination. The rest of this section will introduce the implementations of three execution orders firstly. The functionalities, such as Max Failure, Available Time, Test Loops and Customization, will be introduced afterwards.

The implementation of sequential order is shown in Figure 4.8. Firstly, a timer is set and the variables are initialized. Then, the first test case in the buffer is executed. After that, the timer "`SeqTimer`" is called and events defined is performed. The event defined in the timer are shown in the grey box. The status of executed test case is checked when entering the timer. If the status is showing "FinishedPassed" and "FinishedFailed", the next test case will be executed. This procedure will be repeated until every test case in the buffer has been executed.

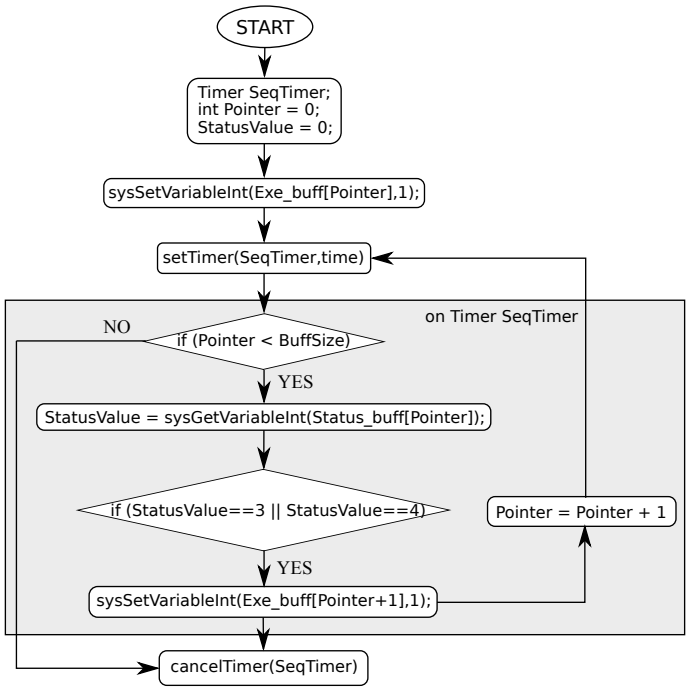


Figure 4.8: The flow chart of automatic execution sequence with sequential order.

The other two orders use the same method above but with different buffer inputs. To have a reversed order, the test cases in the buffer are reversed, shown in Figure 4.9. Two pointers are set for two buffers, one is pointing to the first elements and the other is pointing to the last element. After that, the pointed elements are exchanged. Next, the first pointer is increased by one and the last pointer is decreased by one. This process will be repeated until the two pointers are pointing the same element.

Random combination is performed using Python. Python programming provides a module "`random`" to shuffle a given list. The basic method of random combination is shown in Figure 4.10. The index number of the selected test cases and the combination number shall be provided as inputs. The two inputs are sent to Python script and a randomized index list is generated and sent back to CAPL. Finally,

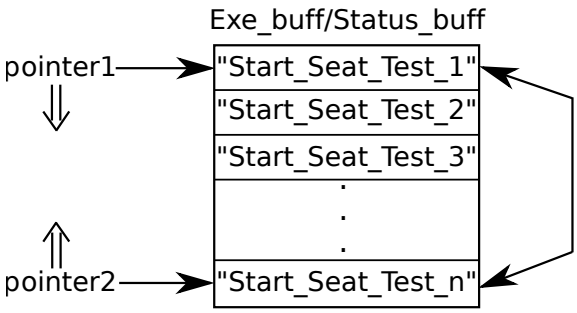


Figure 4.9: The method of automatic execution sequence with reversed order.

the selected test cases in the buffers are re-ordered according the generated random index.

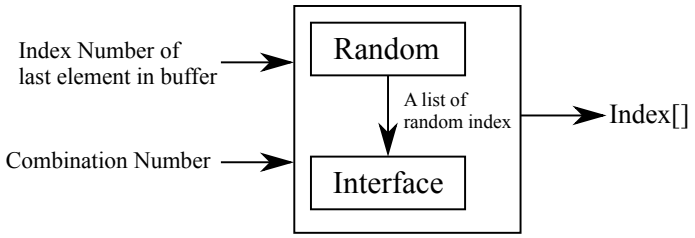


Figure 4.10: The method of automatic execution sequence with random order.

The functionalities aforementioned are added into the execution sequence implementation. Max failure is the basic functionality and available time and test loops are two other independent functionalities.

Figure 4.11 shows the implementation of available time with max failure. This uses almost the same method shown in Figure 4.8. Before calling the timer, the test loop number, max failure number and the available time are obtained from the control panel or from "Customization". In the timer "`SeqTimer_AvaiTime`" (grey box), some more actions are added to perform max failure and test loop fuctionalities. Before executing the next test case, the execution time and failure number are accumulated firstly. Next, the failure number is compared with the max failure number and the execution time is compared with the available time. If the failure number or the available time does not reach the parameters from control panel, the next test case will be executed. When the failure number or the execution time reaches the parameters, the testing process will be terminated. Then the `Analysis()` will start. In this Analysis function, a new test case list, a new available and a new max failure will be given. Finally, the next test phase will be executed based on the parameters obtained from Analysis function.

The implementation of test loop with max failure is shown in Figure 4.12 and is similar with the implementation of available time with max failure. The only difference is the event defined in the timer. In this timer (`SeqTimer_Loop`), the execution time and the failure number will be also calculated. Next, only the failure number will be compared with max failure number. If the failure number is smaller, the next

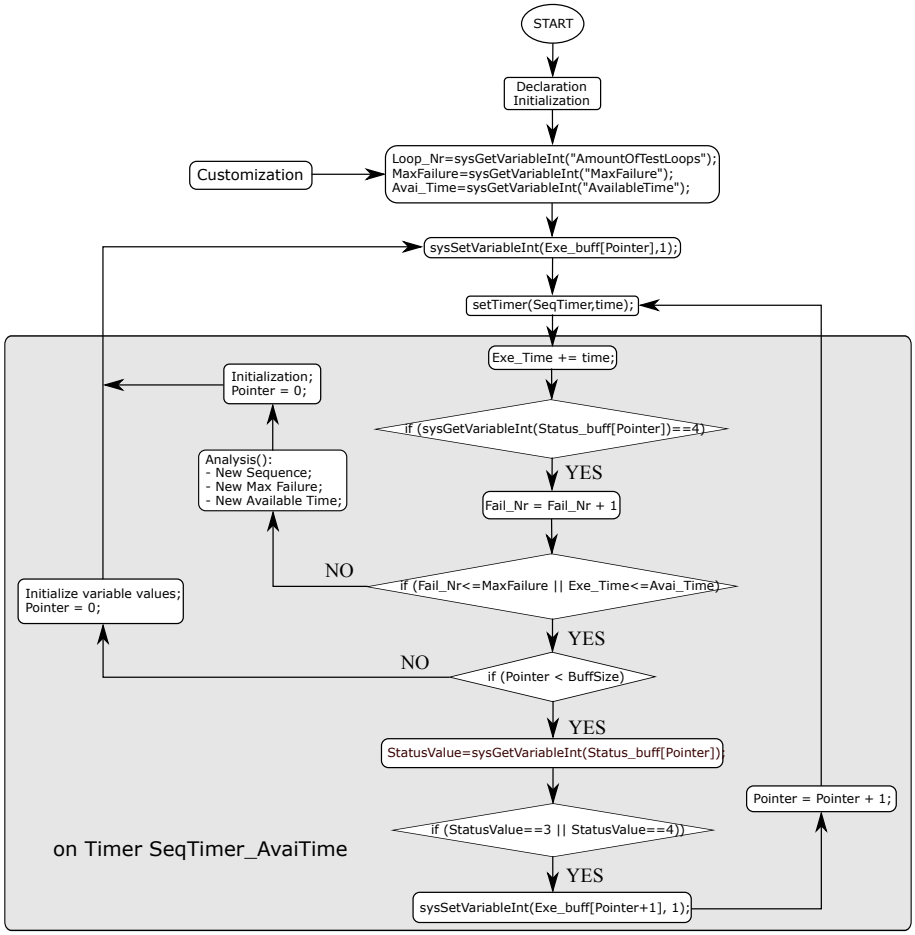


Figure 4.11: The flow chart of automatic execution sequence with functionalities of available time and maximum failure.

test case will be executed. After all test cases in the buffer are executed, the loop number will be checked. If the loop number is larger than 0, the pointer will be reset and the test cases in the buffer will be executed in turn again. The loop number will then decreased by one. When the failure number reaches the max failure parameter or the loop number becomes 0, the current test phase will be terminated and the analysis starts.

The functionality "Customization" is performed in Python. The test case list and the testing setting parameters are written in an Excel file. The reading function can read all the content, extract all the information and store them in a dictionary. Some APIs are created to sort the parameters and sent them to CAPL. These APIs are connected to CANoe by interfaces. The main implementation is indicated in Figure 4.13

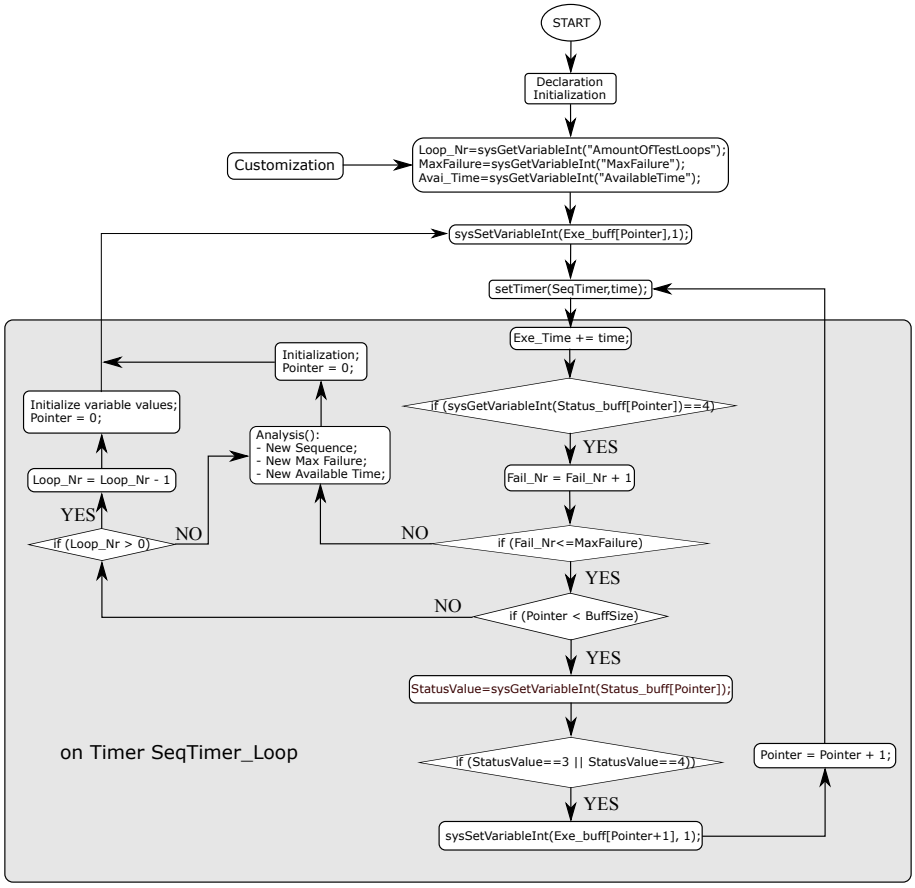


Figure 4.12: The flow chart of automatic execution sequence with functionalities of test loops and maximum failure.

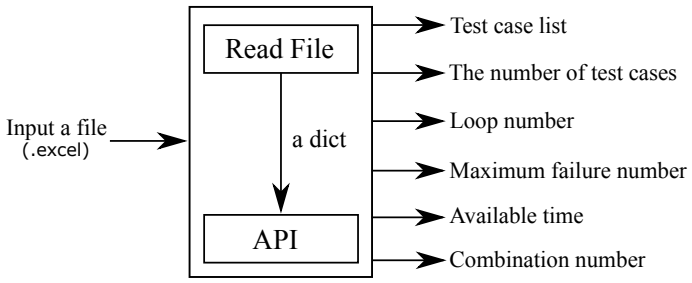


Figure 4.13: The implementation of customization functionality.

4.4 Analysis Tool

All test reports, represented in XML format, are generated automatically after test cases are executed. These test reports are stored in a specific path.

Any XML document under this specific path consists of the information of test cases and test procedures during one test cycle. The whole system can be seen in Figure 4.14. The input of the analysis tool is the path storing test reports and the number of test cases which are executed in current *Testing Process*. Three outputs,

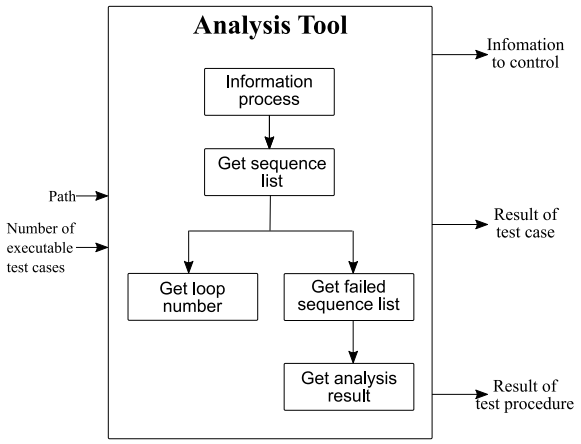


Figure 4.14: The flow chart of analysis tool.

basic information to control the result which will be sent to the *Decision* block, information of test cases and information of test procedures are generated. The basic information to control includes the start time, original sequence, loop number of each test case, and failed sequence. Here, failed sequence stores the title of test cases with the pass rate lower than 90 percent. The information of test case includes the title, the pass rate, the number of pass, the number of failed and the number of warning of test cases. Similarly, the information of test procedure includes the title, the pass rate, the number of passes, the number of failed and the number of warning of test procedure.

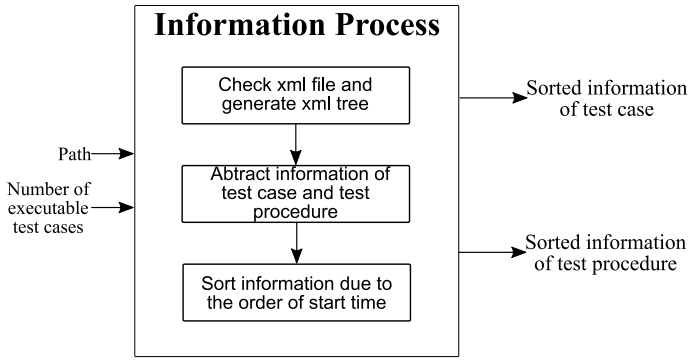


Figure 4.15: The flow chart of information process.

Figure 4.15 shows the work flow of information process. Three steps should be completed during this process. The output of this process shall be the sorted information, which includes the title and verdict of test cases, due to the start time. The first step of the information process is to read all XML files from the given path and to employ `"xml.etree.ElementTree"` module in Python to generate XML trees. This XML tree includes a header and a body containing many smaller elements.

Figure 4.16 shows the process to check XML files in the specific path. If any XML file is found, then the method `"ElementTree.parse"` is used to parse these XML files into an element tree. If not, then a reminder would be printed to tell people that no XML file could be found in this specific path.

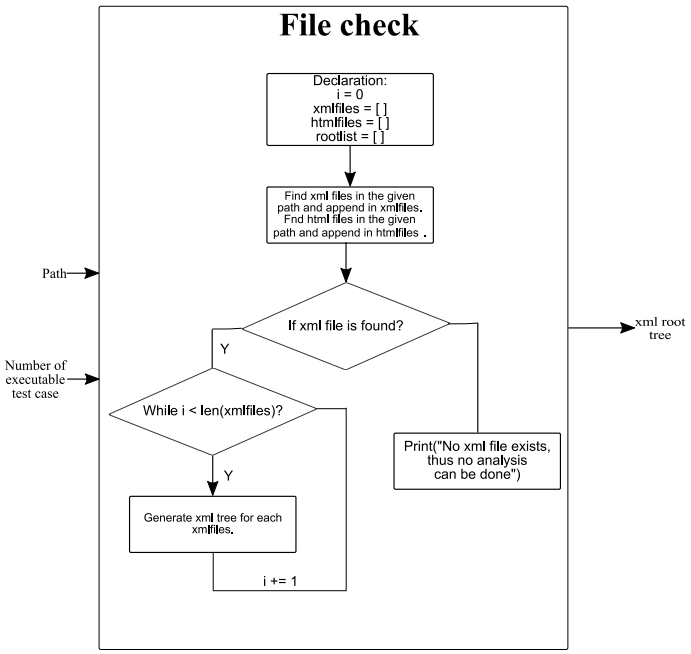


Figure 4.16: The flow chart used to search XML file and generate XML tree.

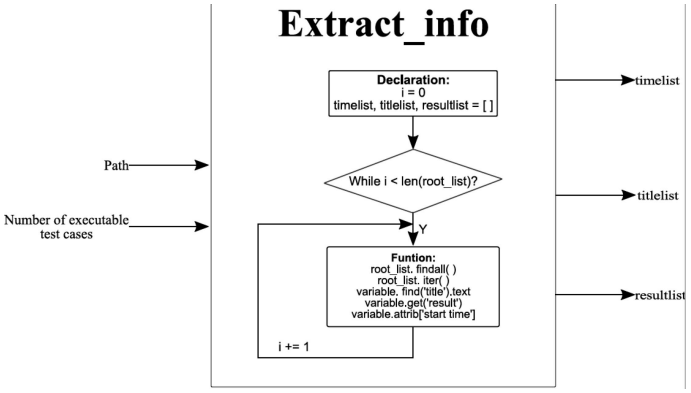


Figure 4.17: The flow chart to describe how to extract information from achieved XML files.

The second step is to extract information of test cases and test procedures from XML. Figure 4.17 shows the function of extracting information of test cases and test procedures. This information includes start time, title and verdict. Each type of information will be put into different memory lists. These different types of information will be gathered into one dictionary and be used in the further analysis process. Some basic introduction will be described here to make a clear idea of how to realize the process of extracting information by using different functions.

The method "`Element.findall()`" can find all child elements within a desired element tree by specifying a keyword. The method"`Element.find()`" is used to find the first child with a particular keyword. The method"`Element.text`" is used to ac-

cess the element's text content. The method"`Element.get()`" can be used to access the element's attributes. Figure 4.18 gives us an example what created XML tree looks like after the execution of *Test Core*. The information should be divided into

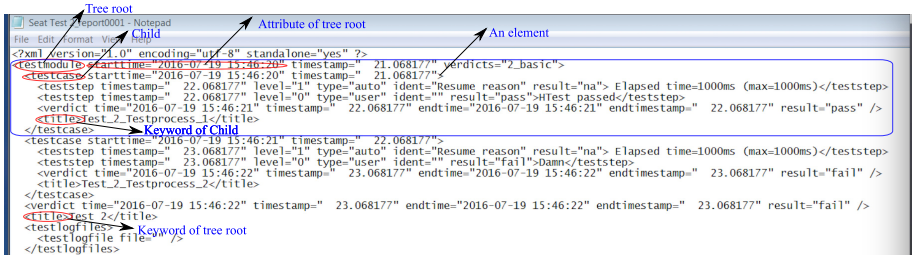


Figure 4.18: The flow chart to show what xml file looks like

two parts: information for test case and information for test procedure.

The third part is to sort information. Different test cases and test procedures are sorted according to the order of start time. "`Sequence_list`" is created as a memory to store the title of sorted test case. Figure 4.19 shows that the work flow used to sort information. Two dictionaries are created to store all information of test case and test procedure.

All generated results of information will be used to achieve the five outputs of analysis tool described above, e.g. original sequence and loop number. After the information process, the original sequence will be firstly generated. Figure 4.20 shows how to achieve the output original sequence list. In detail, since we have already achieved the number of executable test cases in this cycle as an input, sorted title which are stored in the "`Sequence_list`" of this number should be appended in the memory.

What's more, loop number of each test case can be calculated separately by dividing the number of elements in "`Sequence_list`" and the number of elements in dictionary of each test case. Specifically, if the remainder is not equal to zero, another formula shall be used automatically to get correct loop number. In this case, the loop number will be counted as following:

If the index of the element in the dictionary of test cases smaller than the remainder, then

$$loop = (\ \mathrm{len}(\mathrm{Sequence\_List}) - (\ \mathrm{len}(\mathrm{Sequence\_List})\ \mathrm{mod}\ \mathrm{len}(\mathrm{dict\_tm\_list}))) \div \mathrm{len}(\mathrm{dict\_tm\_list}) + 1$$

Otherwise,

$$loop = (\ \mathrm{len}(\mathrm{Sequence\_List}) - (\mathrm{len}(\mathrm{Sequence\_List})\ \mathrm{mod}\ \mathrm{len}(\mathrm{dict\_tm\_list}))) \div \mathrm{len}(\mathrm{dict\_tm\_list})$$

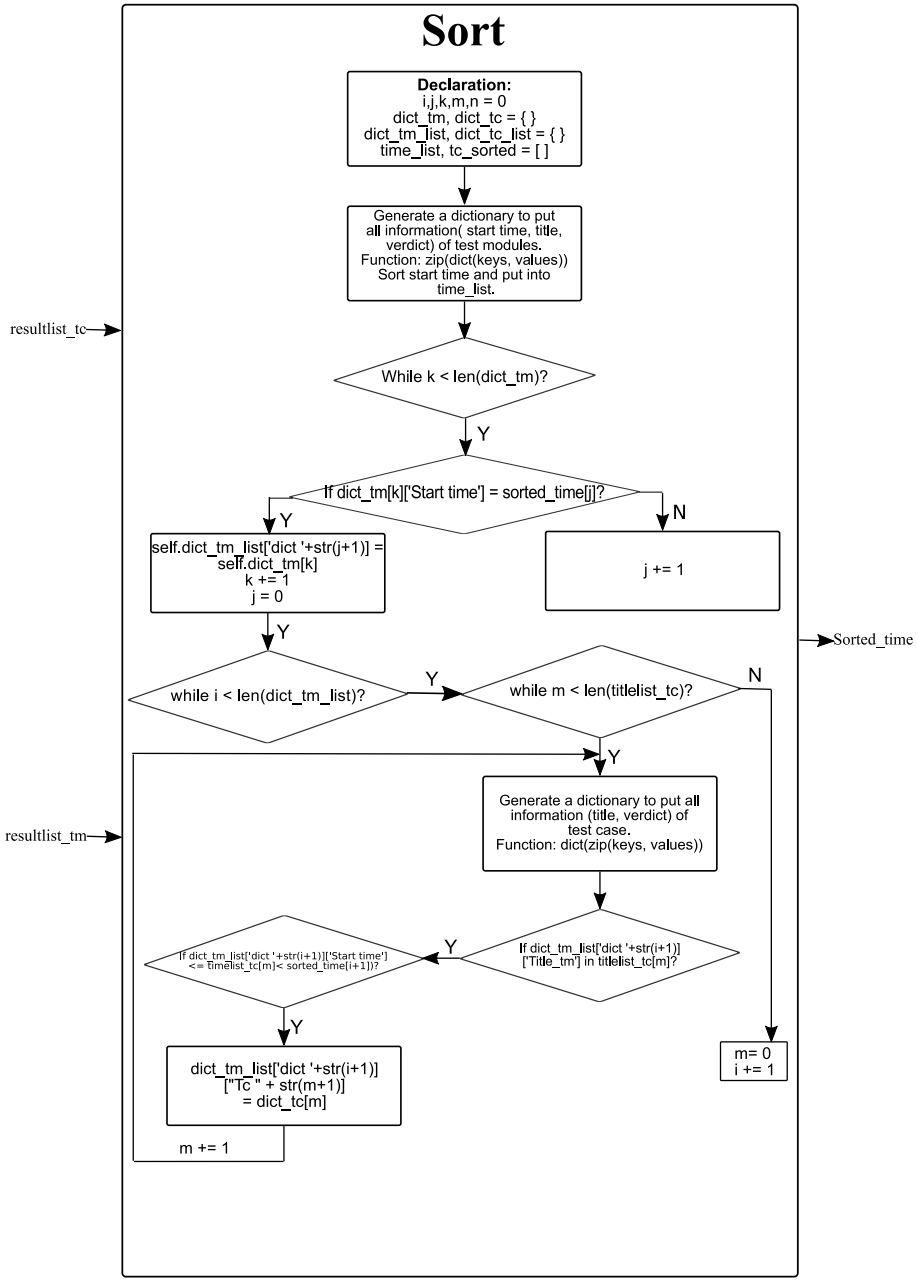


Figure 4.19: The flow chart to show how to sort information of test case and test procedure due to start time

The process to calculate loop number should be completed after the achievement of the original sequence since original sequence act as the input of loop function. Figure 4.21 shows the flow chart of this function. Then an analysis of test case is implemented to obtain the title of test cases with the pass rate lower than 90 percent. This will become a very important foundation to get basic conclusions and to give tips for how to decide which test cases should be executed in the next phase.

Figure 4.22 shows how this conclusion function works to get the output failed sequence. The input of this function shall be the sequence and the gathered infor-

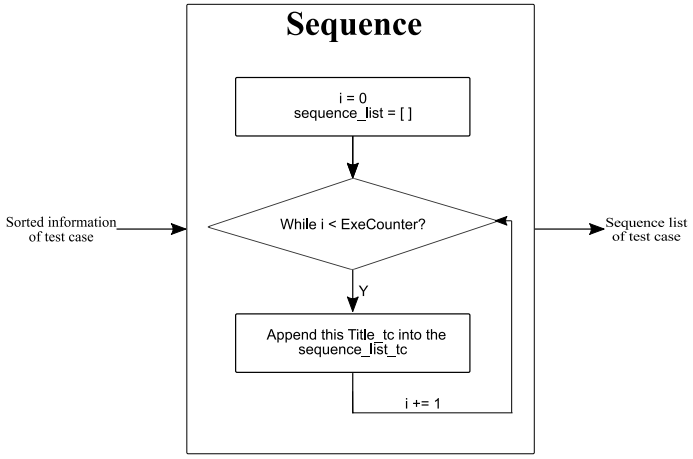


Figure 4.20: The flow chart showing how to achieve the sequence order of executing test cases

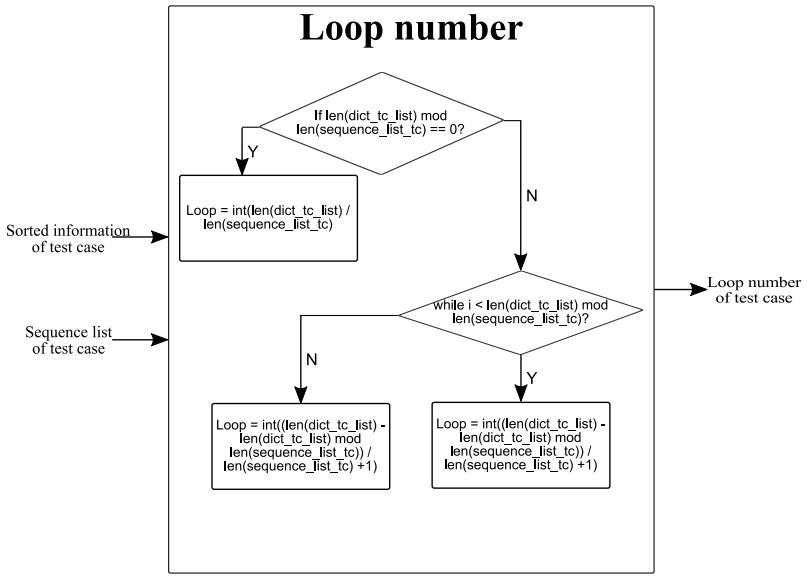


Figure 4.21: The flow chart showing how to calculate the loop number of current test cycle

mation of test cases. As seen in Figure 4.22, the analysis of test cases should be implemented in the Conclusion function. In the function analysis_tm, the failure rate of test cases is calculated individually using the formulation: $pass\_rate = Nr\_fail/(Nr\_fail + Nr\_pass + Nr\_warning)$.

A dictionary is used to store the result information of test cases, which should be easily traced back by users. Such information includes title, pass rate, number of pass, number of fail and number of warning. Figure 4.23 shows the work flow of this function.

In the next step, the function analysis_tc is created. The result information of test procedure will be stored in a dictionary firstly. Moreover, all information in this dictionary will be combined into the regarding result information of test cases if the

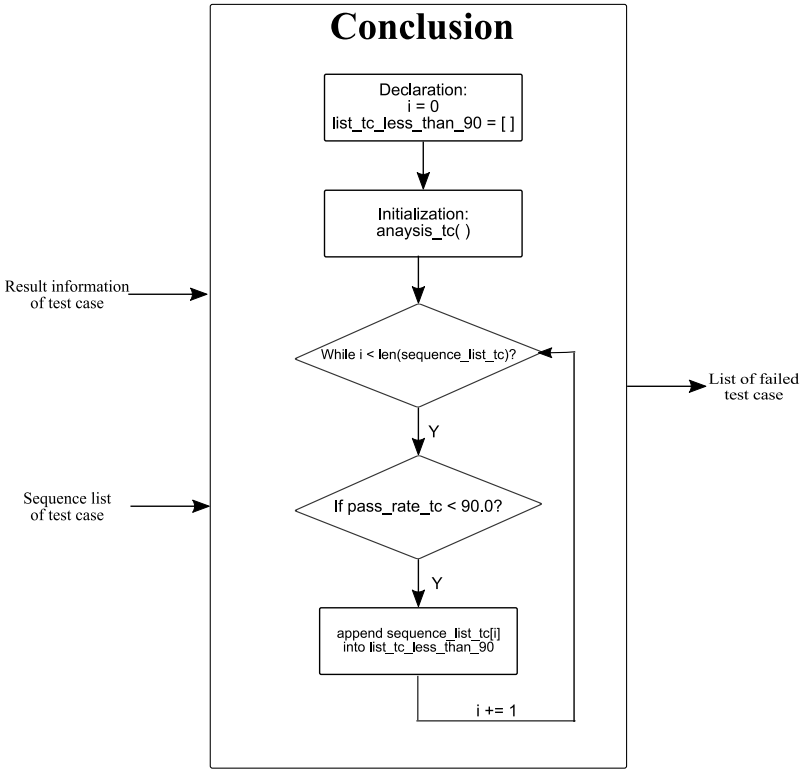


Figure 4.22: The flow chart showing how to obtain the failed sequence

title of test case is one part of the title of test procedure. Figure 4.24 shows the process of this combination of result information.

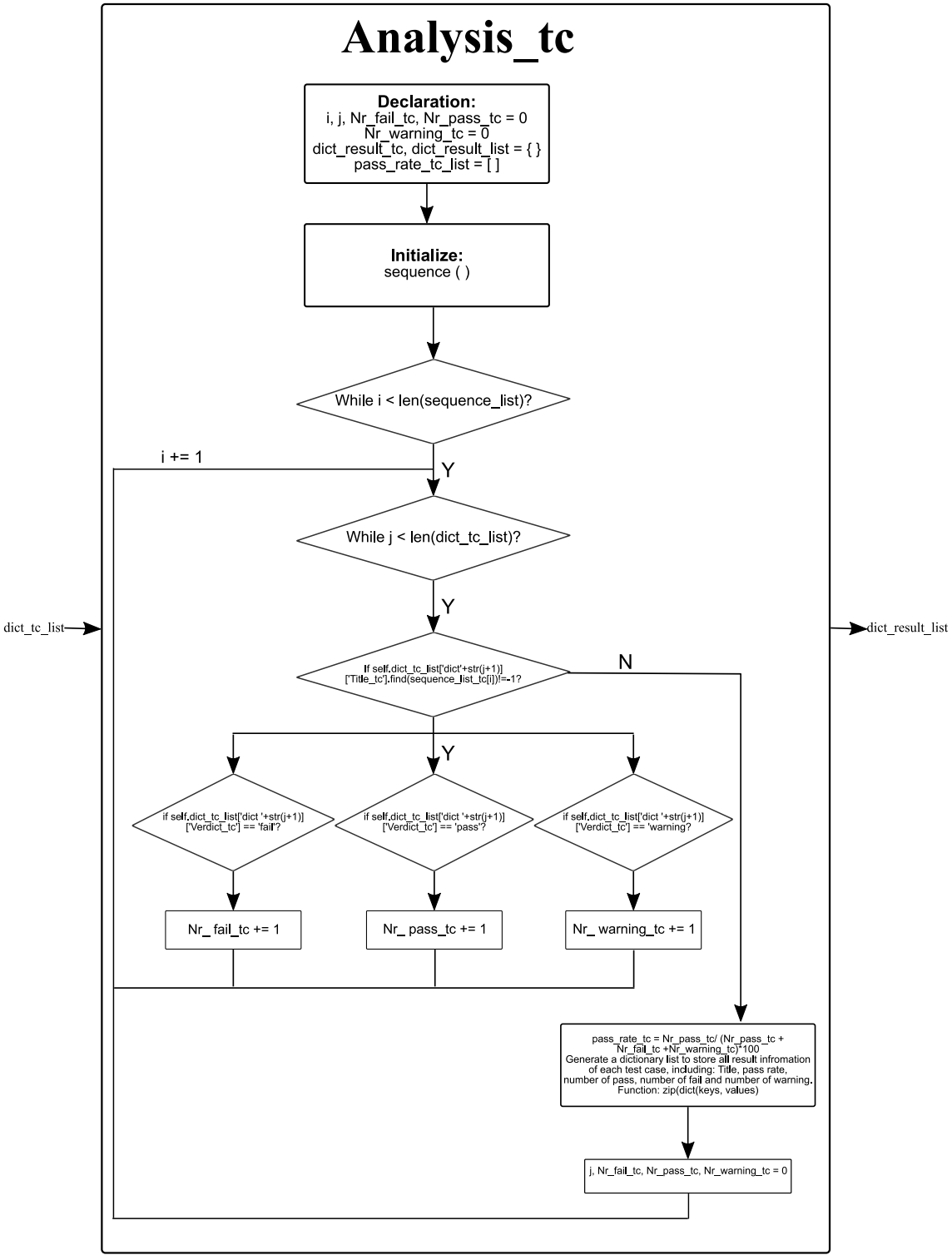


Figure 4.23: The flow chart showing how to obtain the result information of test cases

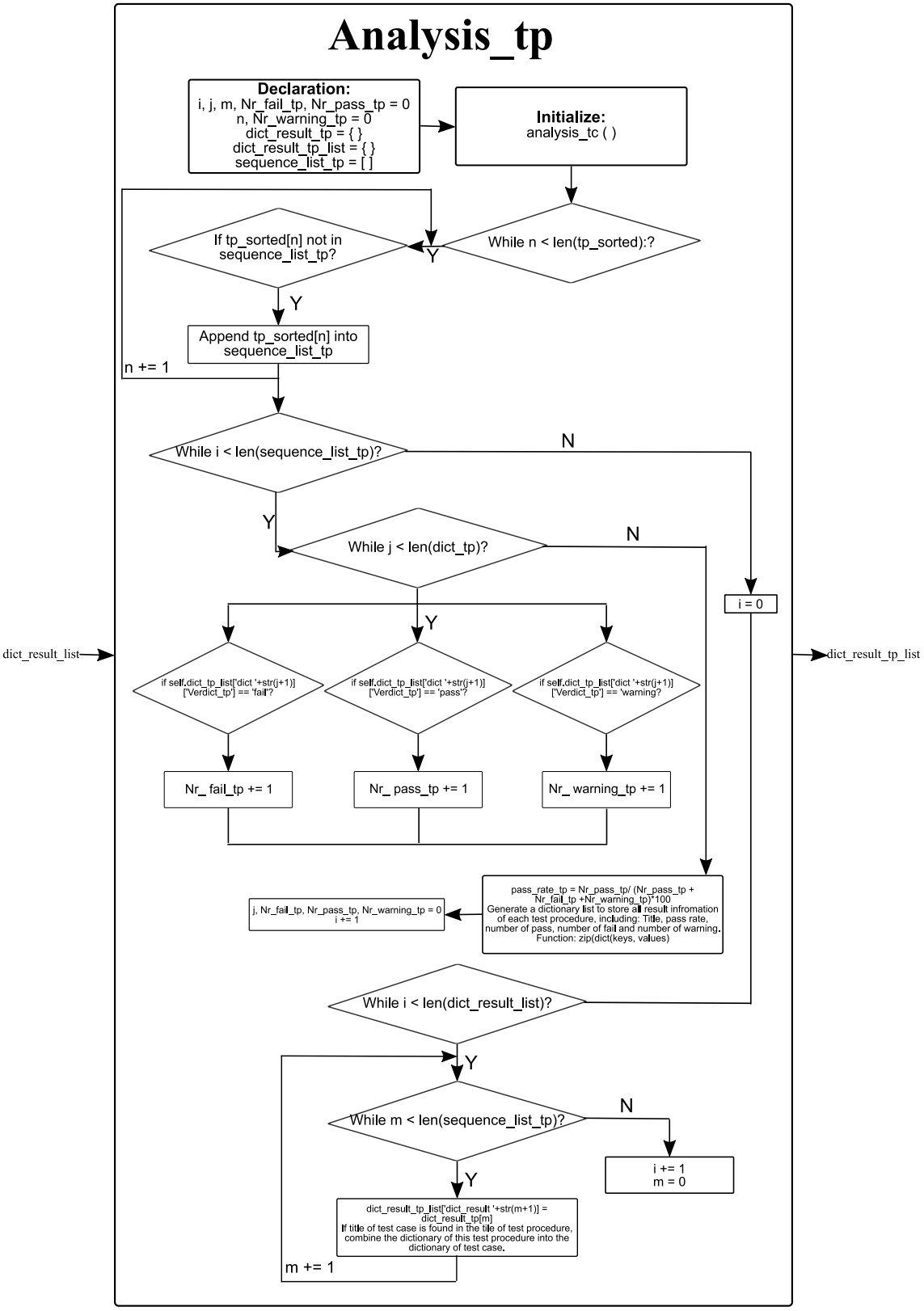


Figure 4.24: The flow chart showing how to achieve the result information of test procedures

4.5 Decision

An algorithm is designed with simple logic in this function. According to the analyzed results, including the control information, information of test cases and information of test procedures, from the analysis tool, new sequence will be formed and sent back to the *Test Core*. Thus new test cycles will be executed in the following according to this sequence.

What's more, an API is created during the *Decision*. One function in API helps return one parameter which could be obtained in the *Analysis Tool*. These parameters could be directly used in different functions to deal with varying algorithms. This action is helpful if an idea to create new functions to deal with another type of testing is created. For example, with the help of function called get_para_original_seq, the parameter original sequence is obtained. With the help of function called get_para_failed_seq, the parameter sequence with the failure rate over 90 % is obtained. Figure 4.25 shows the work flow of the decision block part.

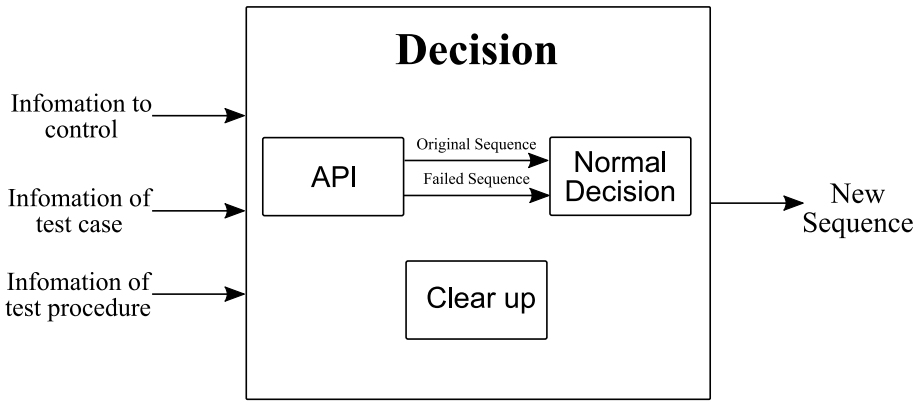


Figure 4.25: The flow chart of implementation of the decision block.

4.6 Log

A log file in html format will be generated in a given path, which is used to help record the actions that occur in each test cycle. These actions include the status of execution, the analysis result and the decision made due to that result. For different situations, different kinds of log files will be generated. For example, whether this test cycle is aborted due to the limit of the maximum failure number, whether this test cycle is executed by the control of available time or loop number and whether test case with pass rate under 90 percent is executed during this test cycle, which of above all affect the content of log files.

4.7 Result to User

An Excel table, including the result information of test cases and test results that is transformed from Python scripts, will be generated in a given path. This file helps users to get the result information much quicker and easier. Several steps and actions are needed to generate such file. First step is to define keys as the keyword of elements that should be written in the first row. Second step is to put the value of these elements as described: First action is to put the value of information of the first test case in the first row. Then following action is to put the value information of the first test procedure of this test case in the next row. This second action should be repeated until the value information of all test procedures of this test case is written. After that, another action is executed to put the value information of the next test case in the next row. Repeat all these actions until the value information of all test cases is written.

4.8 Interface

The implementation of interface is shown in Figure 4.26. Visual Studio is used to generate CAPL DLL files.

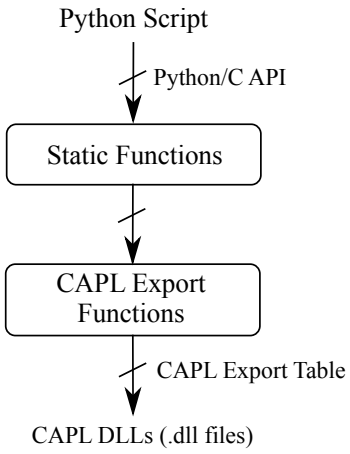


Figure 4.26: The flow chart of implementation of the interface.

The Python script is imported by Python/C API. Static functions are generated to import Python modules and return the desired result. After that, a function is created to export the result to CAPL. CAPL export table is then used to export the defined functions. Finally, a CAPL DLL file is generated by using "Build" in Visual Studio.

5

Results

In this chapter, the result of our project is given. Firstly, the outcome and the achievements of the automated testing framework we developed are described. Next, some figures are presented to show the achievements of our thesis.

In this project, we developed an automatic testing framework with mandatory requirements. The complete framework is built by the required components that are Requirement Management, Test Cases Generation, Test Case Review, Test Core, Result Analysis and Test Complete. Also, the required functionalities were also accomplished and added to this framework. Besides, a simple algorithm that can take simple decision for next testing phase is achieved. Interfaces that connect Python to CANoe were created. Therefore, an entire tool chain that connects different components is finally achieved in this framework. The rest of this section introduces how the developed framework works.

A Web API is built in Requirement Management and can transmit test specifications, test parameters and test description from Elektra to CANoe. An Excel file that contains URL parameters needs to be loaded in CANoe. Then functions created in the interface can be used to obtain information like the name of a test case, the procedure of a test case and so forth. When the necessary information is obtained, test cases (CAPL script) can be generated in CANoe and then stored in SVN. In this project, dummy test cases are generated.

Before a test activity, a test engineer may want to review the script and to check whether these scripts fulfill the requirement. By inputting the path of the script which is needed to check and calling the function for script reviewing the script, the errors will be printed if there are any. Afterwards, emails can be sent to the author of the reviewed script with an attachment of errors. Figure 5.1 shows the result when failure happens after executing test auto-review function.

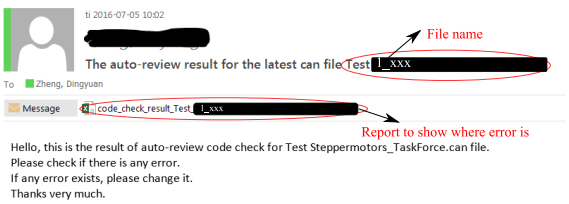


Figure 5.1: The chart to show the result after failure in auto-review function

The testing process is performed in Test Core. Firstly, test cases should be selected and the sequence order (including Front to Back, Back to Front and Random Combination) needs to be chosen next. Afterwards, the test settings need to be set, such as setting the loop number, available time or max failure. By clicking the Start button, this test activity starts and the test cases are executed with the selected sequence automatically. When the amount of the desired test cases are enormous, the functionality Customization can be implemented. An Excel file with the list of test cases can be created and loaded into framework. The function defined in Customization then reads the name of test cases and stores them into a buffer. A test engineer can start the testing process directly after loading this Excel file. Once a test case is executed, a test report of this test case is generated automatically.

After this test activity is finished, the analysis event can be implemented by calling the corresponding functions in the interface. The analysis event contains two elements: Analysis Tool and Decision. The generated test reports are analyzed by the analysis tool we designed and a conclusion is then given to the Decision. Based on the given conclusion, a decision is taken and a new list of test cases is finally provided.

Test Complete is the last step in this framework. An Excel file that indicates the details of the conclusion from analysis is generated. Meanwhile, a log file that keeps a record of details of each implementation within a testing cycle is produced.

In the rest of this chapter, different situations are verified and the outcome of verification is indicated.

5.1 Test Activity with Sequential Order and Available Time

In the project, five dummy test cases are created, in which Test 2 and Test 5 are failed test cases and the others are passed test cases. Firstly, a test activity with sequential order and available time is verified. In this case, Test 1, 2, 3 and 4 are selected. The available time is set to 30 seconds, and the max failure is set to 3. The execution order is set to "Front_to_Back" which means sequential order. All the settings are shown in Figure 5.2

Figure 5.3 indicates the result of this test activity. The selected test cases are executed in sequential order. When entering "Loop 2", only Test 1 is executed because the execution time reaches to 30 seconds. Hence, this test cycle stops and the analysis starts. The new test case list are given by the Analysis function and executed with sequential order. Meanwhile, an Excel file is generated to give the detailed conclusion of analyzed results. Also, a log file is created to show the actions that have made in this test cycle. The Excel file is shown in Figure 5.4. Six columns are generated and they are "Title", "PassRate", "PassNumber","FailNumber", "WarningNumber" and "ExecutionNumber". "Title" column is used to have all executed

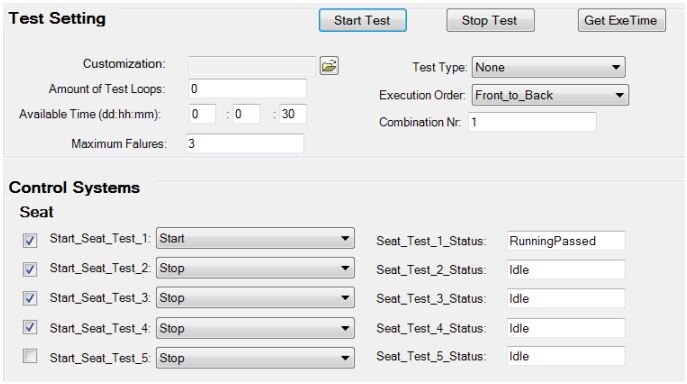


Figure 5.2: The test settings of the test activity with sequential order and available time.

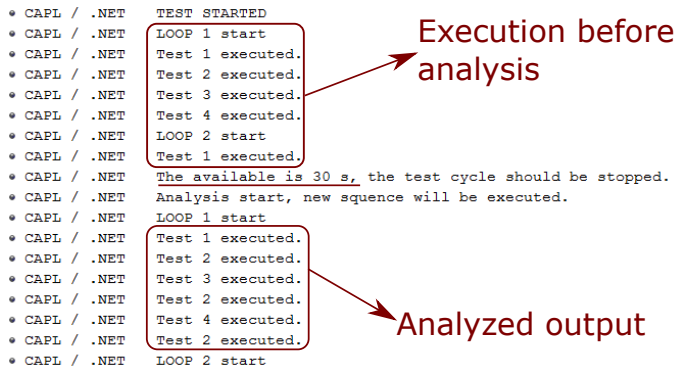


Figure 5.3: The result of the test activity with sequential order and available time.

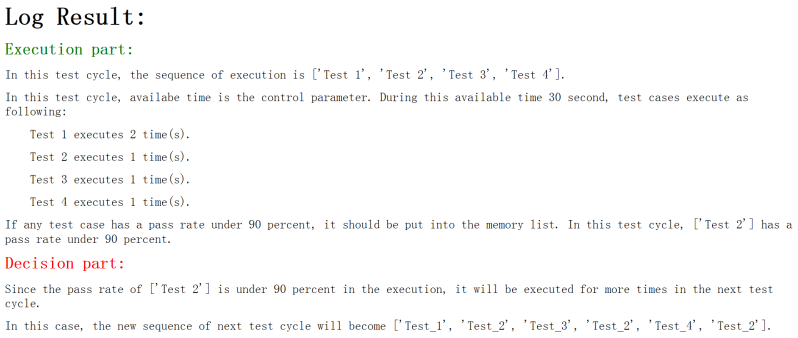
the name of test cases. "PassRate" is used to give the percentage of passed test cases. "PassNumber" is used to show the number of passed test cases. Similarly, “FailNumber” and "WarningNumber" are used to indicate the number of failed test cases and test cases that get warnings. Finally, "ExecutionNumber" is used to record how many times each test case has executed. The log file is shown in Figure 5.5.

Title	PassRate	PassNumber	FailNumber	WarningNumber	ExecutionNumber
Test 1	100.0	2	0	0	2
Test_1_Testprocess_1	100.0	2	0	0	
Test_1_Testprocess_2	100.0	2	0	0	
Test 2	0.0	0	1	0	1
Test_2_Testprocess_1	100.0	1	0	0	
Test_2_Testprocess_2	0.0	0	1	0	
Test 3	100.0	1	0	0	1
Test_3_Testprocess_1	100.0	1	0	0	
Test_3_Testprocess_2	100.0	1	0	0	
Test 4	100.0	1	0	0	1
Test_4_Testprocess_1	100.0	1	0	0	
Test_4_Testprocess_2	100.0	1	0	0	

Figure 5.4: The excel file of the test activity with sequential order and available time.

Two parts are included in the log file. In "Execution part", the list of executed test case and how many times each test case has executed are included. Next, the test cases that have a pass rate below 90% are also listed. Currently, this pass rate

number is hard coding. However, the number can easily become configurable in the framework in the future development. In "Decision part", the new test cases given by *Result Analysis* are listed. This Excel file and log file will be given during each test cycle based on the test results.


Log Result:

Execution part:

In this test cycle, the sequence of execution is ['Test 1', 'Test 2', 'Test 3', 'Test 4'].

In this test cycle, availabe time is the control parameter. During this available time 30 second, test cases execute as following:

Test 1 executes 2 time(s).

Test 2 executes 1 time(s).

Test 3 executes 1 time(s).

Test 4 executes 1 time(s).

If any test case has a pass rate under 90 percent, it should be put into the memory list. In this test cycle, ['Test 2'] has a pass rate under 90 percent.

Decision part:

Since the pass rate of ['Test 2'] is under 90 percent in the execution, it will be executed for more times in the next test cycle.

In this case, the new sequence of next test cycle will become ['Test_1', 'Test_2', 'Test_3', 'Test_2', 'Test_4', 'Test_2'].

Figure 5.5: The log file of the test activity with sequential order and available time.

5.2 Test Activity with Sequential Order and Max Failure

In this section, a test activity with sequential order and max failure is verified. Figure 5.6 shows the test setting. The available time is set to 1 minute and the max failure is set to 2.

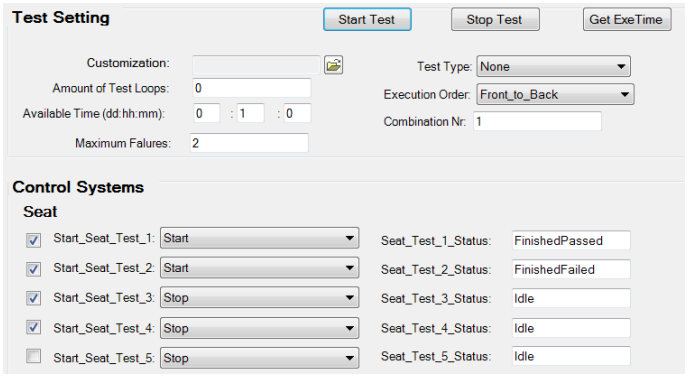


Figure 5.6: The test settings of the test activity with sequential order and max failure.

In the execution result shown in Figure 5.7, the test cycle is stopped when Test 2 is executed in Loop 2 because the failure number has reached to 2. After analysis, a new test case list is given and the next test cycle starts. The Excel file and the log file of this test situation are shown in Figure 5.8 and Figure 5.9.

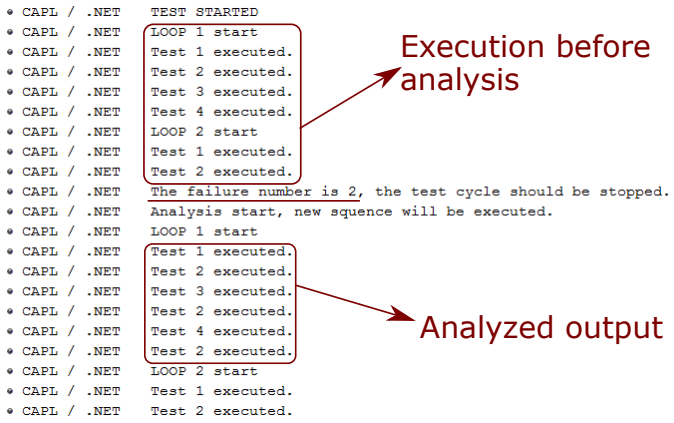


Figure 5.7: The result of the test activity with sequential order and max failure.

Title	PassRate	PassNumber	FailNumber	WarningNumber	ExecutionNumber
Test 1	100.0	2	0	0	2
Test_1_Testprocess_1	100.0	2	0	0	
Test_1_Testprocess_2	100.0	2	0	0	
Test 2	0.0	0	2	0	2
Test_2_Testprocess_1	100.0	2	0	0	
Test_2_Testprocess_2	0.0	0	2	0	
Test 3	100.0	1	0	0	1
Test_3_Testprocess_1	100.0	1	0	0	
Test_3_Testprocess_2	100.0	1	0	0	
Test 4	100.0	1	0	0	1
Test_4_Testprocess_1	100.0	1	0	0	
Test_4_Testprocess_2	100.0	1	0	0	

Figure 5.8: The excel file of the test activity with sequential order and max failure.

Log Result:

Execution part:

In this test cycle, the sequence of execution is ['Test 1', 'Test 2', 'Test 3', 'Test 4'].

In this test cycle, loop number is the control parameter. The following shows the number of times each test case executes in this test cycle:

Test 1 executes 2 time(s).

Test 2 executes 2 time(s).

Test 3 executes 1 time(s).

Test 4 executes 1 time(s).

The execution is aborted when it reaches the maximum failure number 2.

If any test case has a pass rate under 90 percent, it should be put into the memory list. In this test cycle, ['Test 2'] has a pass rate under 90 percent.

Decision part:

Since the pass rate of ['Test 2'] is under 90 percent in the execution, it will be executed for more times in the next test cycle.

In this case, the new sequence of next test cycle will become ['Test_1', 'Test_2', 'Test_3', 'Test_2', 'Test_4', 'Test_2'].

Figure 5.9: The log file of the test activity with sequential order and max failure.

5.3 Test Activity with Reversed Order and Test Loop

A test activity with reversed order and test loop are verified in this section. Test 3, 4 and 5 are selected. The test settings are shown in Figure 5.10. The test loop is set to 2 and the max failure is set to 3. The execution order is set to "Back_to_Front" which means reversed order. The result of this test activity is shown in Figure 5.11. The selected test cases are executed backwards with two test loops. In this test

activity, Test 5 is failed. Hence, the new test case list is shown in the bottom of the Figure 5.11. The Excel file and the log file are shown in Figure 5.12 and Figure 5.13

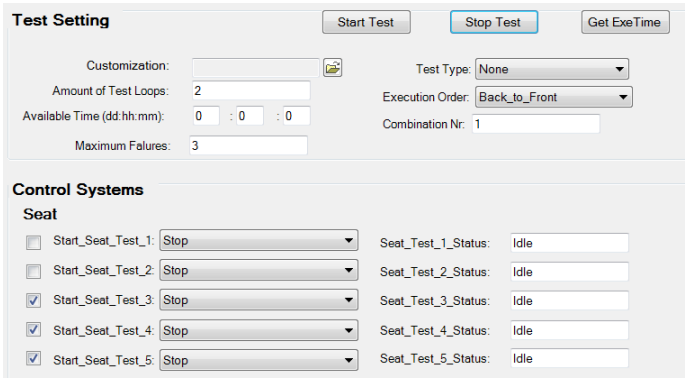


Figure 5.10: The test settings of the test activity with reversed order and test loop.

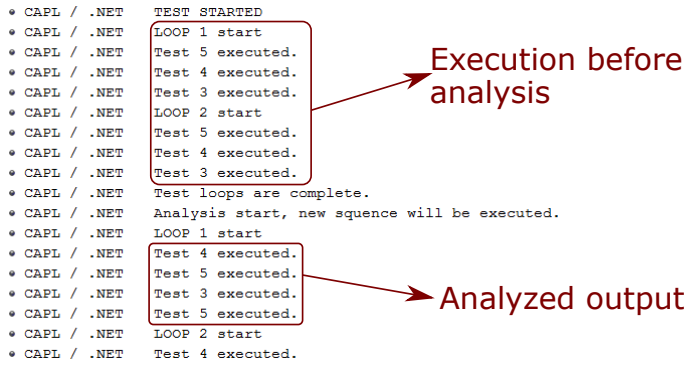


Figure 5.11: The result of the test activity with reversed order and test loop.

Title	PassRate	PassNumber	FailNumber	WarningNumber	ExecutionNumber
Test 5	0.0	0	2	0	2
Test_5_Testprocess_1	0.0	0	2	0	
Test_5_Testprocess_2	0.0	0	2	0	
Test 4	100.0	2	0	0	2
Test_4_Testprocess_1	100.0	2	0	0	
Test_4_Testprocess_2	100.0	2	0	0	
Test 3	100.0	2	0	0	2
Test_3_Testprocess_1	100.0	2	0	0	
Test_3_Testprocess_2	100.0	2	0	0	

Figure 5.12: The excel file of the test activity with reversed order and test loop.

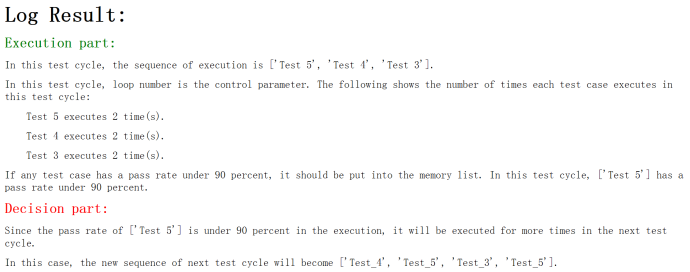


Figure 5.13: The log file of the test activity with reversed order and test loop.

5.4 Test Activity with Random Order and Max Failure

A test activity with random order and max failure are verified also. The test settings are shown in Figure 5.14. All test cases are selected, the max failure are set to 3 and the test loop are set to 2.

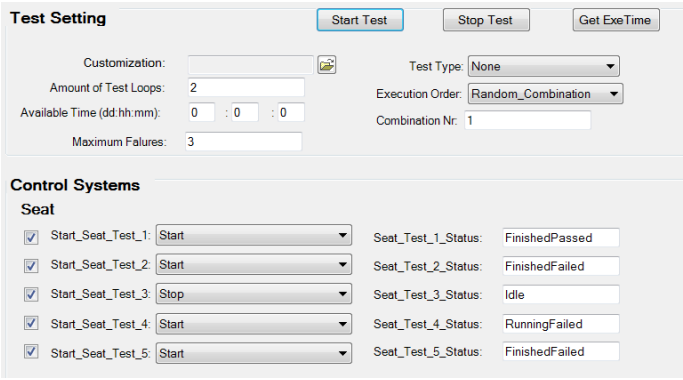


Figure 5.14: The test settings of the test activity with random order and max failure.

The result is shown in Figure 5.15. The test cases are executed in a random order. When Test 2 is finished in Loop 2, the failure number reaches to 3 and the test cycle is stopped. After the analysis function, a new test case list is given, as shown in the bottom of Figure 5.15. Figure 5.16 and Figure 5.17 display the generated Excel file and log file.

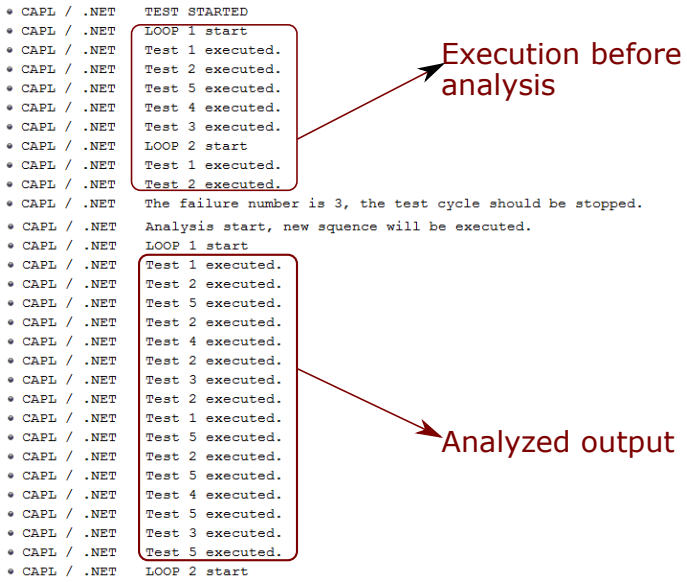


Figure 5.15: The result of the test activity with random order and max failure.

Title	PassRate	PassNumber	FailNumber	WarningNumber	ExecutionNumber
Test 1	100.0	2	0	0	2
Test_1_Testprocess_1	100.0	2	0	0	
Test_1_Testprocess_2	100.0	2	0	0	
Test 2	0.0	0	2	0	2
Test_2_Testprocess_1	100.0	2	0	0	
Test_2_Testprocess_2	0.0	0	2	0	
Test 5	0.0	0	1	0	1
Test_5_Testprocess_1	0.0	0	1	0	
Test_5_Testprocess_2	0.0	0	1	0	
Test 4	100.0	1	0	0	1
Test_4_Testprocess_1	100.0	1	0	0	
Test_4_Testprocess_2	100.0	1	0	0	
Test 3	100.0	1	0	0	1
Test_3_Testprocess_1	100.0	1	0	0	
Test_3_Testprocess_2	100.0	1	0	0	

Figure 5.16: The excel file of the test activity with random order and max failure.

Log Result:

Execution part:

In this test cycle, the sequence of execution is ['Test 1', 'Test 2', 'Test 5', 'Test 4', 'Test 3'].

In this test cycle, loop number is the control parameter. The following shows the number of times each test case executes in this test cycle:

Test 1 executes 2 time(s).

Test 2 executes 2 time(s).

Test 5 executes 1 time(s).

Test 4 executes 1 time(s).

Test 3 executes 1 time(s).

The execution is aborted when it reaches the maximum failure number 3.

If any test case has a pass rate under 90 percent, it should be put into the memory list. In this test cycle, ['Test 2', 'Test 5'] has a pass rate under 90 percent.

Decision part:

Since the pass rate of ['Test 2', 'Test 5'] is under 90 percent in the execution, it will be executed for more times in the next test cycle.

In this case, the new sequence of next test cycle will become ['Test_1', 'Test_2', 'Test_5', 'Test_2', 'Test_4', 'Test_2', 'Test_3', 'Test_2', 'Test_1', 'Test_5', 'Test_2', 'Test_5', 'Test_4', 'Test_5', 'Test_3', 'Test_5'].

Figure 5.17: The log file of the test activity with random order and max failure.

5.5 Test Activity with More Than One Ramdom Combination

With random orders, more than one random combination can be performed. The next test activity, a test cycle with two combination are implemented. Therefore, the test case list is constructed by two randomized test case sets, which is 10 executing test cases in total. The result is shown in Figure 5.19.

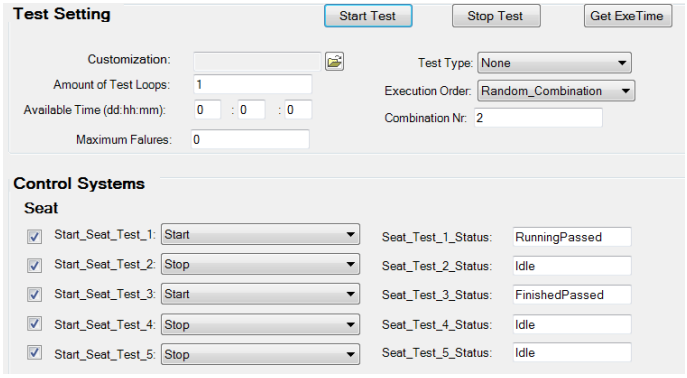


Figure 5.18: The test settings of the test activity with random combination.

```
CAPL / .NET    TEST STARTED
CAPL / .NET    LOOP 1 start
CAPL / .NET    Test 3 executed.
CAPL / .NET    Test 1 executed.
CAPL / .NET    Test 2 executed.
CAPL / .NET    Test 4 executed.
CAPL / .NET    Test 5 executed.
CAPL / .NET    Test 2 executed.
CAPL / .NET    Test 4 executed.
CAPL / .NET    Test 1 executed.
CAPL / .NET    Test 3 executed.
CAPL / .NET    Test 5 executed.
```

Figure 5.19: The result of the test activity with random combination.

5.6 Customization

Finally, the functionality "Customization" is also tested. The customization information is written in an Excel file, as shown in Figure 5.20. In this file, seven columns are created to have the desired test cases, the loop number, max failure, available time and combination number. In this case, Test 1, 2 and 3 are selected. Max failure is 3 and the available time is 36 seconds.

Sequence_Name	Loop_Number	Max_Failure	Available_Time(hr)	Available_Time(min)	Available_Time(sec)	Combination_Nr
Test 1	0	3	0	0	36	1
Test 2						
Test 3						

Figure 5.20: The test settings of the test activity with customization.

The result is shown in Figure 5.21. The test cases listed in the customization file are executed with a sequential order. The test cycle is stopped because execution time reaches 36 seconds.

```
• CAPL / .NET   TEST STARTED
• CAPL / .NET   LOOP 1 start
• CAPL / .NET   Test 1 executed.
• CAPL / .NET   Test 2 executed.
• CAPL / .NET   Test 3 executed.
• CAPL / .NET   LOOP 2 start
• CAPL / .NET   Test 1 executed.
• CAPL / .NET   Test 2 executed.
• CAPL / .NET   Test 3 executed.
• CAPL / .NET   The available is 36 s, the test cycle should be stopped.
• CAPL / .NET   The sequence number is 3.
• CAPL / .NET   Analysis start, new squence will be executed.
• CAPL / .NET   LOOP 1 start
• CAPL / .NET   Test 1 executed.
• CAPL / .NET   Test 2 executed.
• CAPL / .NET   Test 3 executed.
• CAPL / .NET   Test 2 executed.
• CAPL / .NET   LOOP 2 start
• CAPL / .NET   Test 1 executed.
• CAPL / .NET   Test 2 executed.
```

Figure 5.21: The result of the test activity with customization.

6

Discussion

This chapter will discuss the features and the merits of the automatic testing framework "Test Me"compared to the manual testing that is currently used in Volvo Car. Next, the improvements that could be done based on this framework are discussed.

6.1 Features of Automatic Testing Framework

In this project, we accomplished several useful functionalities in the framework. These functionalities make the automatic testing more intelligent and more accurate than manual testing. Features enabled by these functionalities are introduced in the following.

Firstly, test case sequence design is one achievement in this project. Previously, the test cases could only be executed in a predefined sequence order manually, leading to few combinations of test cases. Hence, in system-level testing, the situations that a HIL rig can simulate are seriously limited,potentially hiding problems. This will finally result in waste of time and money for the industry. In this framework, the selected test cases can be executed automatically in different sequence orders. Moreover, random combination mitigates the problem of limited combinations, which provides a comprehensive testing activity.

Secondly, functionalities including test loops, max failure and available time let testing process execute without human supervision. With testing methodology used currently, test engineers are still necessary to supervise a test activity. Therefore, the time that can be used for testing is limited. By these three functionalities, test engineers are able to implement testing outside working hours and during weekends.

Customization is another functionality that can increase the convenience in the testing. When the amount of test cases is enormous, selecting test cases one by one takes much time, which is not convenient. While with customization, test engineers can write the names of test cases and all test settings in an Excel file and load this file into the framework. The framework could read the names and settings, and then start the testing process automatically.

Furthermore, analysis tool is an important feature in the framework. Currently,

test reports are still analyzed manually. Also, the amount of executed test cases is normally very large, leading to a great deal of analysis work. In this framework, the report analysis is implemented automatically, which is time-saving and more accurate.

Finally, taking decision is the most important design in the framework. Right now, designer's experience still plays an important role on making decisions for the next testing phase. Hence, the machine learning in making decisions makes the entire framework more intelligent. When more and more algorithms are designed in this framework, the machine can substitute human experience with making decision, which may increases accuracy.

6.2 Future work

As a follow up project, some effort could be made to make the automatic testing system more intelligent. Even with this framework, test engineers need to make test settings manually before starting a test activity. Also, test cases are still generated manually. Hence, some effort could be spent on functionalities that automate test settings and generate test cases. In this framework, the algorithm implemented can only make very simple decision. Therefore, more advanced algorithms should be implemented to allow for more comprehensive analysis of test reports and more intelligent decision for the next testing phase. For example, future work could focus on machine learning. Overall, only five functionalities were achieved in this framework. Other interesting functionalities could consequently be added to this framework, such as setting the history verdicts of test cases. For example, if the history verdict is set to 10000, then the test cases that have passed for 10000 times can be moved out from the test list. The above example can also be applied to a set of test cases. If a combination of test cases have passed for more than 10000 times, this combination can also moved out. In this project, dummy test cases are used to verify the outcome of the framework. However in the future work, the test cases for real control systems will be applied to this framework. For some simple control systems, such as doors and windows, the simple test algorithms are sufficient. But for complicated control systems such as climate control, more advanced algorithms and functionalites might be necessary. For instance, in climate control systems, the relation between subsystems are much more complex, random combination might not appropriate for giving different situations. Therefore, different test algorithms and functionalities are necessary for different control systems.

7

Conclusion

In this chapter, the goals that have been achieved in this master thesis are described firstly. After that, the experience that we have obtained during the project will also presented.

7.1 Achievements

At the beginning, several goals are defined for this project.

- Design an automatic testing framework and a tool chain including the following procedures: managing test requirements, review test cases, automatic testing process and test reports analysis.
- Add functionalities to make the framework more intelligent. The functionalities to be added are: test loops, available time, random combinations.
- Design algorithms supporting smoke test, regression test, long-term test and stress test. The algorithms should also provide suggestions for the next testing phase based on previous test results. (Optional)

To follow up, we have completed the first two goals. The optional goal has however not been achieved. This would be the next step to let make framework more intelligent to provide specific suggestions for the next testing phase as an improvement.

We have developed a framework that can perform a test activity automatically. This framework can automate the testing process, execute test cases in different combinations, analyze test results and take decisions for the next testing phase. The framework can provide more situations that can happen to customers, which helps test engineers have a more full-scale system-level testing. Interfaces between isolated software systems (like Elektra and CANoe in this case) increase the convenience of test activities and solve problems of data exchange. The most important property of the framework is that it can perform the test activities without human supervision, which means system-level testing can take place outside working hours. This greatly decreases the workload of test engineers. However, it should be noted that the work this framework can perform is fairly simple. More effort is needed to make the

framework more advanced.

7.2 Experience

During the project, we came across some challenges and problems. By overcoming these challenges, we obtained much experience of designing an automatic testing framework.

First of all, object-oriented programming has been a challenge during the whole thesis work. Compared to functional programming, the object-oriented programming has its advantage when a more generic system is constructed. However, object-oriented programming requires a lot of work to create, especially a great deal of planning on how to design the whole system, which presented us with difficulties. Moreover, the lack of experience brought us difficulty to understand what object-oriented programming is at the beginning. To overcome this problem, online course study provided us a general information of the basic concept of object-oriented programming. What's more, an adequate planning phase before programming gave us more understanding of the whole system and reduced flaws to achieve a better design. During the planning phase, we considered the whole program in the system level, especially for the analysis and decision part. Also, the work flow for different functionalities was drawn during this phase to help understand the whole system, thus making it much easier to achieve object-oriented programming. To be specific, during the system level, what we considered should be what the input and output parameters are, what the outer parameter is needed to execute the object, how different parameters should be connected and how to identify and react different objects through different behaviors or methods.

Secondly, it is quite important to find the way how to deal with the program more generically. Because the way to connect Python programming and CAPL programing to build up the system is a bit complex, thus it is much better for us to set a more generic programming in Python. Here, we take analysis tool part as an example to explain what the meaning of generic is. For example, when counting loop number, the result of this functionality should be suitable for all executable sequence, including customization, random, sequential order and reversed order. The way we used to solve this problem is to create the basic version of system which matched one testing mode. Then changes could be applied after practical testings. Algorithm helps us to further improve the accuracy and intelligence of the system and check its performance.

Moreover, how to connect Python and CANoe is the biggest challenge we had in the project. Data transmission is also necessary, which increases the difficulty of creating interfaces. We decided to use CAPL DLL to build the interface, but CAPL DLL only support C and C++ programming, which makes the interface design more complicated. A solution of embedding Python programming into C application is needed. At the beginning, we considered to settle Python embedding and CAPL

exporting into only one function. This solution didn't work and caused hardware problems in CANoe when simulating in HIL rig. Then we decided to define two functions, one for embedding Python into C Application and one for exporting the desired output to CAPL. From solving this challenge, we have obtained the experience of how to create interface between different software systems. Firstly, we should check what support we can get from the software systems that are using. Then, we should look for solutions based on the support we can get. To increase the compatibility, different functions should be defined for different software systems. In our case, we defined two different functions that can support Python and CAPL respectively. One function is used for importing Python programming and give the desired outputs. The other one is used for exporting the outputs to CAPL. Furthermore, reference learning and are always beneficial before design and implementation.

During the project, we faced the situation that progress lagged behind the plan because of optimistic estimation of time and the occurrence of new problems. When we run other projects in the future, this situation can be improved by two aspects. Firstly, a comprehensive pre-study is necessary before the project starts, which can give a more comprehensive overview of the project. Secondly, a good estimation of time for each task is needed since the time spent is always longer than you thought. Besides, when the progress lags behind, the plan needs to be modified according to the real situation. Finally, risk analysis is very important for the entire project.

Bibliography

[1] D. Huizinga and A. Kolawa, *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[2] J. Rushby, "Automated test generation and verified software," in *Verified Software: Theories, Tools, Experiments*. Springer, 2008, pp. 161–172.

[3] E. Bringmann and A. Kramer, "Model-based testing of automotive systems," in *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 485–493.

[4] E. H. Kim, J. C. Na, and S. M. Ryoo, "Implementing an effective test automation framework," in *33rd Annual IEEE International Computer Software and Applications Conference*, vol. 2. IEEE, 2009, pp. 534–538.

[5] L. Heidrich, B. Shyrokau, D. Savitski, V. Ivanov, K. Augsburg, and D. Wang, "Hardware-in-the-loop test rig for integrated vehicle control systems," in *Advances in Automotive Control*, vol. 7, no. 1, 2013, pp. 683–688.

[6] C. Ebert, *Improving Electronic Engineering and Efficiency with Automated Processes*, 1st ed., Vector, 2010.

[7] W. C. Hetzel and B. Hetzel, *The complete guide to software testing*. John Wiley & Sons, Inc., 1991.

[8] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[9] J. Bach, "James bach on risk-based testing," *STQE Magazine*, vol. 1, p. 6, 1999.

[10] J. Watkins and S. Mills, *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2010.

[11] P. C. Jorgensen, *Software testing: a craftsman's approach*. CRC press, 2013.

[12] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 242–251.

[13] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04*, 2010.

[14] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[15] P. Skruch, M. Panek, and B. Kowalczyk, "Model-based testing in embedded automotive systems," *Model-Based Testing for Embedded Systems*, pp. 293–308, 2011.

[16] A. Hartman and K. Nagin, "The agedis tools for model based testing," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 129–132, 2004.

[17] A. Huima, "Implementing conformiq qtronic," in *Testing of Software and Communicating Systems*. Springer, 2007, pp. 1–12.

[18] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[19] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[20] D. Booth, H. Haas *et al.*, "Web services architecture, w3c working group note 11 february 2004," *http://www. w3. org/TR/2004/NOTE-ws-arch-20040211/*, 2004.

[21] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. big'web services: making the right architectural decision," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 805–814.

[22] K. Mockford, "Web services architecture," *BT Technology Journal*, vol. 22, no. 1, pp. 19–26, 2004.

[23] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture," *IBM Systems journal*, vol. 41, no. 2, p. 170, 2002.

[24] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "Extensible markup language (xml) 1.0," *W3C Recommendation, third edition, February*, 2004.

[25] T. Bray, C. Frankston, and A. Malhotra, "Document content description for xml," 1998.

[26] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering *et al.*, "Web services addressing (ws-addressing)," 2004.

[27] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web services description language (wsdl) 1.1," 2001.

[28] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C recommendation*, vol. 26, p. 19, 2007.

[29] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[30] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. " O'Reilly Media, Inc.", 2013.

[31] G. Reese, "The rest api design handbook," *Amazon Digital Services*, 2012.

[32] D. Spinellis, "Version control systems," *Software, IEEE*, vol. 22, no. 5, pp. 108–109, 2005.

[33] R. Somasundaram, *Git: Version control for everyone*. Packt Publishing Ltd, 2013.

[34] B. Collins-Sussman, B. W. FITZPATRICK, and C. M. PILATO, "Version control with subversion for subversion 1.5 (compiled from r3305)," *Ben CollinsSussman, Brian W. Fitzpatrick, C. Michael Pilato, c2008, Modified: Mon*, vol. 5, p. 08, 2008.

[35] B. De Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *Cooperative and Human Aspects on Software Engineering, 2009. CHASE'09. ICSE Workshop on*. IEEE, 2009, pp. 36–39.

[36] "Ecu designing and testing using national instruments products," *http://www.ni.com/white-paper/3312/en/*, 2009.

[37] K. Reif, *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics*. Springer Vieweg, 2015.

[38] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, no. 4, pp. 42–52, 2009.

[39] *Product Information CANoe*, 3rd ed., Vector, 2015.

[40] M. Hammond and A. Robinson, *Python Programming on Win32: Help for Windows Programmers*. " O'Reilly Media, Inc.", 2000.

[41] A. Downey, *Think Python*. " O'Reilly Media, Inc.", 2012.

[42] *Programming With CAPL*, 1st ed., Vector, 2004.