12-18-2011

Dynamic Test Scheduling in Hardware-In-the-Loop Simulation of Commercial Vehicles

Tenil Cletus
tenil.cletus@volvo.com

Dynamic test scheduling in Hardware-In-the-Loop simulation of Commercial vehicles

by

Tenil Cletus

A thesis

Submitted in Partial Fulfillment of the

Requirements for the degree of

Master of Science in Electrical Engineering

at the

University of Connecticut

2011

Master of Science Thesis

Dynamic test scheduling in Hardware-In-the-Loop simulation of Commercial vehicles

Presented by

Tenil Cletus, M.S.

Major Advisor _____

Chengyu Cao, University of Connecticut, USA

Associate Advisor _____

Krishna R. Pattipati, University of Connecticut, USA

Associate Advisor _____

Robert S. Lynch, Naval Undersea Warfare Center, USA

University of Connecticut

2011

LEGAL NOTES

Permission was granted for the research work from University of Connecticut under research abroad program and completed at Scania AB, Sweden in compliance with Swedish Migration board employment regulations. Financial support was granted by Scania AB and meets Migration board work permit requirements.

ACKNOWLEDGEMENT

This dissertation would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance to achieve this milestone in my life and career.

First and foremost, my sincerest gratitude to my academic advisor, Prof.Chengyu Cao for his support, encouragement and guidance.

Mr. Johannes Bergkvist, my supervisor at Scania AB, Sweden for offering this rare opportunity to perform thesis work at one of the leading commercial vehicle manufacturers. He supported me throughout with patience and knowledge at the same time allowing me to be independent at work. I also thank Mr. Sundbäck Pär and all other colleagues at Scania for answering my technical questions.

Prof. Krishna Pattipati, my co-advisor and many of his research papers related to scheduling problems were used as references in my work.

Dr. Robert S. Lynch of NUWC, USA who accepted the invitation to my thesis advisory committee and for his help to lay a good foundation in Systems theory when I took one of his courses.

Prof. Rajeev Bansal, Staff at Graduate school and Office of International Affairs at University of Connecticut for their administrative support.

Swedish Migration board for granting me work permit for this assignment.

My parents, Mr. M.T Cletus and Mrs. Maria Cletus, my sister Caroline Anto, my friends and the one above all of us, God almighty for emotional strength and continuous encouragement.

Page intentionally left blank

CONTENTS

LIST OF FIGURES

ABRREVIATIONS

AB (Swedish)- Aktiebolag (Limited Company)

HIL- Hardware In the Loop

I-Lab2 – Integrations Lab 2

V&V –Verification and Validation

NUWC- Naval Undersea Warfare Center

ECU- Electronic Control Unit

CAN- Controlled Area Network

UFT- User Function Tests

SOPS- Scania Onboard Product Specification

REST- System and Integration Test

GmbH (German)- Gesellschaft mit beschränkter Haftung (Limited Company)

ASM- Automotive Simulation Models

GUI- Graphical User Interface

VT- Virtual Truck

CSV- Comma Separated Variable

VLSI- Very Large Scale Integration

RTOS- Real Time Operating Systems

RMS- Rate Monotonic Scheduler

RRP- Round Robin Policy Scheduler

EDF - Earliest Deadline First

EDD - Earliest Due Date algorithm

MS - Minimum Slack algorithm

FSM – Finite State Machine

FSA – Finite State Automata

RPM – Revolutions per minute

FMEA – Failure Mode Effective Analysis

PID – Proportional Integral Derivative control

UF – User functions

UC – User cases

SCN – Scenario

UML – Unified Modeling Language

MSC – Message Sequence Charts

DEFINITIONS

1. Object oriented programming

Class: A set of objects that share a common structure and a common behavior.

Constructor: An operation that creates an object and/or initializes its state.

Framework: A collection of classes that provide a set of services for a particular domain.

Function: An input/output mapping resulting from some object's behavior.

Method: An operation upon an object, defined as part of the declaration of a class; all methods are operations, but not all operations are methods.

Module: A unit of code that serves as a building block for the physical structure of a system.

Object: Class is only a definition. To execute tasks defined in a class instance of that class is created which yield objects. It is possible to have multiple objects created from one class.

2. Automotive engineering

All processes are in the context of test process at I-Lab2.

Starting vehicle: The process of turning ON Electric power to the vehicle and then turning the Engine ON with an engaged parking brake.

Stopping vehicle: The process of braking the vehicle to zero speed, engaging parking brake, resetting the electrical system to its condition before the vehicle was started and turning the Ignition OFF.

Cruise control: Cruise control is a system that automatically controls the speed of a motor vehicle to maintain a steady speed as set by the driver.

Hill hold: A system that holds the vehicle in the stop position for a short duration when the driver shifts control from accelerator to clutch. A desirable and safety feature while changing gear position up a hill to prevent the vehicle from rolling back.

Retarder: A device used to augment or replace some of the functions of primary friction-based braking systems, usually on heavy vehicles. One way of achieving this is to create pressure in the exhaust system and force the engine to work harder to achieve a breaking effect.

Vehicle configuration: Define specifications of vehicle subsystems and what functionalities are required from a customer's point of view. e.g., its Engine (make and power), Gear box (manual/automatic, make), Vehicle color etc.

3. Test process

User Functions: Scania defines a large number of User Functions (UF) that are realized in several ECUs. Each UF is divided into User Cases (UC) which describe all possible actions that might be performed. The User Cases again are divided into Scenarios (SCN) which give the most detailed information, especially

concerning the CAN communication, in the form of UML Message Sequence Charts (MSC).

Example of a User Function

UF – Central locking

UC- Lock/Unlock vehicle

SCN1 – Lock with remote control

SCN2 – Unlock with remote control

SCN3 – Lock with Key

SCN4 – Unlock with Key

White box testing: White-box testing is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality.

Black box testing: Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings.

Integration testing: Is the phase in software testing in which individual software modules are combined and tested as a group

System testing: Of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

Test pass and fail: For a given set of inputs to a test object (software module/s or hardware unit) if the actual response is equal to the expected response, the test is considered passed, otherwise failed.

Aborted test case: Canceling execution of one test case and continuing with the next test case due to unfavorable test environment conditions.

4. Finite State Machines

Alphabet: Finite, non empty set of symbols. Eg: I = {1, 0}, binary alphabet.

Strings: Finite sequence of symbols from the alphabet I. Eg: w = {00, 01, 10, 11}. The empty set is denoted by, 'ε'.

Length of a string: $| w |$ = 4, length of the string w.

Power of the alphabet, 'I' is denoted by I^k and is the set of strings of length k using symbols from I. Eg: $I^3 = 8$

Set of all strings, excluding {ε} over 'I' is denoted by I^+.

Also, $I^* = I^+ \cap \{\varepsilon\}$.

Language: L is a language over I, such that $L \subseteq I^*$.

A finite state machine is a 6-tuple, $M = (S, I, O, \omega, \nu, s_1)$, where

S is the finite set of states of M

I is the finite input alphabet for M

O is the finite output alphabet for M

$\omega: S \times I \rightarrow O$ is the output function

$\nu : S \times I \rightarrow S$ is the next-state function

$s_1 \in S$ is the initial state

5. Finite State Automata

A deterministic finite state automaton is a 5-tuple, $A = (S, I, v, T, s_1)$, where:

S is the finite set of states for M

I is the finite input alphabet for M

$v : S \times I \rightarrow S$ is the next-state function

T is a non-empty subset of S

$s_1 \in S$ is the initial state

A non- deterministic finite state automaton is a 5-tuple, $A = (S, I, v, T, t_1)$, where:

S is the finite set of states for A

I is the finite input alphabet for A

$v : S \times I \rightarrow P(S)$ is the next-state function

$P(S)$ is the collection of all subsets of S

T is a non-empty subset of S

$t_1 \in S$ is the initial state

SYMBOLS

1. Variable definitions.

E = {K, V, Ω, v, θ, α, ζ } , Set of real time values of vehicle state.

K – Key position, K = {0,1,2} for {Ignition OFF, Ignition ON, Engine ON}

V – Vehicle voltage in volts

Ω – Engine RPM in rad/s

v – Vehicle speed in km/h

θ – Ambient temperature in degree Celsius

α – Road gradient in degrees

ζ – Road slip factor.

E_t = {K_T, V_T, Ω_T, v_T, θ_T, α_T, ζ_T}, set of test pre-requisites. Each element has an upper and lower limit. Eg: v_T = {v_{TL}, v_{TH}} = {70,100}.

2. Discrete mathematics

$p \rightarrow q$ - if p, then q

$p \leftrightarrow q$ - p if and only if q

$\forall$- for all

$x \in Y$ – x is an element of Y.

$x \notin Y$ – x is not an element of Y.

$x \subseteq Y$ – x is a subset of Y.

(x1,x2,…,xn) – n-tuple

P x Q – Cartesian product of P and Q

f: X → Y – function from X to Y.

ABSTRACT

Modern day commercial vehicles are controlled by various Electronic Control Units (ECU). They are not only tested as single units, but also by networking them in Controlled Area Network bus (CAN) to form a complete electrical control system. This is achieved using Hardware In the Loop (HIL) Integration Lab. In HIL, the electrical system is connected to a real time mathematical model of the vehicle plus it's environment so as to form a loop.

Testing functionality of the electrical system begins by defining functional tests. An example would be testing cruise control activation. Executing each test is made possible by parameterizing variables in the vehicle dynamic model and externally controlling them.

HIL based Verification and Validation (V&V) is moving towards automation. This is because of the complexity of electrical control systems is increasing and manual V&V is time consuming. In an automated test environment, a Test Engineer develops test scripts to implement functional tests. These test scripts execute the vehicle model in real time, control parameterized variables, and observe the electrical system response. This is compared to the expected response to decide if a functional test passed or failed.

Tests are designed to remain independent of each other. Scheduling of tests is done by the Test Engineer, which is a difficult task owing to their large number and possible combinations. Hence, the normal practice is to execute tests in a predefined sequence.

To solve the test scheduling problem in Hardware In the Loop simulation, two solutions are proposed. Both the solutions exploit relationship between test case and state of the vehicle in a dynamic simulation environment. An example of such relationship is engaging cruise control only when vehicle speed is above 20 km/h. It can be proved that a test process that is sensitive to the simulation environment will be more realistic and hence efficient.

One solution is to model the test execution as a state machine. Tests are treated as states. Entry conditions for each state are defined using state variables of the dynamic model. When a simulation is run, state variables of the dynamic model are sampled in real time. One sample of state variables trigger a transition from one state to another in the state machine. When the state machine is in one state, a test case corresponding to that state is selected and executed. A sequence of these transitions results in a test process evolving in time.

The second proposed solution is functionally similar to a state machine but it's implementation is derived from logic design. Here, one sample of state variables is compared with entry conditions of each test case. Test cases whose entry conditions match with the current sample are selected for execution.

Both the solutions use Failure Mode Effective Analysis (FMEA) to resolve test selection conflicts, that is, situations where more than one test is selected.

Results show that test execution using this approach is sensitive to the simulation environment and comparable to that of a real test drive scenario. An improvement in test efficiency both in Qualitative and Quantitative terms is also achieved. Test runs show how the new method of test execution allows faults to propagate from one test to another like in a real test drive.

Chapter 1

Scania is a leading manufacturer of heavy trucks, buses and coaches and industrial and marine engines. This chapter briefly describes the company and the Integrations department at Scania research and development center.

1 SCANIA AB, SWEDEN

Scania Aktiebolag (Publ.), commonly referred to as Scania AB or just Scania, is a global automotive industry manufacturer of commercial vehicles - specifically heavy trucks and buses. It also manufactures diesel engines for motive power of heavy vehicles, marine, and general industrial applications. Founded in 1891 in Södertälje, Sweden, the company's head office is still in the city along with the research and development department and a production facility for components, engines, trucks and bus chassis.

Figure 1: Scania R 730 truck (Image courtesy, Scania image bank)

1.1 System and Integration Test department

System and Integration Test (REST) department at Scania R&D center focuses on the complete electric system tests. REST is responsible for testing of about 20 ECU's. These ECU's are networked using three CAN buses in Scania vehicle electrical system. REST owns a number of test vehicles and a HIL lab called I-Lab2. It is a multi-processor test platform with a dynamic model for the vehicle and associated test framework for dealing with tests. Tests are performed in both vehicles and I-Lab2. Integration tests* are white box tests* and focus on specific User Functions* verifying all states on CAN signal levels between ECU's. System tests* are black box tests* for distributed User Functions that include communication on at least one of the vehicles three main CAN buses.

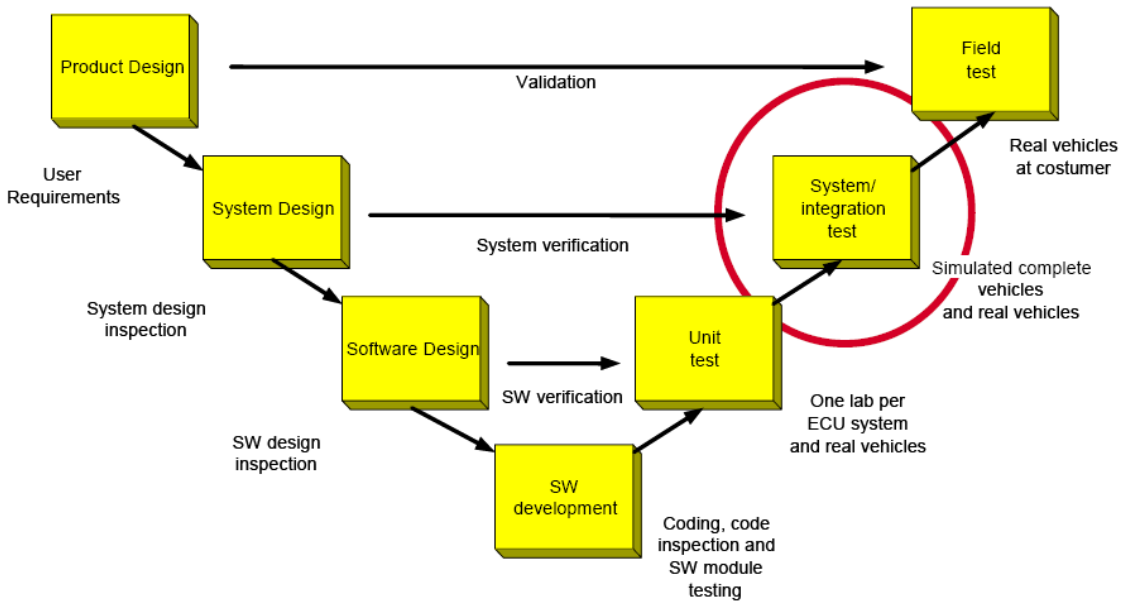


Figure 2: REST position in the development process

* See definitions section of the thesis

Chapter 2

In the automotive industry, HIL simulation has become one of the standard tools for testing ECU's. This chapter contains a brief introduction to HIL followed by it' s use in the automotive industry. The HIL lab at Scania, I-Lab2 is discussed and the test framework is explained.

2 HARDWARE IN THE LOOP SIMULATION

Hardware In the Loop simulation has become a standard tool in the automotive industry for testing ECU's [1]. In HIL (Figure 3), a dynamic model of the environment (vehicle, other hardware, road and driver behavior) surrounding the hardware under test is developed. This model is compiled and downloaded from a Host PC to a hardware capable of executing the model in real time. This real time hardware uses I/O boards that are interfaced to the ECU's. They receive sensor signals from the dynamic model after a digital to analog conversion by the I/O boards. Actuator signals generated by the ECU's are received by the I/O hardware which is then converted to numerical values for the dynamic model.

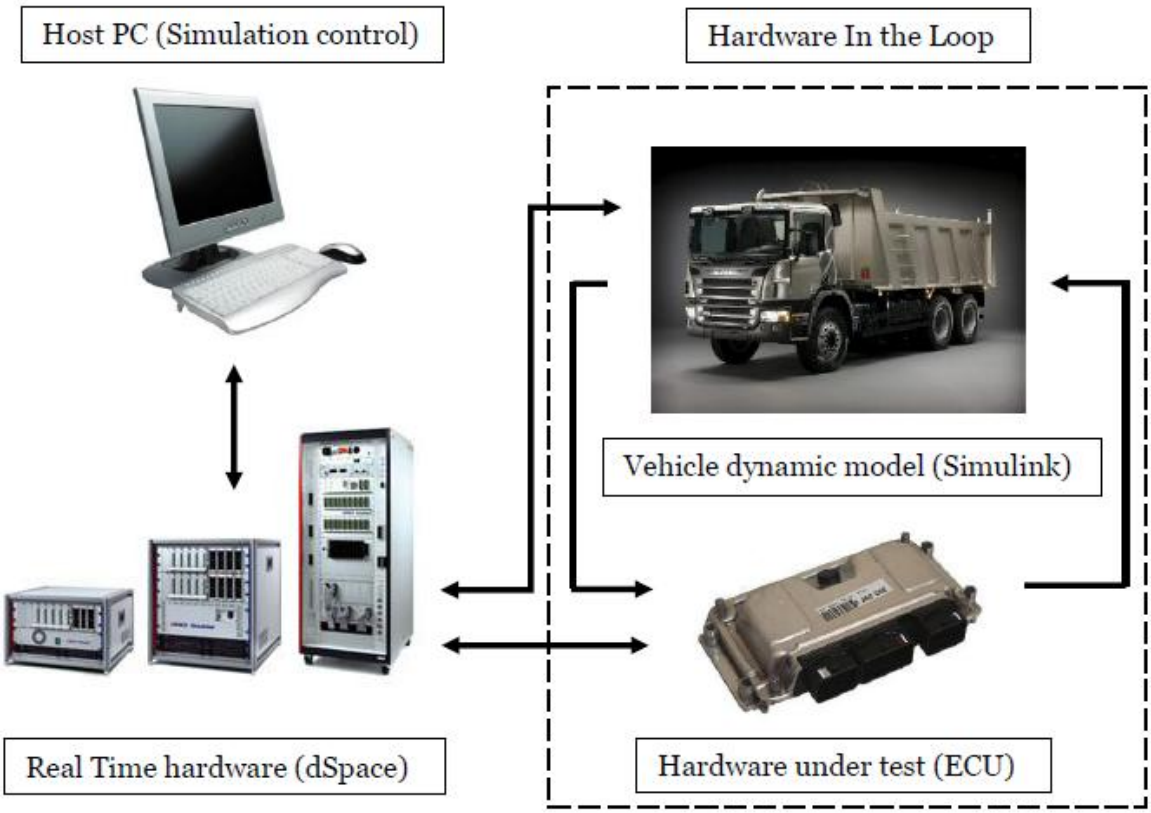


Figure 3: Hardware-In-the-Loop

Testing the vehicle electrical control system consisting of different types of ECU's is a challenge owing to their functional complexity and a large number of vehicle configurations. REST tests ECU's in I-Lab2 and real vehicles parallel.

HIL simulation is not a replacement to the actual test drive. But it saves time by automating the test execution. Also, testing an actual vehicle is expensive. HIL simulations typically consist of performing Virtual test drives. In order to facilitate a virtual test drive, certain variables in the dynamic model are parameterized. Accelerator pedal position, brake pedal position and reference vehicle speed are some common variables that are parameterized. These variables, defined for each time step can then virtually drive the vehicle model. The simulator can handle test drives outside the range of what real vehicles can do, and the tests are reproducible and automatable. Virtual test drives are an immense task for a simulator, which has to handle a complex system model such as the engine or the entire vehicle. Real-time execution capability is mandatory.

2.1 HIL simulation at REST

The HIL simulator I-Lab2 consists of several dSPACE* full size simulators and windows workstations for operating the system. Figure 4 shows I-Lab2 environment.

*dSPACE GmbH is a leading developer, service provider and vendor of HIL systems.

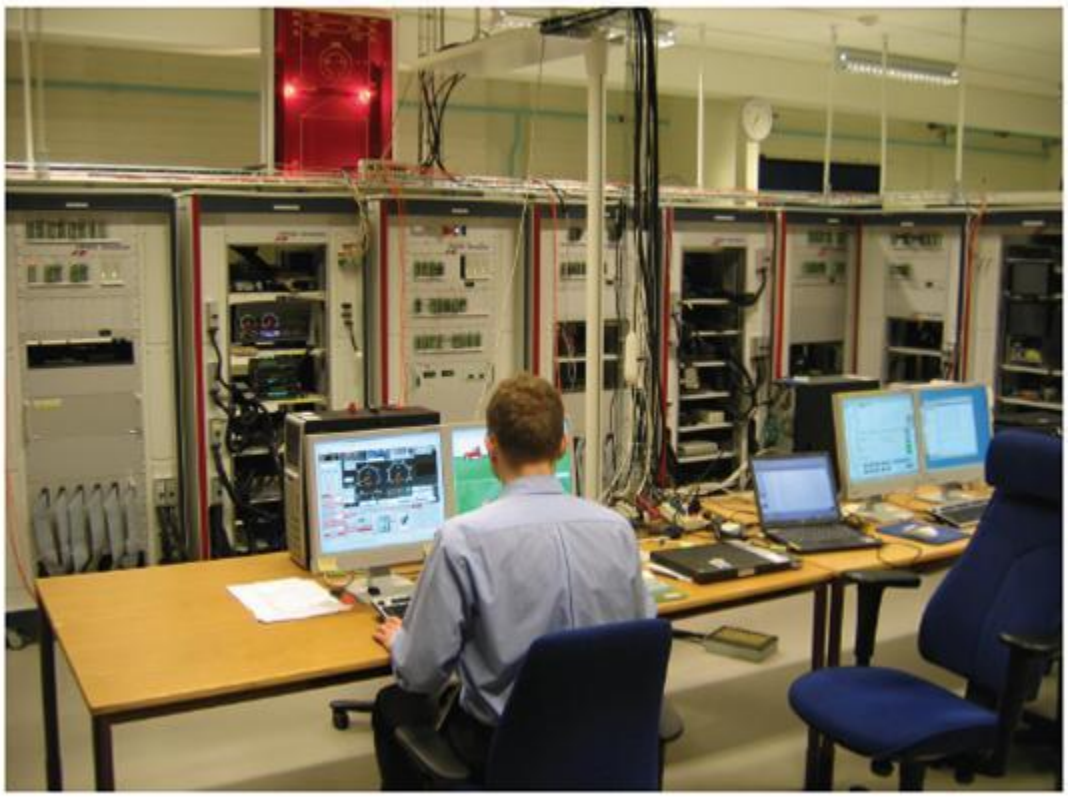


Figure 4: I-lab2 environment

2.1.1 Vehicle dynamic model

HIL simulator in I-Lab2 runs the vehicle dynamic model, called Virtual Truck (VT) shown in Figure 5. The model is implemented using an off-the-shelf Matlab/Simulink model library from dSPACE called Automotive Simulation Models (ASM). It includes dynamic models of the engine, drive train, environment and remaining parts of the vehicle. To improve quality, accuracy, and totality, the model is subject to continuous maintenance.

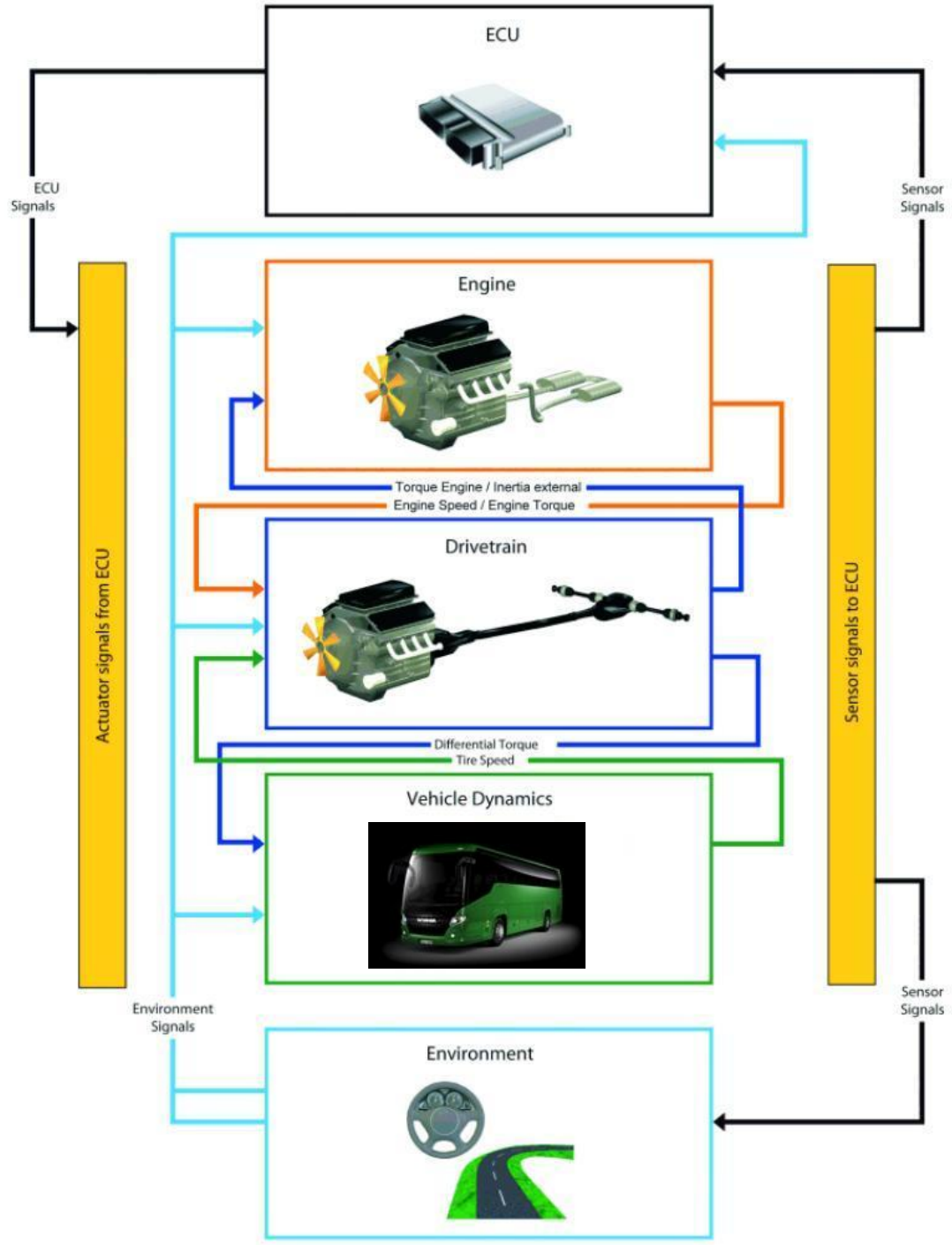


Figure 5: Virtual Truck dynamic model

2.1.2 Tools and interfaces in I-Lab2

2.1.2.1 Matlab/Simulink

Matlab/Simulink, is used for model development.

2.1.2.2 dSPACE Control Desk

dSPACE Control Desk is the central software of the experiment environment. It is a GUI that displays virtual instruments, inputs from sensors, reactions of actuators and CAN messages. All signals are visualized and can be acquired in real time. A Test Engineer can manually drive the VT in Control Desk. Figure 6 shows a screenshot of the virtual instrument panel.

Figure 6: Control Desk screenshot

2.1.2.3 Python and test automation frame work

Test Automation Framework is a set of assumptions, concepts and tools that provide support for automated software testing. I-Lab2 test framework, built on the Python language, offers the following functionalities.

- Test preparation, execution, logging and report creation
- Access to CAN communication and I/Os
- Operating the virtual truck
- Hardware configuration templates

Python is a dynamic object-oriented programming language. Python exploits model parameterization to set variable values and can read variables at every time step of simulation. Loading and execution of a compiled model into real time hardware is also possible. I-Lab2 makes use of these features to automate the test environment.

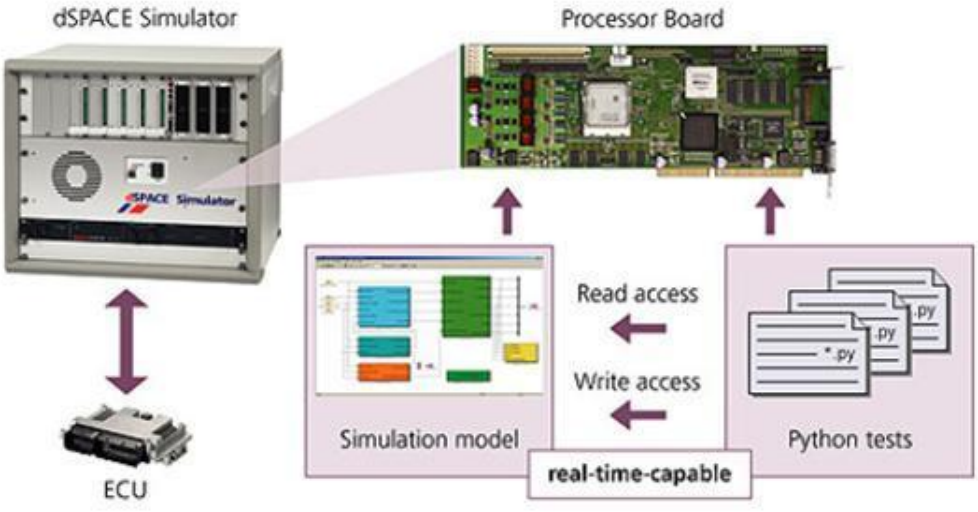


Figure 7: REST test framework and HIL simulator

Chapter 3

I-Lab2 HIL simulator and automated test process is similar to the test environment used by several other vehicle manufacturers [2]. This chapter describes the test execution process at I-Lab2. Test efficiency measurement techniques are discussed and the existing process is evaluated. Recommendations for improving test efficiency are put forward. The chapter concludes with a Problem identification of the research in this thesis.

3 AUTOMATED VEHICLE FUNCTIONAL TEST

Scania defines a large number of User Functions (UF) realized using several ECUs. There are several test cases for each UF. Central locking is an example of a UF with the associated tests being: Lock and Unlock with remote control and Lock and Unlock with Key. Test cases are documented in test design documents which are directly translated to Python scripts. A typical test script contains the following three sections.

- Pre-requisites: Virtual Truck initialization and starting the vehicle.
- Actions: Perform at least one action to change VT's state (e.g. write a value to an I/O or a CAN signal). I/Os are manipulated through Python methods provided by the framework, the resulting CAN messages are read and compared with expected values to generate report.
- Post requisites: Resetting and stopping the vehicle.

3.1 Test process

Before test execution is started, the Test Engineer lists all tests in a .csv file. This forms a test set. The automated test execution process is illustrated in Figure 8. It begins by setting up the HIL simulator and preparing ECU's. Each test case name is fetched by the test framework from the Test set file and the corresponding Python script executed. Within each script, Pre-requisites, actions and Post requisites are performed and the results written to the report file. This process is continued until all tests in the test set are executed.

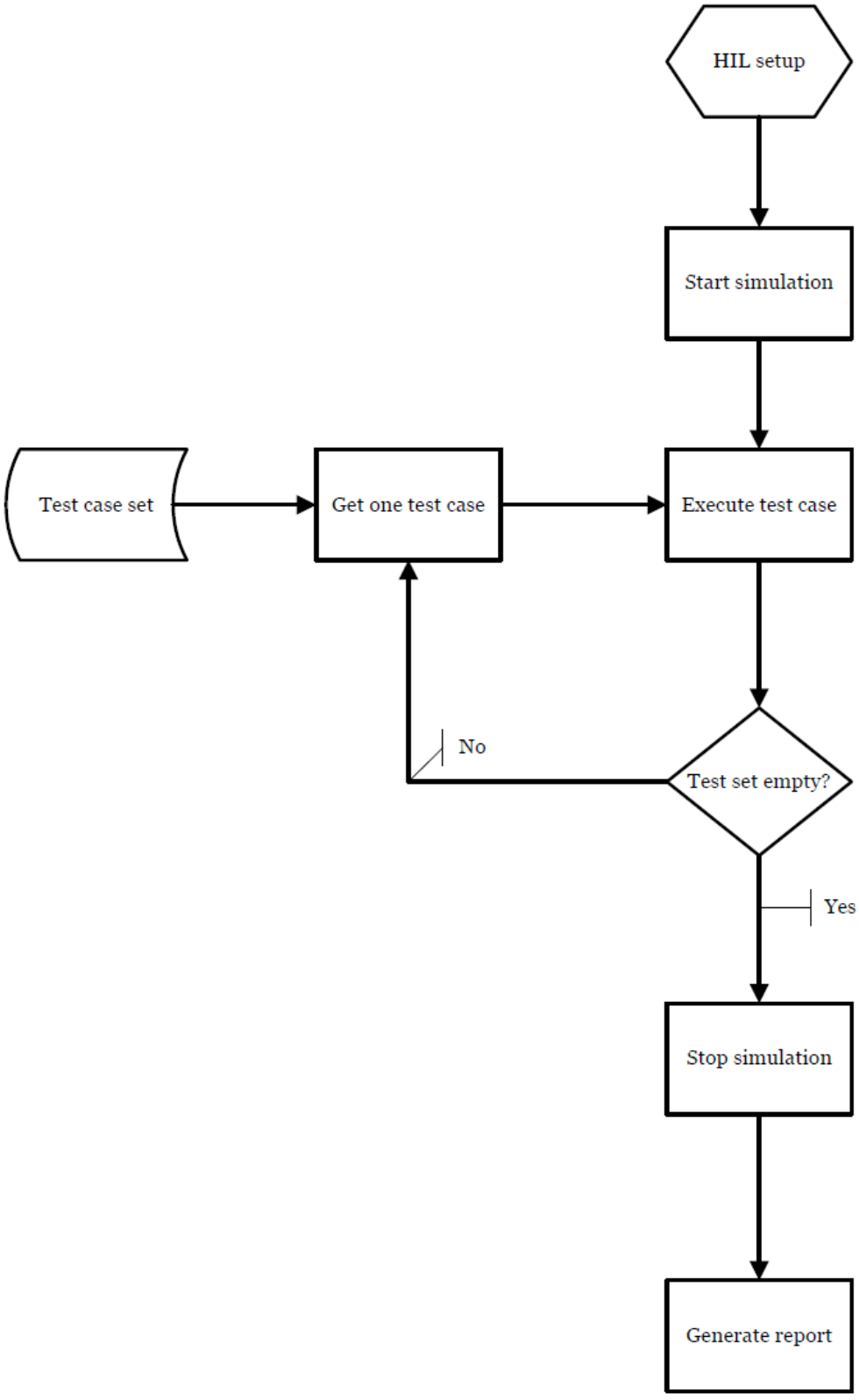


Figure 8: Automated test execution flow

3.2 Simulated test drive Vs Real test drive

Scania Research and Development center owns a test track to perform test of real vehicles. A subgroup within REST is responsible for verification and validation of functionalities in a real vehicle. To illustrate a typical test drive scenario consider the following hypothetical tests of user functions (TC1 to TC10)

1. Central lock with key/remote
2. Seat belt warning
3. High beam with and without Ignition
4. Fuel level display
5. Low engine oil pressure warning
6. Hill hold feature
7. Retarder activation
8. Engage cruise control
9. Disengage cruise control using accelerator/brake/retarder
10. Brake to stop distance at 80 km/h using 25% brake pedal.

Test Engineer/driver designs a test plan before the test drive, describing the actions to be performed and order of execution of functions. An Example of a test plan using TC1 to TC10 described above would be: Check function (1), insert key, perform function (3), Turn ON vehicle, perform functions (2), (3), (4) and (5), drive the vehicle to a hill, check (6), and then (7) while descending, drive on a road at zero grade, check (8) and (9), check (2), (4) and (5) again before braking to stop by performing (10).

However, preparing a test plan in a simulated environment is not an easy task for a Test Engineer. This is due to the growing number of test scripts and the difficulty in sequencing tests like in a real test drive. The Number of vehicle configurations tested is much higher than in an actual test drive and the Test Engineer must do scheduling for each configuration separately. To save time, the normal practice at I-Lab2 is to have the same order of execution for test cases. Since the Virtual truck is started and stopped during each test, random sequencing is possible and ensures independence of test usage in all vehicle configurations.

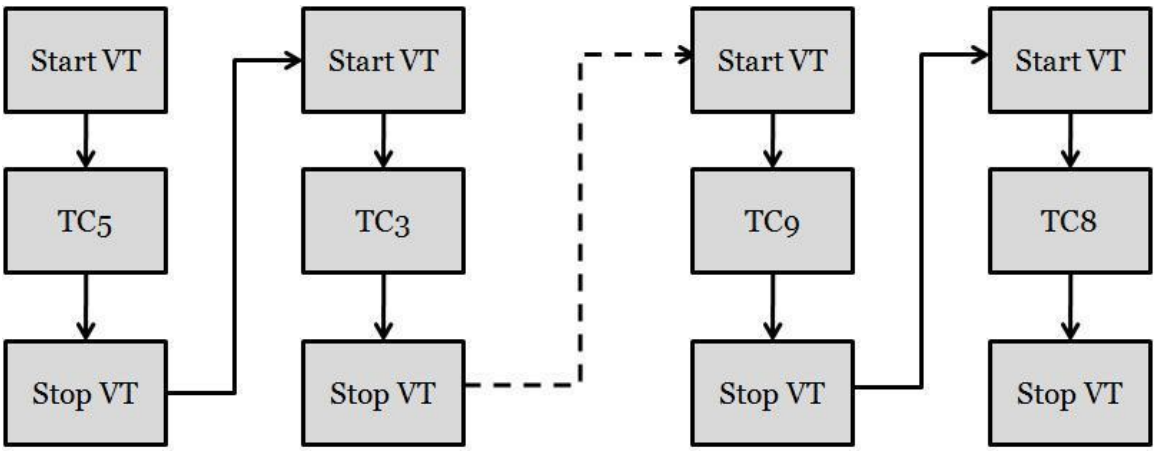


Figure 9: Virtual test drive scenario

This test drive scenario deviates considerably from reality. First, the vehicle is started and stopped for each test. Second, test scheduling is not accordance with the current state of the vehicle. That is, the relationship between a test case and the physical environment in which a vehicle is driven is not used. The state of the vehicle is decided by Key position, speed of the vehicle, speed of engine,

ambient temperature, road grade etc. Current state of the vehicle during simulation is not taken in to consideration while selecting a test case and making transition from one test case to another. For example, the cruise control disengage test could be performed before the cruise control engage test in a real scenario.

3.3 Test process efficiency measurements

Test case efficiency focuses mainly on revealing as many faults in the system as possible. Test process is mainly concerned with how optimal test cases can be scheduled and run, and is focused on factors such as test repeatability and execution time. Fault detection is not a direct goal, though by optimally scheduling test cases it is achieved. Parameters used for measuring test process efficiency depend on the domain where functional test is performed (Automotive or VLSI), type of test activity (unit or functional testing), and the test philosophy followed by the organization.

3.3.1 Quantitative measurements

At Scania I-Lab2, test process efficiency is evaluated primarily by the percentage of test cases that failed or aborted. After the test execution, an analysis is performed on what caused the test cases to fail. Failures due to ECU's are reported to the development team. There could be other reasons for test failure. One type of failure arises because of test cases sharing the same functionality. If the cruise control engage test fails, it is evident that cruise control

disengage test would fail. Test process at I-Lab2 aims to maximize number of tests that fail due to ECUs and minimize that due to shared functionality among test cases. If we define:

N_{TCF} – Number of test cases that failed

N_{TCA} – Number of test cases that were aborted

N_{TCT} – Total number of test cases.

η_{TEFF} – Test efficiency.

Then, $\eta_{TEFF} = \left(1 - \frac{(NTCF+NTCA)}{NTCT}\right) * 100$

FLT_k - Faults that can occur in a system (k is any integer)

TC_k – k-th test case.

TRQ_k – Test case execution requirements, defined in terms of state variables in the vehicle model which should be set to a particular value before the test itself could be executed. For example, minimum vehicle speed should be 20 km/h before testing cruise control. When it is not possible to set a TRQ to a particular value, due to a system fault, TC's using that TRQ fail as in Figure 10.

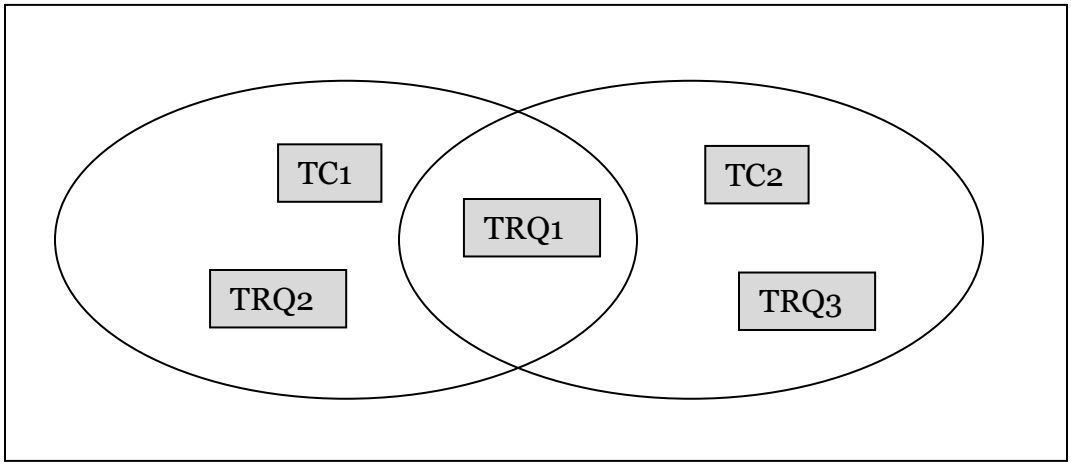


Figure 10: Multiple test failures

In Figure 10, Failure of TC1 due to TRQ1 implies that TC2 will also fail and hence can be withdrawn from the Test set even before it's execution. This reduces (N_{TCF} + N_{TCA}). Current test philosophy in I-Lab2 is to continue testing TC2 despite a failure in TC1. This is because of the need to analyze the other causes for failure, if any.

Test cost, in terms of test execution time and test design time, can be considered as a measure of test efficiency. This is discussed by W. Eric Wong, Joseph R. Horgan et al. in [3]. Time reduction is also beneficial since there is an increase in the number of test cases and a limitation in the availability of hardware resources. In this context, maximizing η_{TEFF} minimizes Test cost.

3.3.2 Qualitative measurements

In the automotive industry, considerable effort is being invested to bring HIL simulations close to real vehicle behavior. Faults that occur only in test drives are not reproducible in simulation due to assumptions and simplifications made during modeling and simulation. It is expected that simulated test drives that are close to reality can reveal more faults.

3.4 Test process evaluation

We will experimentally measure test efficiency using Tc1 to TC10 described in Sec. 3.2. A failure was simulated in the cruise control engage function. As a result, TC8 and TC9 failed. According to the existing test philosophy,

$N_{TCF}+N_{TCA}=2$ and $\eta_{TEFF} = 80\%$. If TC9 is prevented from execution as it is known to fail, $\eta_{TEFF} = 89\ \%$. It highly depends on which test cases failed and how many other test cases are dependent on the failed test cases. But removing test cases known to fail is guaranteed to improve efficiency.

Improvement also comes by saving total test time. When TC9 is excluded total test time is reduced by 10%. As before, this cannot hold for all test scenarios.

Chapter 4

Chapter 3 discussed the Test process at Ilab2 and evaluated it using a test scenario. This chapter suggests solutions to improve the process and introduces a new test philosophy to make testing in I-Lab2 more realistic. The literature survey done on several related topics is discussed.

4 DYNAMIC TEST SCHEDULING

Based on the discussion about the I-Lab2 test process in Chapter 3, the following problems are considered for further investigation.

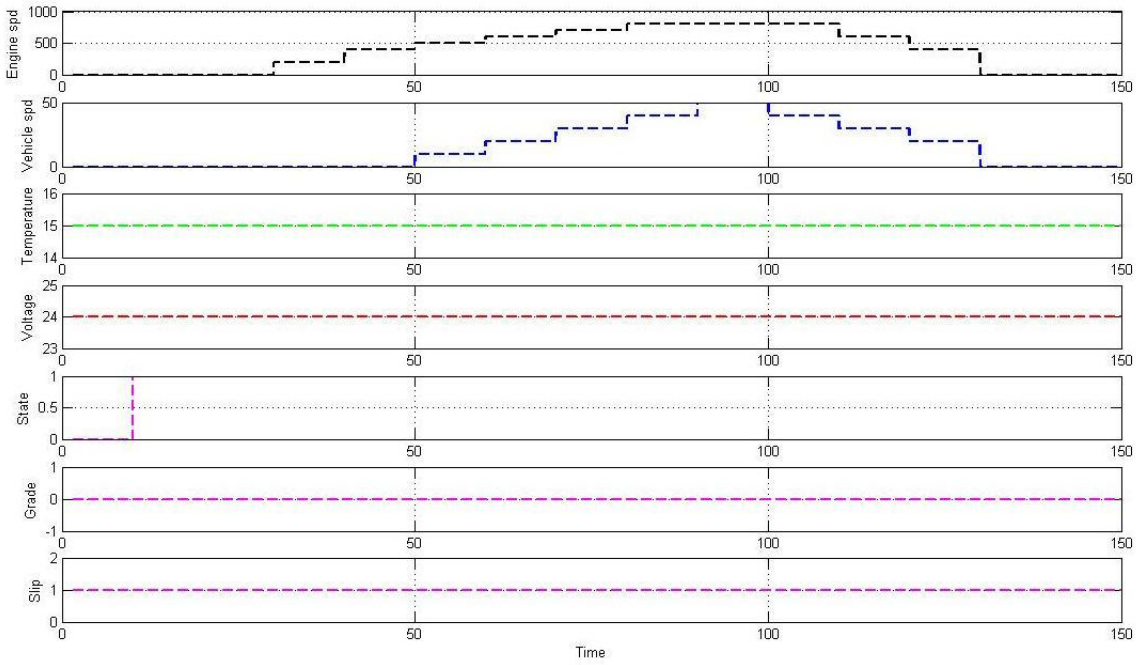
- Test case (TC) scheduling
- Reality in virtual drive scenarios

Solving the above problems has the following advantages.

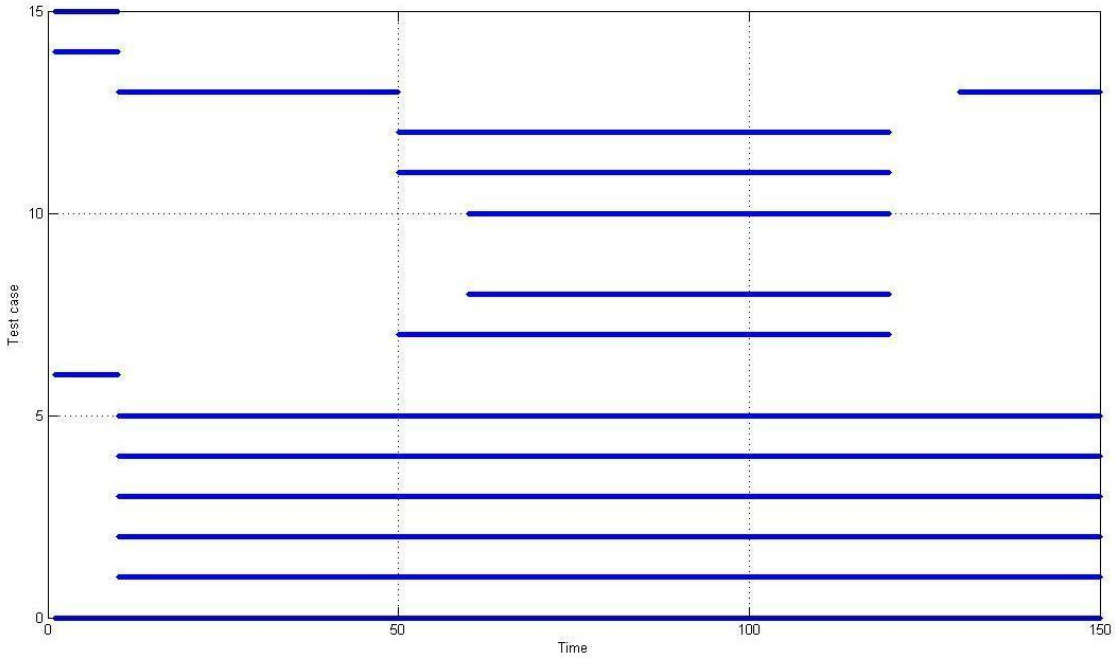
- Brings simulated test drive scenario closer to a real test drive by executing test cases in a virtual drive environment
- Reduce burden on Test Engineer using automated test scheduling
- Save test suite execution time by preventing test cases from executing if it is known to fail due to functional dependency.

4.1 Test scheduling

Scheduling is the process of deciding how to commit resources between a variety of possible tasks. Scheduling can be offline or online and designed to adapt to a dynamic environment. User function testing using HIL simulation can be viewed as a dynamic scheduling problem with a time and simulation environment, committed to a finite number of test cases. In I-Lab2, the resources set consists of ECU's under test and Real time hardware (dSPACE processor and IO boards) with associated software and time. The Dynamic test scheduling concept is illustrated in Figure 11. For every instance of vehicle state (consisting of several variables) the scheduler selects a set of suitable test cases.



Simulation environment (state variables)



Test case selections

Figure 11: Dynamic test scheduling concept

Dynamic test scheduling and execution process has the following features.

- Task scheduling is online since execution of a given TC depends on the current state of the vehicle.
- Due to dynamic behavior of the vehicle control systems, time taken to finish a task (TC) cannot be predicted. It can only be estimated.
- More than one test case might be eligible for execution for a given state of the vehicle.
- In I-Lab2, no two TC's can be executed at the same time. A given TC needs 100% commitment from simulation software and hardware resources while executing. In a real test drive, two UFTs can be performed at the same time in some instances.
- Test case execution itself is a dynamic process, evolving over time and driving the vehicle from one state to another.

4.1.1 Mathematical representation of Test scheduling problem

Consider a set of *N* test cases, numbered 1, 2, …, N, with N as the terminal point as shown in Fig. 12.

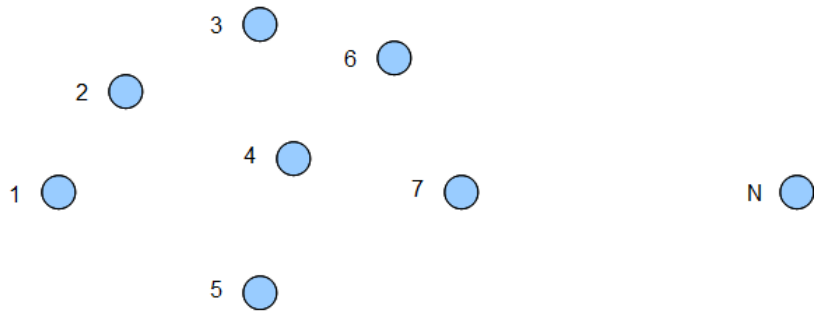


Figure 12: Test cases as N points in test space

Suppose that there exists a direct link between any two points i and j which require a cost C_{ij} to traverse, assuming that staying in the same state is minimal cost, such that C_{ii} = 0, i = 1,2,...,N. In all the following discussions $C_{ij} \geq 0$. The aim is to find a path starting at 1, passing through some subsets of points 2,3,..., (N-1) and terminating at N, involving minimum total cost, *C*. We are required to minimize the expression

$$C(i_1, i_2, i_3, ..., i_k) = C_{1i_1} + C_{i_1 i_2} + C_{i_2 i_3} + ... + C_{i_k N},$$

Where $(i_1, i_2, i_3, ..., i_k)$ is some subset of (2,3,....,N-1).

Problems of this type can be treated by means of dynamic programming.

Let f_i = minimum cost to go from i to N, i = 1,2,..., (N-1).

Using principle of optimality (R. Bellman [25]).

$f_i = \min_{j \neq i} \left[C_{ij} + f_j \right]$, i = 1,2,..., (N-1), with the boundary condition $f_N = 0$.

The unknown function *f* occurs at both sides of the equation. But using dynamic programming, f_i can be broken down to several smaller sub problems, similar in nature until f_i represents the cost of transition from one node to another. This basic unit has to be given an initial value based on assumptions or an initial condition. Solution of f_i has upper and lower bounds. The lower bound is given by the sequence:

$\left\{ \phi_i^{(r)} \right\}$, r = 0,1,2 defined for each i in the following fashion:

$$\phi_i^{(0)} = \min_{j \neq i} (c_{ij}) \qquad i = 1,2,..., (N-1) \qquad \phi_N^{(0)} = 0$$

$$\phi_i^{(r+1)} = \min_{j \neq i} (c_{ij} + \phi_j^{(r)}) \qquad i = 1,2,..., (N-1) \qquad \phi_N^{(r+1)} = 0.$$

$\{\phi_i^{(r)}\}$ is uniformly bounded, monotone increasing and $\phi_i \le f_i$

Upper bound is given by the sequence:

$\{\psi_i^{(r)}\}$ r = 0,1,2 defined for each i in the following fashion:

$\psi_i^{(0)} = c_{iN}$ $\quad\quad$ i = 1,2,..., N

$\psi_i^{(r+1)} = \min_{j \neq i}(c_{ij} + \psi_j^{(r)})$ $\quad\quad$ i = 1,2,..., N

$\{\psi_i^{(r)}\}$ is a monotone decreasing sequence, converges and $\psi_i \ge f_i$.

Thus we have $\phi_i \le f_i \le \psi_i$, i = 1,2,..., N, where ϕ_i is the lower bound and ψ_i is the upper bound of f_i. The example below from illustrates the scheduling problem discussed above (Bellman-Ford shortest path algorithm)

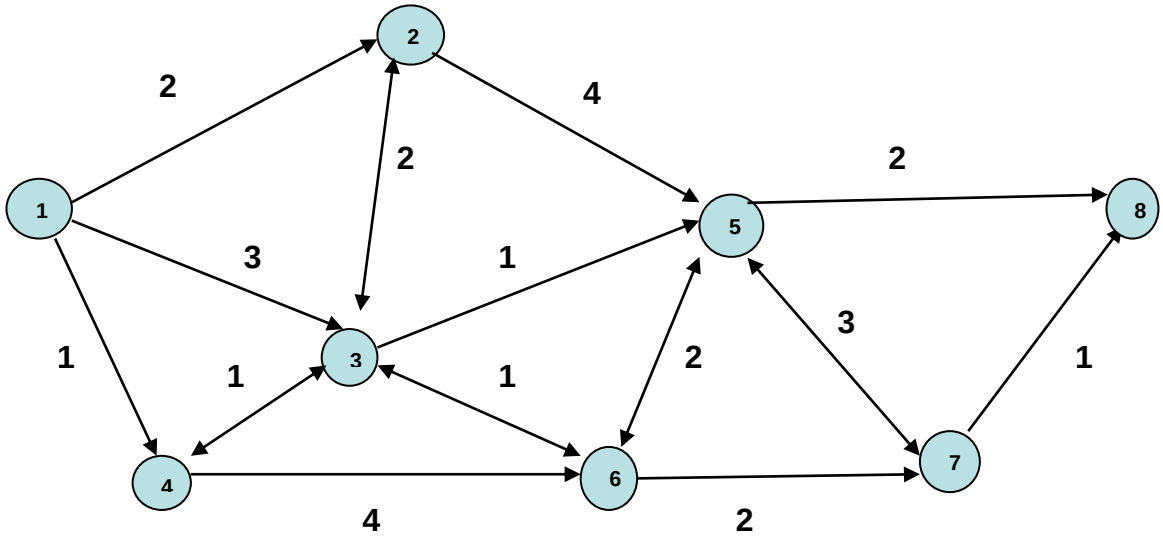


Figure 13. Illustration of test scheduling using iteration

The sequences $\{\phi_i^{(r)}\}$ and $\{\psi_i^{(r)}\}$ can be calculated iteratively and are given below. Thus, $f_i = \phi_i^3 = \psi_i^3$.

i	$\phi_i^{(0)}$	$\phi_i^{(1)}$	$\phi_i^{(2)}$	$\phi_i^{(3)}$	$\phi_i^{(4)}$		$\psi_i^{(0)}$	$\psi_i^{(1)}$	$\psi_i^{(2)}$	$\psi_i^{(3)}$	$\psi_i^{(4)}$
1	1	2	3	5	5		∞	∞	6	5	5
2	2	3	5	5	5		∞	6	5	5	5
3	1	3	3	3	3		∞	3	3	3	3
4	1	2	4	4	4		∞	∞	4	4	4
5	2	2	2	2	2		2	2	2	2	2
6	2	3	3	3	3		∞	3	3	3	3
7	1	1	1	1	1		1	1	1	1	1
8	0	0	0	0	0		0	0	0	0	0

The test scheduling problem discussed in this thesis tries to minimize the cost in terms of test execution time and by reducing test failures due to interdependency of tests.

4.1.2 Scheduling techniques

One of the first steps to solving dynamic test scheduling problems was to explore similar problems in other areas. Scheduling problems exist in a wide variety of domains from Real Manufacturing systems to Fault diagnosis. Dedicated algorithms exist in these areas. However, many parallels can be drawn among these algorithms. Some of these algorithms will be discussed now. Selection of a technique to solve the test scheduling problem depends on its ability to handle test case's features discussed in Section 4.1. Since the scheduling

has to be online, one criteria is to keep it computationally simple. Implementation and maintainability constraints are also considered.

4.1.2.1 Scheduling in Manufacturing Systems

In manufacturing, the purpose of scheduling is to minimize production time and costs, by telling a production facility what to make, when, and on which equipment. Various scheduling algorithms are considered by Pinedo [5]. Scheduling algorithms try to minimize costs, in terms of total completion time taken to process all jobs. Most schedulers in manufacturing are static or offline and online schedulers in manufacturing tend to be complex. This makes them unsuitable for use in dynamic or online test scheduling.

4.1.2.2 Test sequencing in fault diagnosis

In fault diagnosis, a series of tests are performed on the systems to locate a fault. The goal is to minimize the number of tests performed and hence minimize some cost function. In [6], Pattipati et. al describes graph search based methods to find near optimal solutions. Both single and multiple faults can be isolated using this algorithm. These studies mainly focus on test sequencing problems for diagnosing systems during the operational phase of the systems as described in [7]. The Test case selection problem is not combinatorial, in contrast to problems discussed in manufacturing and fault diagnosis. Hence, test sequencing problems during system and integration testing require a different approach.

4.1.2.3 Finite State Machines

A Finite State Machine (FSM) is an abstract model of a dynamic system with a primitive internal memory. It consists of (1), States which define behavior of the system that may produce actions (2) State transitions, which is a movement from one state to another (3) Rules or conditions, which must be met to allow a state transition, and (4) Input events generated either internally or externally, which may possibly trigger rules and lead to state transitions

Finite State Automaton (FSA) is a special kind of finite-state machine. Here, state machine as a dynamic system does not produce any output itself. Instead, state transitions terminate in one or more states called accepting states. The Test execution process can be viewed as an FSA. This concept is illustrated in Fig. 14.

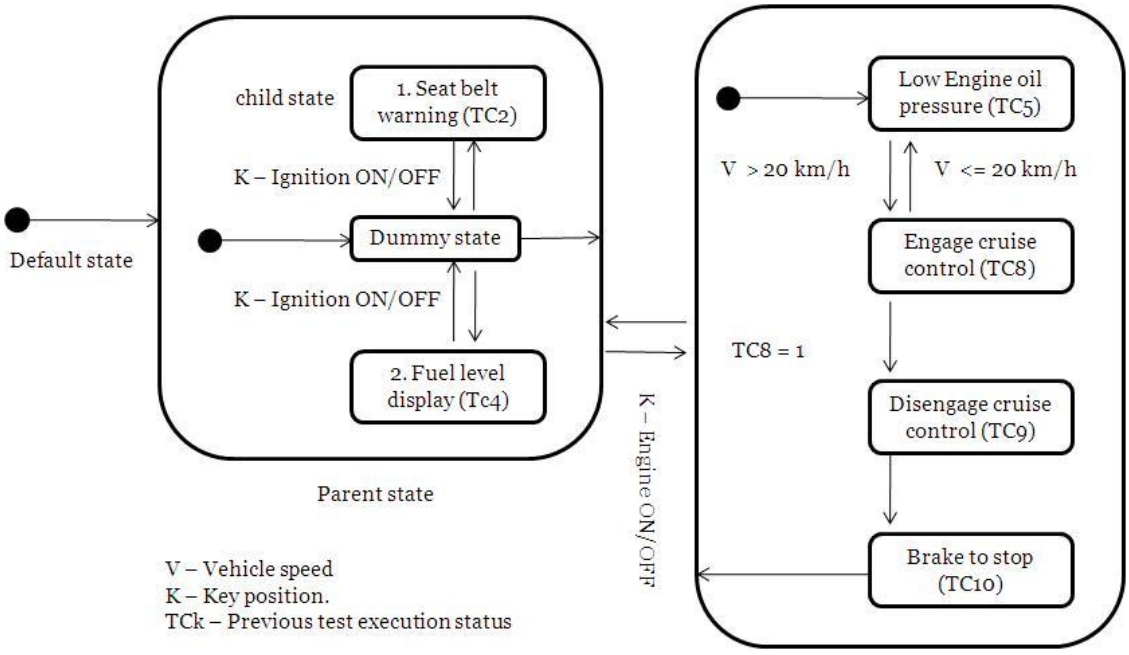


Figure 14: FSA concept in vehicle testing

Referring to Figure 14, Each test case is a state whose execution depends on entry conditions defined in the Pre-requisites of a function test. Entry conditions to a state are defined by two input variables: key position and vehicle speed. The State transition table is given in Figure 15. Don't care conditions are indicated by an 'x'. Transitions are read in the format: *if current state and input, then next state*. As an example, if current state is Dummy and input is {Ignition ON,0} then next state is the set of tests {TC2,TC4}.

Inputs: {Key position, Speed}\States	Dummy	TC2	TC4	TC5	TC8	TC9	TC10
{Ignition OFF, Speed =0}	Dummy	Dummy	Dummy	x	x	x	x
{Ignition ON, Speed=0}	{TC2, TC4}	TC2	TC4	Dummy	x	x	TC5
{Engine ON, 0 <= Speed <=20 }	TC5	x	x	TC5	TC5	x	x
{Engine ON, Speed >20}	x	x	x	TC8	TC9	TC10	x

Figure 15: State transition table for vehicle test scenario

A Dummy state is included to exclude test cases from being a default state. TC10 and TC9 are executed based on the execution statuses of previous test cases. The FSA, is non-deterministic since there are instances when a new input causes a machine to be in more than one state (Dummy to {TC2, TC4}). In practice, only one state can be executed and conflict resolution strategies must be used. One solution is to use Failure Mode and Effects Analysis (FMEA).

FMEA is not new to the test process. When user functions are designed they are assigned an FMEA level, which are decided based on how critical the functions are for safe operation of the vehicle. A system with a high FMEA

number is more critical compared to another system with a lower FMEA number. An example is given in Figure 16. A function test that belong to a user function with a higher FMEA number may be tested first. When two selected tests have the same FMEA level, one could resort to a random selection.

Function	FMEA
Brake to stop	5
Engage Hill hold	5
Activate retarder	4
High beam with Ignition	3
Fuel level display	2

Figure 16: FMEA level for user functions

4.2 Reality in virtual test scenarios

The second area identified for improvement at the beginning of this chapter was moving the virtual drive scenario closer to reality by attempting to imitate a real test drive.

4.2.1 Modification of existing test philosophy

I-Lab2 test philosophy during Systems and Integration test is dictated by Regression testing. Regression testing is done to verify functionality of the electrical system after new changes or error fixing in the control software. This is

done by testing the whole electrical system and executing the same test scripts before and after the change. Test scripts that passed before the change should also pass after.

Test scripts start the virtual vehicle, perform actions, and stop the vehicle. This prevents faults propagating to other test cases. This might not happen in a real test drive when the vehicle is not stopped after every test case. A fault that occurs during the execution of one test case might leave its foot print on the electrical system that might affect the next test case. If the vehicle can be continuously driven in the simulation, we have a possibility to study how function tests are related. It was decided to explore this interesting behavior during testing as part of the research work.

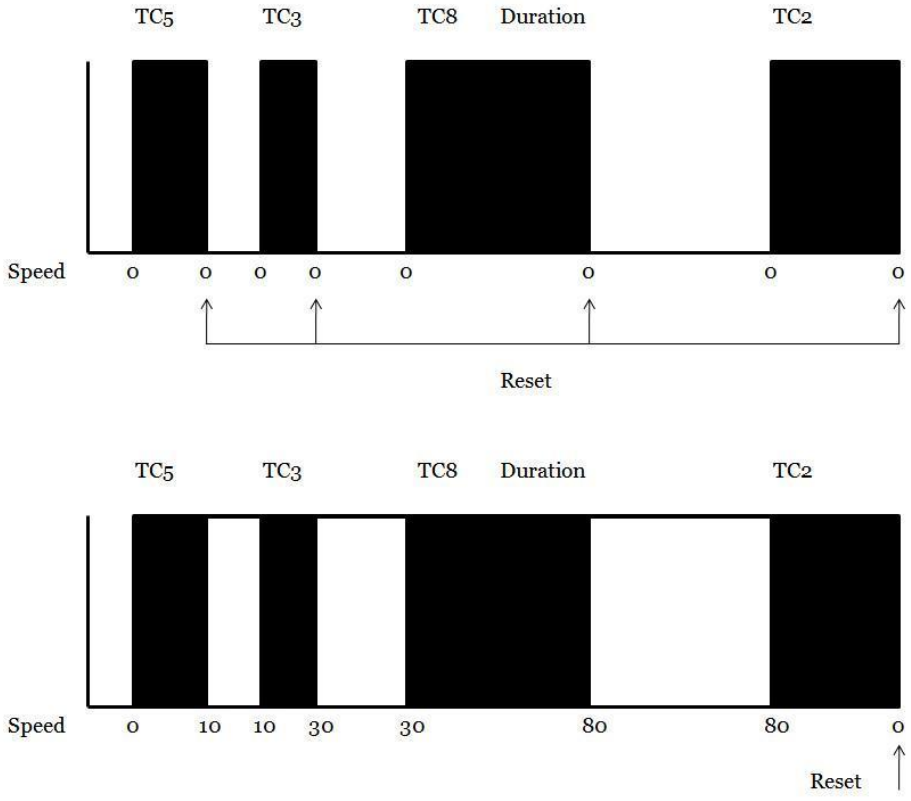


Figure 17: Test case bridging

To create dependency between functional tests, test cases need to be bridged together so that the vehicle is started at the beginning, and stopped and reset after the last test case. This concept is illustrated in Figure 18. Test schedule on top of the figure illustrates the existing process. In the bottom, and when the vehicle is not stopped, the next test case starts with a vehicle speed equal to the speed at the end of the previous test case. Spacing between test cases shows dynamic behavior of scheduling when no test cases could be found suitable for execution.

One simple solution is to remove vehicle start, stop and reset processes from every test script. Two new modules are introduced into the test framework. A maneuver virtually drives the vehicle like a test driver. It is responsible for starting, driving, and stopping the virtual vehicle. The second module is called a scheduler. It functions like a test engineer in the real world and is responsible for scheduling test cases as discussed in Section 4.1. Test framework is still responsible for executing the test.

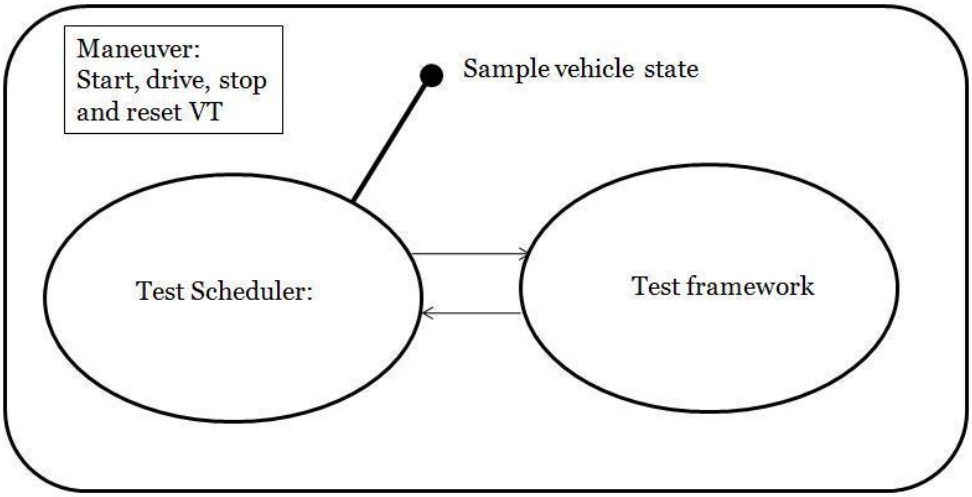


Figure 18: Maneuver and Scheduler in test environment

Test scheduler samples the vehicle state when the vehicle is driven by the maneuver module. When a test case is being executed, test case assumes control of vehicle maneuver by means of actions defined for the test case, until it terminates.

Chapter 5

Previous chapters presented concepts of scheduling and virtual maneuvering to solve the dynamic scheduling problem in the I-Lab2 test process. This chapter details the theoretical framework necessary to implement the solution and discusses how it was performed.

5 SOLUTION FRAMEWORK

Methods by which the dynamic test scheduling problem could be addressed were discussed in Chapter 4. To implement the solution in practice and test it we need a theoretical framework, which is decomposed into the following components.

1. Scheduler module modeled as a state automaton.
2. Transition conflict resolving algorithm.
3. Maneuver module that can start, drive and stop the virtual vehicle.

5.1 Scheduler module

Test scheduler is modeled as a State automaton with each test case representing a state. Because of its dynamic behavior, it can also be called a Dynamic test scheduler. The Following general definitions are used while formulating the problem as an FSA. *E(t)* is a set of real time values of vehicle state (also termed as vehicle environment) sampled at predetermined instances during simulation time t. This is done by the scheduler from the simulation environment when the virtual vehicle is driven. *E* is one sample of *E(t)*

$E(t) = \{K(t), V(t), \Omega(t), v(t), \theta(t), \alpha(t), \zeta(t)\}, 0 \leq t \leq \Delta,$

Δ is the total simulation time.

K – Key position, K = {0,1,2} for {Ignition OFF, Ignition ON, Engine ON}

V – Vehicle voltage in volts

Ω – Engine speed in revolutions per minute (RPM)

v – Vehicle speed in km/h

θ – Ambient temperature in degree Celsius

α – Road gradient in percentage

ζ – Road frictional co-efficient.

An example of a real time vehicle state is given in Appendix-A1. The seven variables in the set were considered after carefully screening about 50 variables. Many of them could be removed due to their negligible impact on test case selection.

The second variable E_i is the set of test case Pre-requisites defined for each test case design document. This should not be confused with *E(t)* or *E*

$E_i = \{K_i, V_i, \Omega_i, v_i, \theta_i, \alpha_i, \zeta_i\}$, $i = 1,..,N$. where N is the total number of test cases. Each element has an upper and lower limit.

Eg: $v_1 = \{v_{L1}, v_{H1}\} = \{70,100\}$.

An example of a test case requirement table is given in Appendix-A2.

5.1.1 Scheduler design using Finite State Automata (FSA)

Basic information about state machines is given in the Definitions section of this thesis. For further reading, one can use text books given in References [22] section. To simplify the problem formulation, only ten test cases listed in Section 3.2 will be considered. They are repeated here for convenience.

1. Central lock with key/remote
2. Seat belt warning
3. High beam with and without Ignition
4. Fuel level display
5. Low engine oil pressure warning
6. Hill hold feature
7. Retarder activation
8. Engage cruise control
9. Disengage cruise control using accelerator/brake/retarder
10. Brake to stop distance at 80 km/h using 25% brake pedal.

We begin with the general definition of a non deterministic FSA.

A non- deterministic FSA is a 5-tuple, $A = (S, I, \nu, T, t_1)$, where:

S = {TC1, TC2,...TC10}, is the finite set of states for A

I = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, is the finite input alphabet for A

$\nu : S \times I \rightarrow P(S)$ is the next-state function

$P(S)$ is the collection of all subsets of S and hence is a function of S

$T \subseteq S$, $\{\} \notin T$, final state(s) and $t_1 \in S$, initial state

A hierarchal state automaton design is suggested containing a set of parent states and each parent state consisting of child states (test cases). For simplicity, a given test case is assumed to be present in only one parent state. Transitions within parent states are based on the vehicle key position 'K' in the set E.

Transitions within child states are based on the evaluation of four binary variables for all test cases as shown below.

$\bar{v}_i = 1 \leftrightarrow v_{Li} < v \leq v_{Hi}$ and $\bar{v}_i = 0,\ otherwise$

$\bar{\Omega}_i = 1 \leftrightarrow \Omega_{Li} < \Omega \leq \Omega_{Hi}$ and $\bar{\Omega}_i = 0,\ otherwise$

$\bar{\alpha}_i = 1 \leftrightarrow \alpha_{Li} < \alpha \leq \alpha_{Hi}$ and $\bar{\alpha}_i = 0,\ otherwise$

$\Psi_i = 1 \leftrightarrow (\bar{v}_i\ \&\ \bar{\Omega}_i\ \&\ \bar{\alpha}_i) = 1$ and $\Psi_i = 0,\ otherwise$

i = 1,….,10

Ψ_i is introduced to simplify the expression

In test cases TC1 to TC10, Voltage V_i, Temperature θ_i, and Frictional co-efficient ζ_i are not defined in their test cases pre-requisites. Therefore, the variables, $\bar{V}_i$, $\bar{\theta}_i$, $\bar{\zeta}_i$ do not affect state transitions and can be omitted from the design. The set *S,* is appended with three dummy test cases TD1, TD2 and TD3 to serve as default/initial states in three parent states defined below.

$\bar{S}_1 = \{TC_1, TD_1\}$

$\bar{S}_2 = \{TC_2, TC_3, TC_4, TC_5, TD_2\}$

$\bar{S}_3 = \{TC_6, TC_7, TC_8, TC_9, TC_{10}, TD_3\}$

and $\bar{S} = \{S \cup \{TD_1, TD_2, TD_3\} = \{\bar{S}_1, \bar{S}_2, \bar{S}_3\}$

The hierarchal design has a main state automaton, $\bar{M} = (\bar{S}, I, \nu, \{\bar{S}_3, \bar{S}_1\}, \bar{S}_1)$, defined by three sub automatons, $\bar{M}_1, \bar{M}_2, \bar{M}_3$.

$\bar{M}_1 = (\bar{S}_1, I, \nu, \{TC_1\},\ TD_1)$,

$\bar{M}_2 = (\bar{S}_2, I, \nu, \{TC_2, TC_3, TC_4, TC_5\},\ TD_2)$

$\bar{M}_3 = (\bar{S}_3, I, \nu, \{TC_6, TC_7, TC_8,\ TC_{9,}\ TC_{10,}\},\ TD_3)$

State transitions for parent and child states will be now stated.

State automaton $\bar{M} = (\bar{S}, I, \nu, \bar{S}_3, \bar{S}_1)$, state transition table.

$\nu: \bar{S} x I \rightarrow P(\bar{S})$	K = 1	K = 2	K = 3
$\rightarrow \bar{S}_1$	$\bar{S}_1$	$\bar{S}_2$	n
$\bar{S}_2$	$\bar{S}_1$	$\bar{S}_2$	$\bar{S}_3$
$* \bar{S}_3$	*n-(invalid)*	$\bar{S}_2$	$\bar{S}_3$

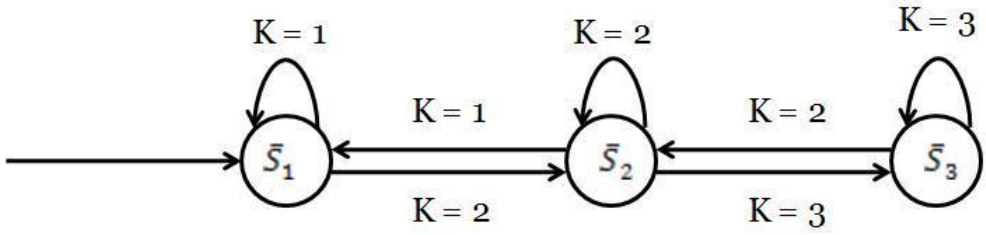


Figure 19: Scheduler state automaton, Parent automaton transitions

For the state automaton $\bar{M}_1 = (\bar{S}_1, I, \nu, \{TC_1\}, TD_1)$, state transitions are as defined below.

$\nu: \bar{S}_1 x\, I \rightarrow P(\bar{S}_1)$	$\Psi_1 = 1$	$\Psi_1 = 0$
$\rightarrow TD_1$	TC_1	TD_1
$* TC_1$	TC_1	TD_1

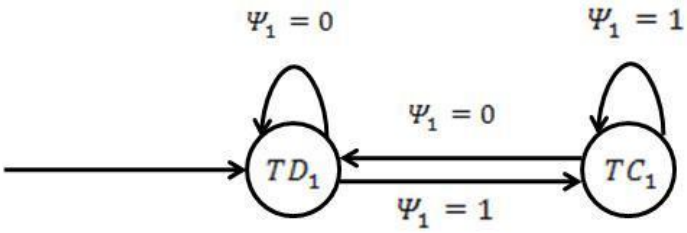


Figure 20: Scheduler state automaton, child transitions (a)

For the state automaton $\bar{M}_2 = (\bar{S}_2, I, \nu, \{TC_2, TC_3, TC_4, TC_5\}, TD_2)$, state transitions are as defined. To simplify the diagram, state transition criteria are omitted. An arrow connecting TD2 to TC_i means that the transition occurs from TD2 to TC_i when $\Psi_i = 1$, and from TC_i to TD2 when $\Psi_i = 0$.

$\nu: \bar{S}_2 x\, I \rightarrow P(\bar{S}_2)$	$\Psi_i = 1\ (i = 2,3,4,5)$	$\Psi_i = 0$
$\rightarrow\ TD_2$	$\{TC_2, TC_3, TC_4, TC_5\}$	TD_2
* $\{TC_2, TC_3, TC_4, TC_5\}$	$\{TC_2, TC_3, TC_4, TC_5\}$	TD_2

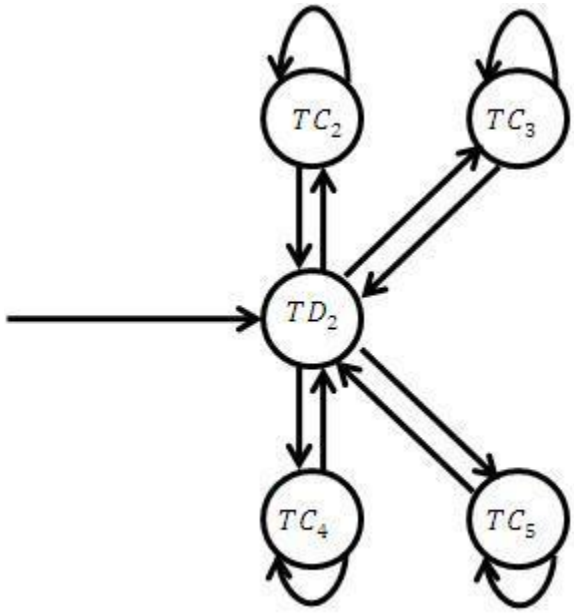


Figure 21: Scheduler state automaton, child transitions (b)

For the state automaton $\bar{M}_3 = (\bar{S}_3, I, \nu, \{TC_6, TC_7, \{TC_8, TC_9\}, TC_{10}\}, TD_3)$, state transitions are as defined below. TC9 and TC8 occur in pairs. This is taken care of in the state transition by grouping them together as a subset.

$v: \bar{S}_3 x\ I\ \rightarrow P(\bar{S}_3)$	$\Psi_i = 1\ (i = 6,7,8,9,10)$	$\Psi_i = 0$
$\rightarrow\ TD_3$	$\{TC_6, TC_7, \{* TC_8, TC_9\}, TC_{10}\}$	TD_3
$*\ \{TC_6, TC_7, \{* TC_8, TC_9\}, TC_{10}\}$	$\{TC_6, TC_7, \{*\ TC_8, TC_9\}, TC_{10}\}$	TD_3

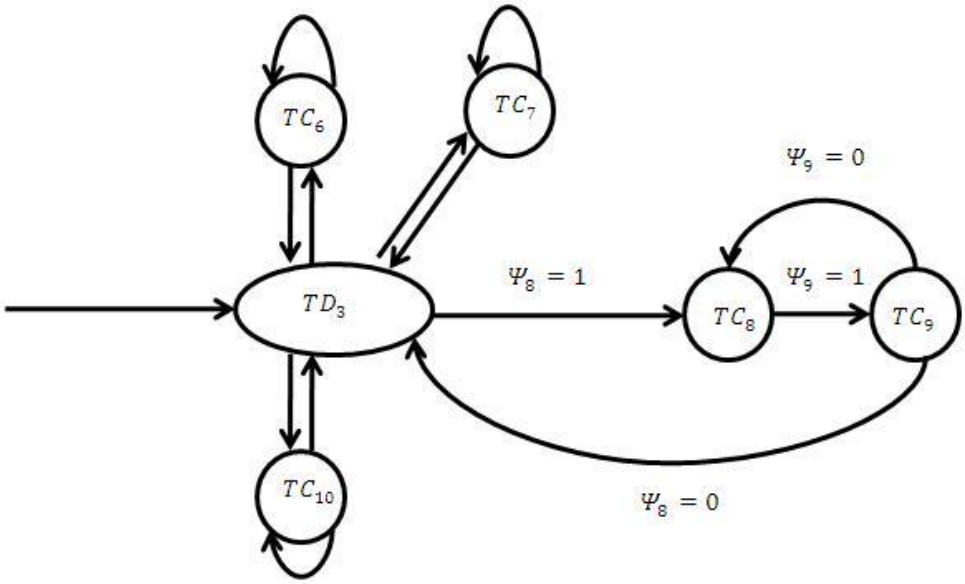


Figure 22: Scheduler state automaton, child transitions (c)

Test scheduler theory described above was implemented in Matlab/Simulink and evaluated for its suitability to be used in the test execution process. Simulink has a state flow block set in its library to implement FSA. State charts in the library act as subsystems in a Simulink model, which accept State inputs and execute actions specified in each state. The actions could be as simple as returning the test case number itself, or could be executing the test case itself. Each input variable to state chart are in the form of two dimensional matrices with one dimension holding time values and the other holding variable values.

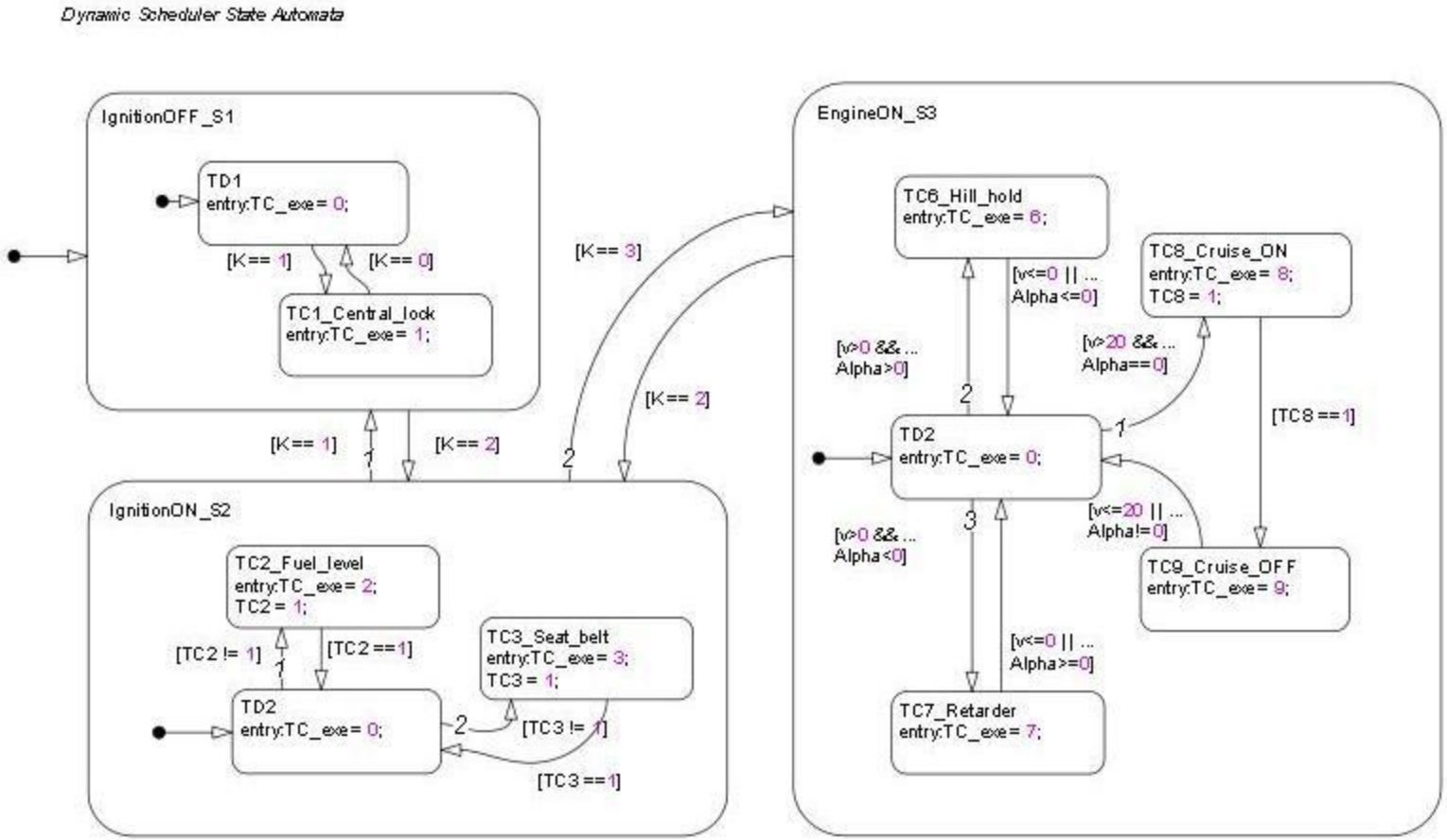


Figure 23: Dynamic scheduler state automaton in Simulink

Set1.

Time	K	v	α	TC
1	1	0	0	1
2	2	0	0	2
3	2	0	0	3
4	3	10	1	6
5	3	10	-1	7
6	3	30	0	8
7	3	30	0	9

Set2.

Time	K	v	α	TC
1	1	0	0	1
2	2	0	0	2->3
3	3	30	-1	7
4	3	30	1	6
5	3	10	0	TD3
6	3	10	0	TD3
7	2	0	0	TD2

Set3.

Time	K	v	α	TC
1	1	0	0	1
2	2	0	0	2
3	2	0	0	3
4	3	30	0	8
5	3	30	0	9
6	2	0	0	TD2
7	1	0	0	1

Set4. Fault simulation

Time	K	v	α	TC
1	3	0	0	TD1
2	3	0	0	TD1
3	2	0	0	2->3
4	3	-10	1	TD3
5	3	10	-1	7
6	1	0	0	TD1
7	2	0	0	TD2

Figure 24: Dynamic scheduler (FSA) simulation results

Referring to Figure 23: Test execution status flags for each test case are defined and set to 1 when that test is executed. This is used to keep track of the execution status of test cases. Within each state, a global variable *TC_exe* is set to the value of the test case that was selected.

We observe that the results shown in Figure 24 are as expected. In *Set1*, the vehicle is started, driven up a hill, the down, and then set to cruise at 30 km/h. Test cases are scheduled by the state automata in the order 1->2->3->6->7->8->9. In S*et2*, TC3 is forced to execute after TC2 by defining the transitions from TD2 that depend on the test flag and not any of the pre-requisites. In *Set3* use of test flags is more evident. When K makes a transition from 3 to 2, TC3 and TC2 executions are not repeated since transition to them depend on the previous execution status. *Set4* is a fault simulation, by forcing K to have error values (K=3 instead of 1 when the vehicle is started). FSA moves to a safe mode, without executing test cases during invalid instances of the vehicle state.

Simulink design is visually based and hence troubleshooting is simple. Within the scheduling problem, the test engineer can define different strategies. For the state automaton in Figure 24, TC2 and TC3 were executed only once using test status flags that are set to '1' after test cases are executed. At the same time, TC1 was still allowed to repeatedly execute whenever key position K, was 1.

However, Simulink based FSA design has the following shortcomings:

- State chart becomes hard to manage for large numbers of test cases.
- Entry conditions become complex as number of pre-requisites increase.
- Simulink resolves transition conflicts. It is desirable that the designer has control over solving the conflict.

5.1.2 Scheduler design using digital logic

In order to attend shortcomings of an FSA design, an alternate design is proposed using a simple logic design. It's principle of operation is very similar to the FSA in the previous section. Implementation was done in python. Design focus is on having control over transition conflict resolution. The scheduler should not be difficult to implement even with large number of test cases and number of variables in pre-requisites. Logic design of the test scheduler is given in Figure 25.

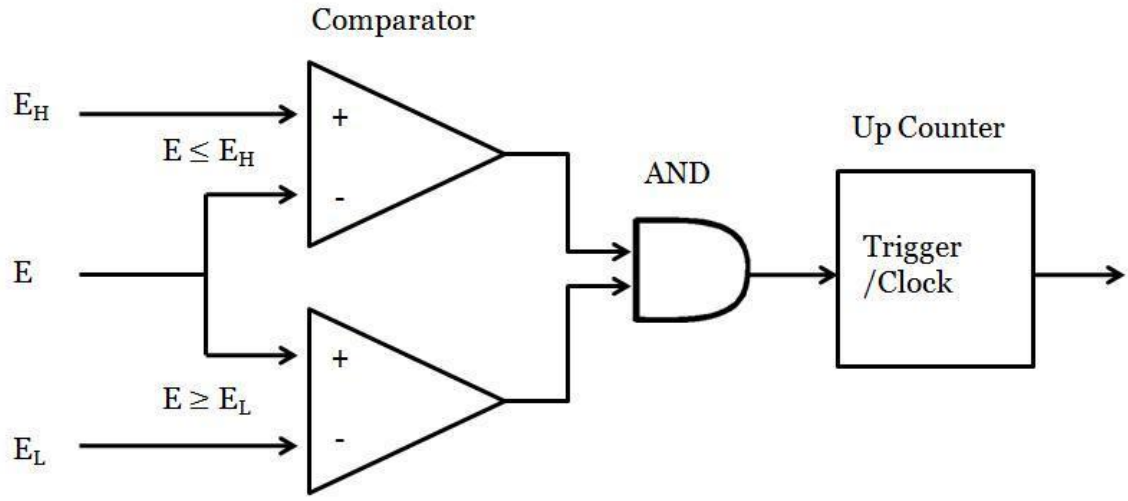


Figure 25: Dynamic scheduler logic design

For each sample E of vehicle state E(t) the scheduler selects a test case best suited for execution. The first stage consists of a comparator that compares each variable in the set E to upper and lower limits of the corresponding variable in the test case pre-requisite. If $E_L \leq E \leq E_H$, the counter is triggered by the AND gate output C and is incremented by one. This is repeated for all variables in the

set E. With seven state variables, a perfectly matched test case will yield the value 7 at the output of the counter. The same instantaneous value of vehicle state E, is matched with all other test cases. This results in selection of one or more perfectly matched test cases. A transition conflict resolver selects one test case from the set for execution.

		TC1		TC2		TC3		TC4		TC5	
	$E(t)_{t=1}$	E1	C1	E2	C2	E3	C3	E4	C4	E5	C5
Key Position	3	1	0	2	0	3	1	3	1	3	1
		1		2		3		3		3	
Voltage	24	0	1	24	1	24	1	24	1	24	1
		24		24		24		24		24	
Engine Speed	1000	0	0	0	0	450	0	450	1	450	1
		0		0		750		x		x	
Vehicle speed	40	0	0	0	0	0	0	20	1	20	1
		0		0		0		x		x	
Amb. temperature	20	x	1	x	1	x	1	x	1	x	1
		x		x		x		x		x	
Road gradient	0	x	1	x	1	x	1	0	1	0	1
		x		x		x		0		0	
Road friction	0	0	1	0	1	0	1	x	1	x	1
		0		0		0		x		x	
Total matches			4		4		5		7		7

Figure 26: Test result table

Referring to Figure 26, it can be observed that the transition conflict resolver must resolve between two test cases, TC4 and TC5. We can randomly select TC4, or the selection can be based on FMEA level as explained in Section 4.1.1.3. If TC5 has an FMEA level of 9, and TC4 a level of 4, TC5 will be selected. Pseudo code for implementing scheduler design, without a conflict resolver in python is given in Figure 27.

```
FOR i = 1 ... number of test cases
        FOR j = 1 ... number of state variables
                IF (E(j) ≥ E(j)L(i)  AND  E(j) ≤ E(j)H(i))
                        counter(i) = counter(i) + 1
```

Figure 27: Dynamic scheduler pseudo code

5.2 Maneuver module

The next part in the solution framework is to design a virtual driver that can start, drive, and stop the virtual vehicle. While the maneuver drives the vehicle, the test scheduler samples the vehicle state and selects a suitable test case for execution. Two techniques are discussed below, where one is GUI based using dSPACE Model Desk software, and the other one based on python.

5.2.1 Maneuver design using dSPACE Model desk.

Model Desk is the GUI for intuitive parameterization and parameter set management for Automotive Simulation Models. Model Desk has a Road

generator and maneuver editor that let the user define the individual road and maneuvers. There is also traffic editor for simulating traffic scenarios. Once the environment is designed in model desk, it can be downloaded to online or offline simulations. A typical Model desk environment screenshot is given in Figure 28.

Maneuver design involves the following steps.

1) Design a road profile by specifying parameters in the GUI. Road parameters considered in this thesis are; length, width, gradient, friction coefficient and banking angle.

2) Design maneuver: In maneuver, the designer specifies dynamic behavior of the vehicle. To drive the vehicle, one needs to specify target velocity, Accelerator and Brake pedal positions and steering angle. In case of a manual gear box, clutch position and gear number also need to be given. The maneuver specifications are then linked to the road created above.

3) Download the maneuver to online simulations: Parameters defined in the road and maneuver are directly linked to the Simulink model. Before simulation is started, the user selects the road and maneuver from the dSPACE Control Desk instrument panel (Fig. 6 in Section 2.1.2). To enable simulation controlled by the user defined maneuver, simulation control is switched from manual to maneuver. The maneuver ON button is enabled to drive the vehicle as defined.

Maneuver design environment screenshot is in Figure 29.

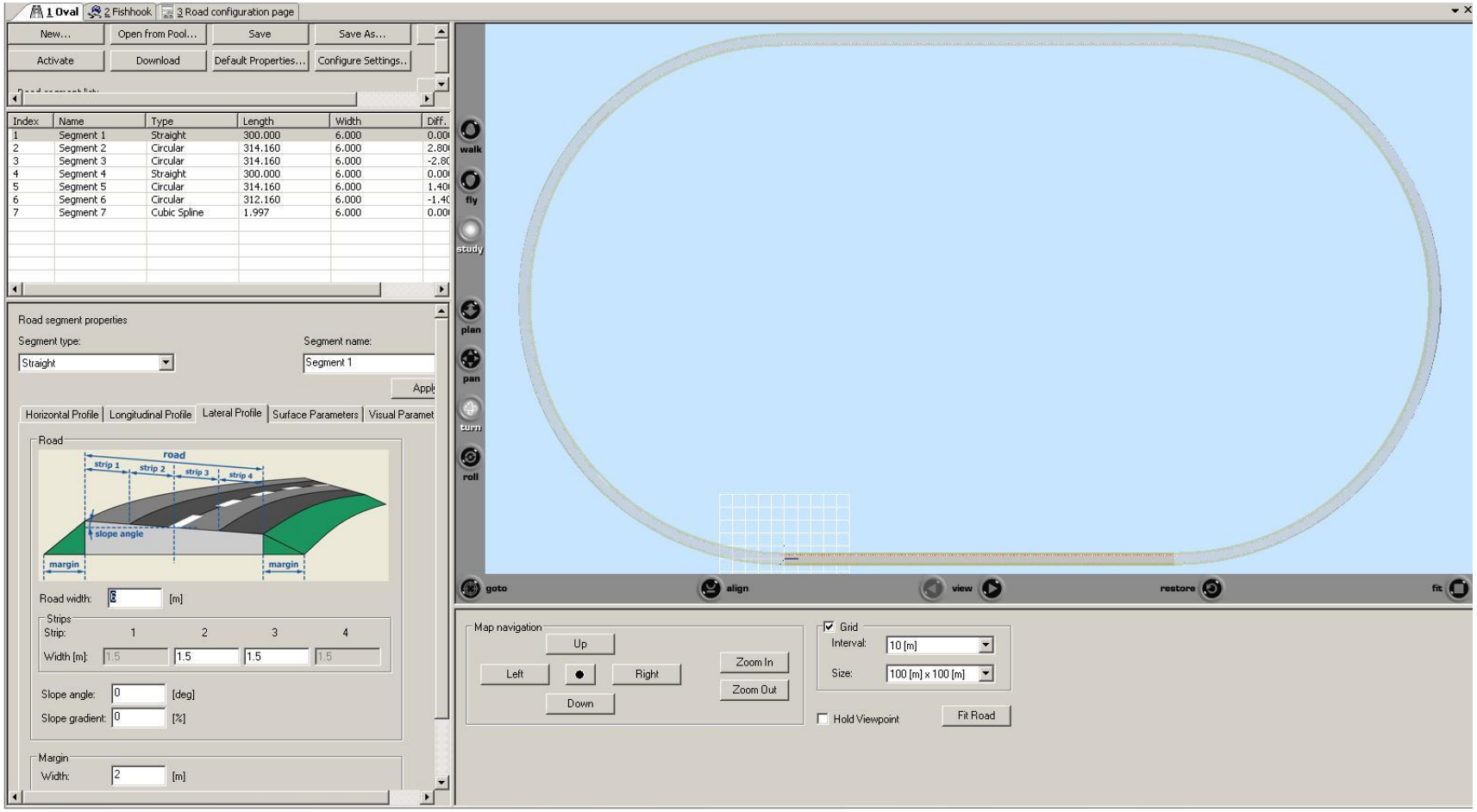


Figure 28: Model desk environment: Road design

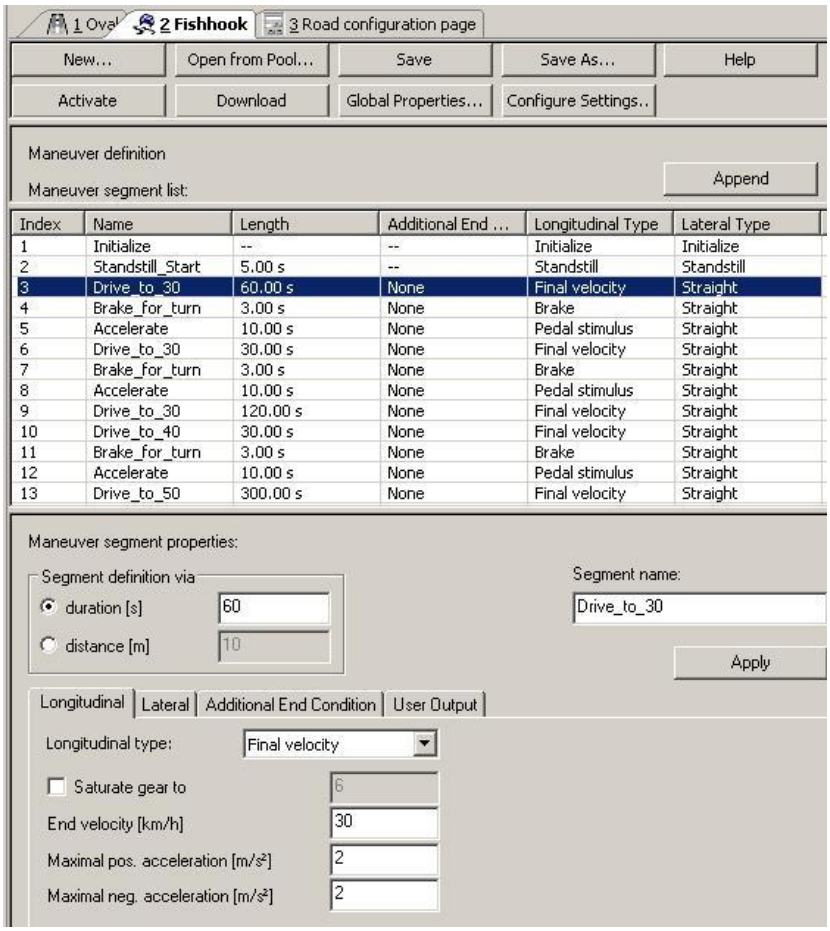


Figure 29: Model desk environment: Maneuver design

5.2.2 Maneuver design using Python

This design is non GUI based, but the implementation is similar to a GUI based design. We make use of Python methods that can control the Automotive Simulation Models used to develop a vehicle dynamic model in ILab2. These are the same methods used by test scripts to manipulate vehicle behavior. For this reason, simulation control is still kept at manual mode and need not be switched to maneuver mode. The vehicle can be maneuvered without the need of a road, although parameters like road gradient can be specified.

The steps involved in the design of a maneuver using Python are as described below.

Record state variables by running HIL simulator: In order to obtain realistic values of state variables, we run the HIL simulator either manually or by a maneuver defined in Model desk. State variable values during simulation are available as variables in the ASM models or in the vehicle electrical system CAN bus. These numerical values can be directly read to a text file or a spreadsheet using Python methods. A python script containing these methods, including specifications about the number of samples and sample spacing, can be developed to automate the process. The process is illustrated with Figure 30.

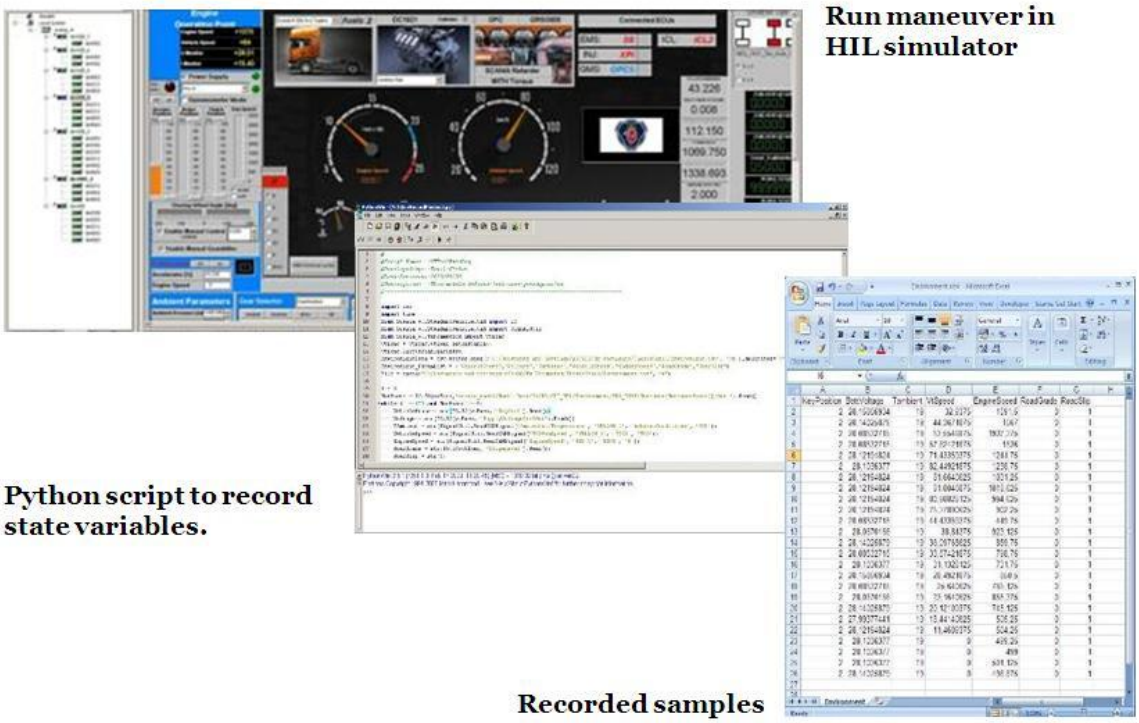


Figure 30: Maneuver recording using python script

Once recorded values (Appendix A1) are obtained, they are played back using the *maneuver* python module (Appendix A4). This maneuver module is script based and not the same as the dSPACE Control Desk maneuver. It fetches each state sample from the recorded data and uses python methods to the drive virtual vehicle to that state. Parking brake, Gear selection, Clutch/accelerator/brake positions are also handled within the methods. The following code starts the vehicle, drives it to 60 km/h, and stops the vehicle.

```
from Scania_VT.Driver import BasicDriver
vtDriver = BasicDriver.CreateBasicDriver(vtExec)
vtDriver.IgnitionOn()
vtDriver.EngineOn()
vtDriver.DriveAtSpeed(60)
vtDriver.Stop()
vtDriver.TurnOff()
```

Figure 31: Maneuvering using python

5.2.3 Comparison of Model desk and Python based maneuvers

Maneuver design using Model desk had the biggest advantage of flexibility and realistic implementation compared to script based design. The biggest disadvantage is the difficulty in running the test scripts while running the maneuver. When a Simulink model is in maneuver mode, using a Model desk maneuver, python based methods cannot be used, since they have to be run in manual mode. This prevents test cases from being executed since they use python methods to manipulate vehicle behavior. One solution is to modify the Simulink

model with a switch to move the simulation between maneuver and manual mode. The switch was controlled from the test framework using a python module. Attempts to implement the switch yielded no consistent and promising results. It was found out that switching control from maneuver to a test script produces undesirable vehicle performance. Further investigation is left as a future work since it was beyond the scope of this thesis.

Python based maneuver has less flexibility and less realistic implementation compared to Model desk maneuver. However, it has an edge over Model desk in the sense that maneuver and test scripts can be run side by side without the need of a switch as described above. The controllability is better since all parameters can be controlled at a lower level. It was decided to use a Python based maneuver, at least until the issue of simultaneously running maneuver and test scripts are resolved.

Maneuver design should be aimed at covering as many tests as possible while keeping maneuver duration short. Knowledge of test case functionality, and duration of test cases, are useful for an efficient maneuver design.

Chapter 6

The previous Chapter formally developed theory behind the problem. It was broken down into a scheduling part and a virtual maneuvering part. Several solutions were proposed based on literature surveys and existing practices at I-Lab2. This chapter discusses the implementation of the solution framework.

6 IMPLEMENTATION

Dynamic scheduling is implemented using two modules, namely virtual maneuvering and test scheduling. Tasks entrusted with these modules are:

- Maneuver module: Virtually drives the vehicle according to a pre determined test drive plan.
- Scheduler module: Samples the state of the vehicle at every instant of time during simulation to identify the best test case for execution.

The modified test process involves two steps: (1) Record a maneuver in real time. (2) Playback the maneuver to schedule tests in real time.

Steps involved in Recording a maneuver are:

1. Design vehicle maneuver/s using Model desk software.
2. Initialize simulation environment: Download the dynamic model of the vehicle into real time simulation hardware and configure the ECU's.
3. Start simulation.
4. Play one recorded maneuver (Option: manually drive the vehicle in the simulation. In this case, step 1 is not necessary).
5. Start to record the maneuver by sampling state variables at the rate of 5 minutes while the simulation is running. Average test execution time is 3 minutes. Therefore this sample rate allows one test case to be finished before the next state is sampled in the maneuver. Stop recording when maneuver is finished.
6. Go to step 4 until all designed maneuvers are played and recorded.
7. Stop simulation.

The python script can be used to automate steps 3 to 5.

Playing back the maneuver and scheduling tests are done together. The steps are:

1. Initialize the simulation environment: Download a dynamic model of the vehicle into real time simulation hardware and configure the ECU's.
2. Read list of all possible test cases valid for execution under the given vehicle configuration.
3. Start simulation.
4. Drive the vehicle to one state specified by the recorded maneuver.
5. Sample vehicle state variables before scheduling a test case.
6. Read test pre-requisite variables (variable names in pre-requisites are the same as state variables) of one test case from the list in Step 2.
7. Get one variable value from vehicle state in Step 5.
8. Read value of corresponding variable from test prerequisite variable set in step 6.
9. If the variable in Step 7 is within tolerance limits of the variable in Step 8, a match is found.
10. Go to step 7, until all variables in vehicle state are compared with the corresponding test prerequisite variable. If all pre requisite variables are matched with state variables, that test case is flagged.
11. Go to step 6, until all test cases in the test case list are considered.
12. If no test cases could be flagged (ie, no test cases are suitable for execution), go to step 18.
13. If only one test case is flagged after step 11, go to step 17.

14. Start transition conflict resolver.
15. Read all flagged test cases.
16. Sort test cases based on FMEA. A test case with the highest FMEA comes first. (FMEA table is used for sorting). Select the test case that is first in the list.
17. Execute the given test case.
18. Go to step 5. If no test cases could be selected for 10 consecutive samples of state variables, go to Step 4. Execute all subsequent steps until the maneuver has come to an end or all test cases in test case list were executed. (whichever comes first)
19. Stop simulation and reset the test environment.
20. Generate test report.

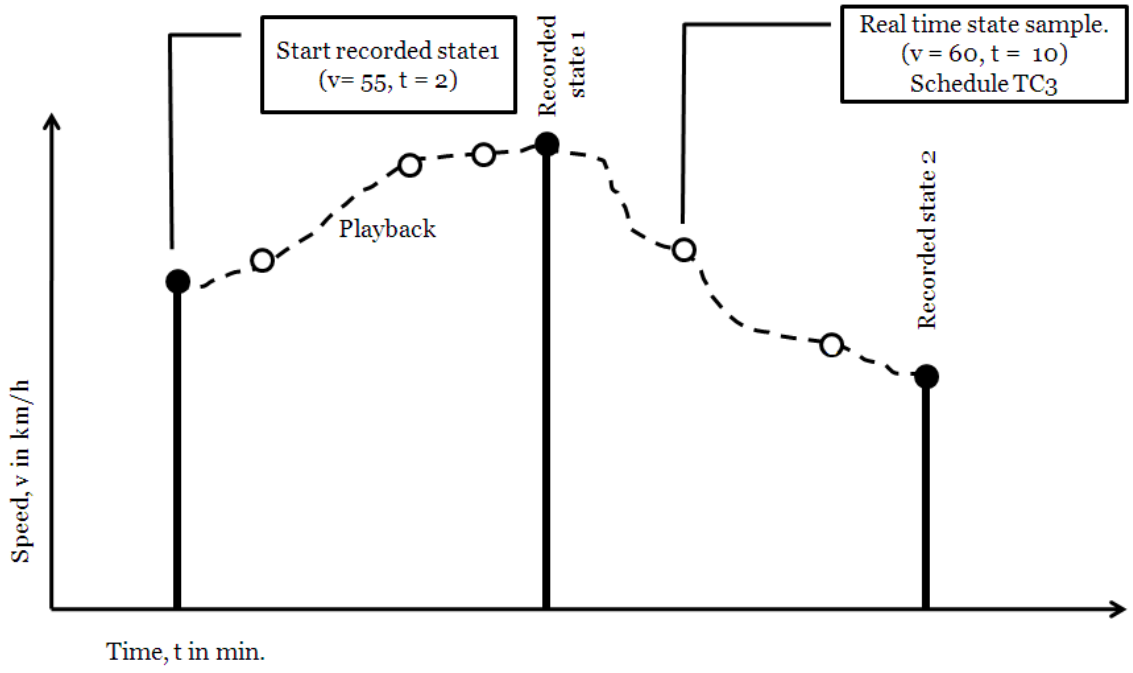


Figure 32: Real time test scheduling

Existing test process flow is given below for comparison with modified process.

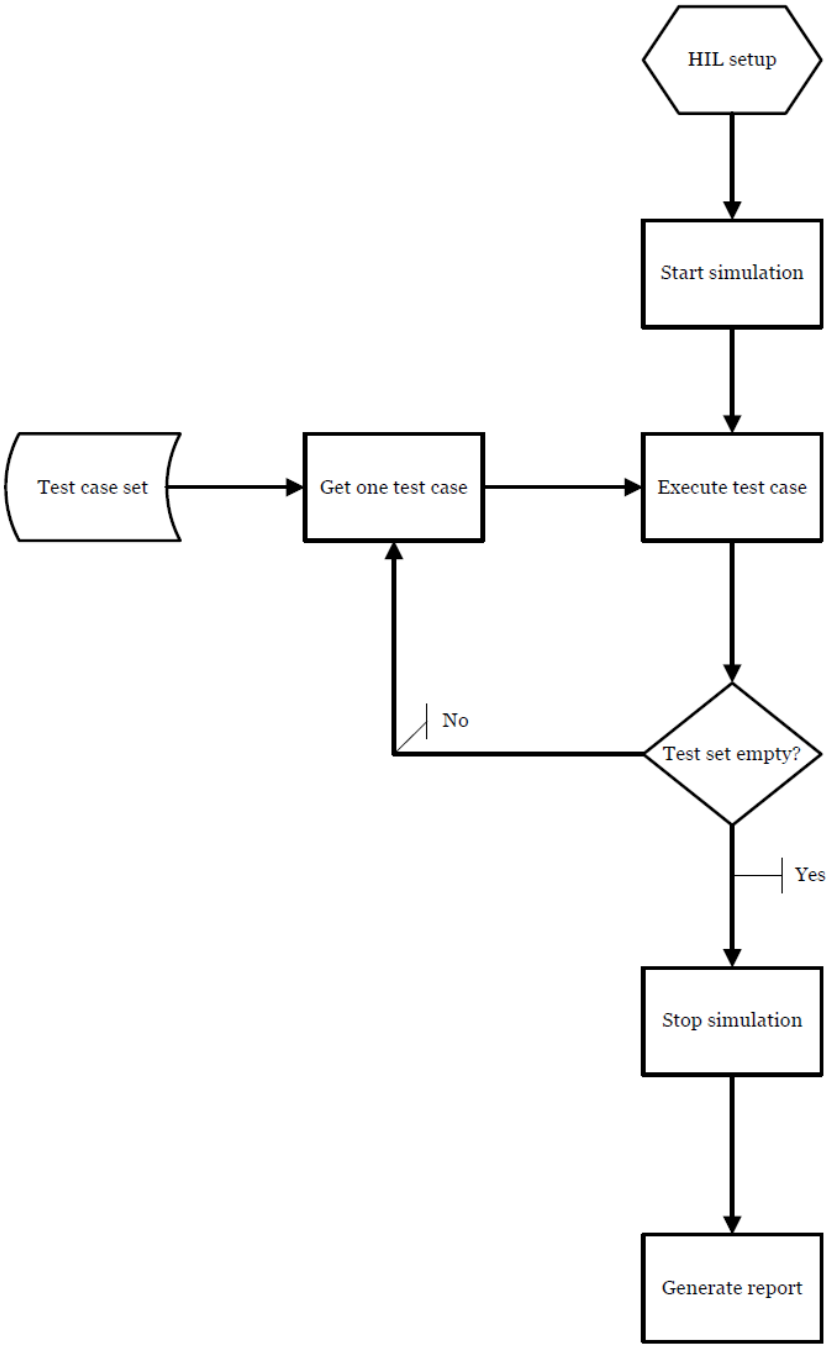
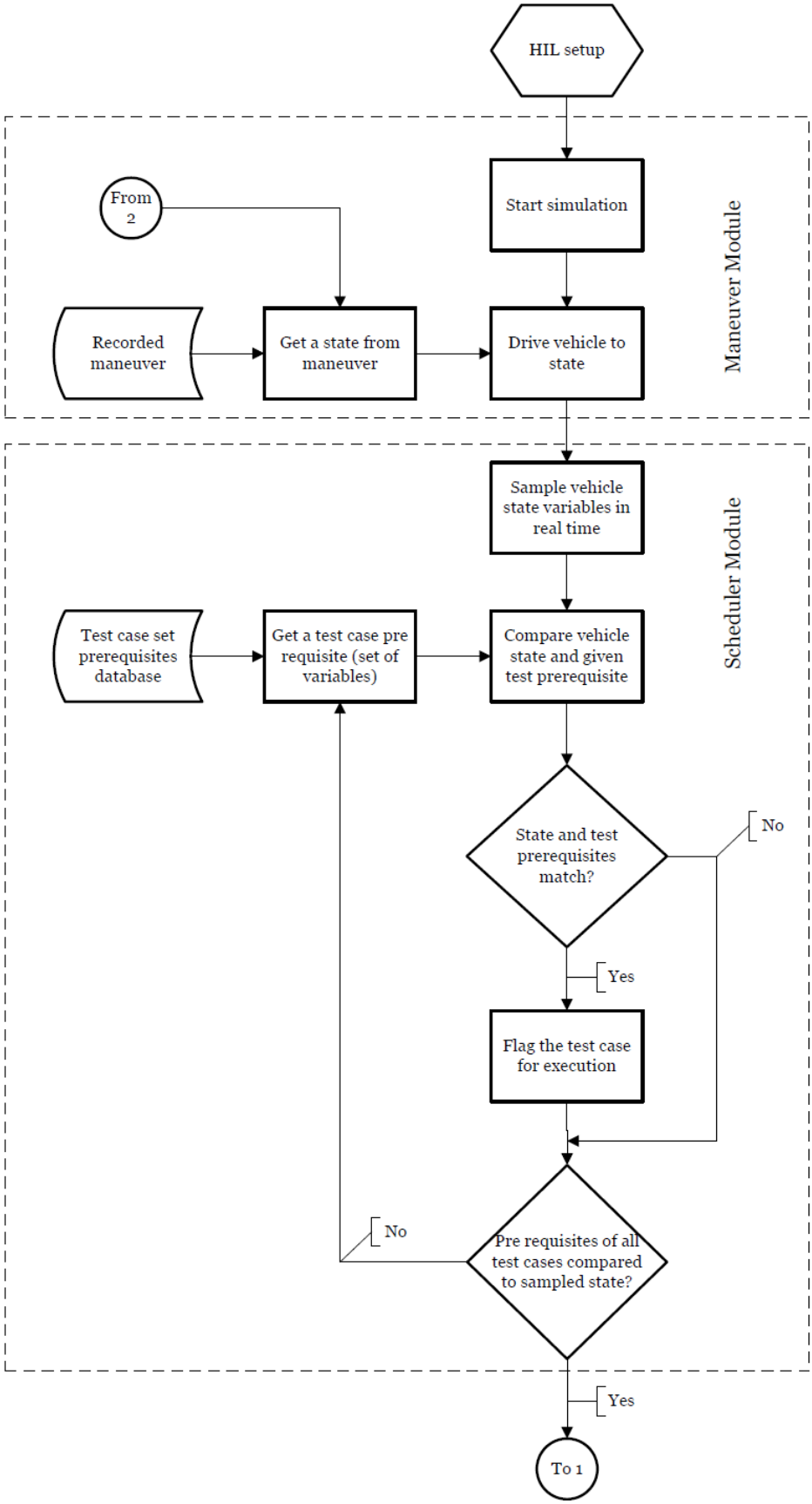


Figure 33: Existing test process at I-Lab2


HIL setup
Start simulation
From 2
Recorded maneuver
Get a state from maneuver
Drive vehicle to state
Maneuver Module
Sample vehicle state variables in real time
Test case set prerequisites database
Get a test case pre requisite (set of variables)
Compare vehicle state and given test prerequisite
Scheduler Module
State and test prerequisites match?
No
Yes
Flag the test case for execution
Pre requisites of all test cases compared to sampled state?
No
Yes
To 1

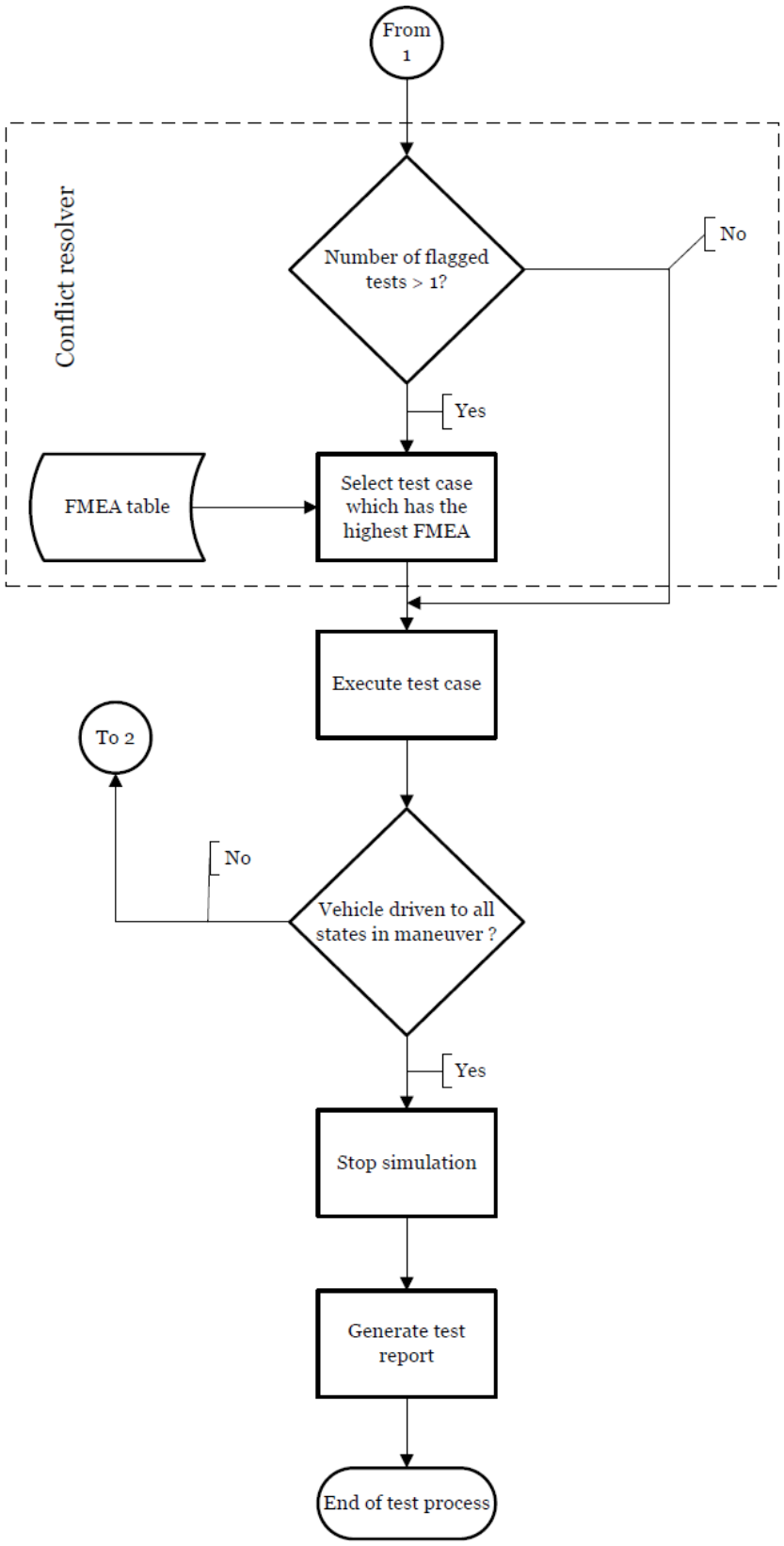


Figure 34: Modified test process at I-Lab2

Chapter 7

Chapter four discussed how the existing test process at I-Lab2 was modified to take it closer to a real test drive scenario. Although various solution presented in Chapter 4 is suitable for the new test philosophy, the one most suitable for implementation at I-Lab2 was selected. This chapter analyzes actual test reports and discusses what might be possible in the future for further improvement.

7 RESULTS

Dynamic scheduler was evaluated with the following test cases. This test set is only a small subset from the total number of tests cases used at Ilab2.

1. Activation test – ignition on (H4).
2. Adjust Cruise control set speed.
3. Display time using tachograph.
4. Enable Cruise Control.
5. Engage Cruise control, acceleration/retardation/resume. Deactivate cruise control using off button and brake.
6. Engine oil-pressure below the limit.
7. Engine speed control by accelerator pedal, Varying the acceleration pedal position.
8. Fuel level display, Display information (Truck).
9. Instrument panel lighting, Activation.
10. Main beam flash activation, Activation – ignition on.
11. Parking brake indication, Indicate that parking brake is active in instrument cluster.

To to enable the scheduler module to select test cases, a maneuver was designed using dSPACE Model Desk and played in HIL simulator, and real time values were recorded. Alternately, for a simple maneuver one could manually create the recorded maneuver, though this method offers less flexibility. One finding of this thesis is the potential of dynamic scripting techniques being used

in the test environment taking the simulation one step closer. The concept is only discussed as it is beyond the scope of this thesis.

7.1 Test reports and discussion

Tests were conducted using the following maneuver. An actual test framework generated report is also given.

VehicleState	Voltage	TAmbient	VehicleSpeed	EngineSpeed	RoadGrade	RoadSlip
0.0	23.69	19.0	0.0	0.0	0.0	1
1.0	23.69	19.0	0.0	0.0	0.0	1
2.0	28.01	19.0	0.0	520.625	0.0	1
2.0	27.92	19.0	0.0	520.25	0.0	1
2.0	27.93	19.0	0.0	519.75	0.0	1
2.0	28.03	19.0	6.92	1018.375	0.0	1
2.0	27.86	19.0	16.94	1003.75	0.0	1
2.0	27.84	19.0	24.62	1518.75	0.0	1
2.0	27.82	19.0	30.27	1149.375	0.0	1
2.0	27.86	19.0	26.67	1013.125	0.0	1
2.0	28.04	19.0	30.00	1141.5	0.0	1
2.0	27.99	19.0	26.96	1004.625	0.0	1
2.0	27.88	19.0	30.33	1152.25	0.0	1
2.0	27.82	19.0	30.56	1157.375	0.0	1
2.0	27.93	19.0	40.42	1230.375	0.0	1
2.0	27.86	19.0	35.85	1087.625	0.0	1
2.0	27.73	19.0	50.36	1214.625	0.0	1
2.0	27.84	19.0	60.51	1175.25	0.0	1
2.0	27.95	19.0	66.71	1330.875	0.0	1
2.0	27.93	19.0	74.12	1169.5	0.0	1
2.0	27.97	19.0	80.96	1257.875	0.0	1
2.0	27.84	19.0	74.88	1026.25	0.0	1
2.0	27.84	19.0	54.63	1269.375	0.0	1
2.0	27.93	19.0	10.60	520.625	0.0	1
2.0	27.99	19.0	5.92	520.25	0.0	1
2.0	27.95	19.0	1.82	519.5	0.0	1
2.0	27.95	19.0	0.75	520.625	0.0	1
2.0	28.01	19.0	0.0	519.875	0.0	1
2.0	28.04	19.0	0.0	520.25	0.0	1
2.0	27.88	19.0	0.0	519.5	0.0	1
1.0	23.67	19.0	0.0	0.0	0.0	1
0.0	23.69	19.0	0.0	0.0	0.0	1

Figure 35: Maneuver used to test dynamic scheduler

Test run (1) was performed without the dynamic test scheduler to serve as a baseline for other test runs. We observe that the test framework executed test cases according to the predefined test list, and not according to the maneuver.

TestSet: 206_1305_CS2ilab2ready

Passed TC's: 5 (5 unique)
Failed TC's: 1 (1 unique)
Aborted TC's: 3 (3 unique)
Total: 9 (9 unique)

1	TestCase:	UFT099_21 : Engage cruise control, acceleration/retardation/resume. - Deactivate cruise control using off button and brake.	Aborted	2010-12-13T11:59:28.012
2	TestCase:	UFT018_01 : Fuel level display, Display information (Truck)	Failed	2010-12-13T12:05:04.365
3	TestCase:	UFT057_01 : Activation – ignition on (H4)	Passed	2010-12-13T12:14:56.996
4	TestCase:	UFT099_22 : Adjust Cruise Control set speed	Aborted	2010-12-13T12:16:48.559
5	TestCase:	UFT110_01 : Engine oil-pressure below the limit	Passed	2010-12-13T12:22:21.444
6	TestCase:	UFT119_01 : Engine speed control by accelerator pedal, Varying the acceleration pedal position	Passed	2010-12-13T12:28:09.417
7	TestCase:	UFT099_20 : Enable cruise control	Aborted	2010-12-13T12:30:30.315
8	TestCase:	UFT060_01 : Main beam flash activation, Activation – ignition on	Passed	2010-12-13T12:35:52.723
9	TestCase:	UFT123_01 : Parking brake indication, Indicate that parking brake is active in instrument cluster	Passed	2010-12-13T12:37:47.187

Figure 36: Dynamic scheduler test run (1)

Test run (2) was performed without any conflict resolution module: that is, the first test case whose prerequisites are matched with the given sample of state variables is executed. Test cases are still scattered (as in cruise control) showing the necessity for using a conflict resolution module.

TestSet: 176Truck_13110_SOP1011P4ilab2ready

Passed TC's: 8 (8 unique)
Failed TC's: 0 (0 unique)
Aborted TC's: 3 (3 unique)
Total: 11 (11 unique)

1	TestCase:	UFT123_01 : Parking brake indication, Indicate that parking brake is active in instrument cluster	Passed	2010-11-03T13:56:26.273
2	TestCase:	UFT060_01 : Main beam flash activation, Activation – ignition on	Passed	2010-11-03T14:02:57.769
3	TestCase:	UFT110_01 : Engine oil-pressure below the limit	Passed	2010-11-03T14:06:25.894
4	TestCase:	UFT119_01 : Engine speed control by accelerator pedal, Varying the acceleration pedal position	Passed	2010-11-03T14:12:48.723
5	TestCase:	UFT473_02 : Engage Cruise control, acceleration/retradation/resume.Deactivation cruise control using off button and brake.	Aborted	2010-11-03T14:16:00.023
6	TestCase:	UFT018_01 : Fuel level display, Display information (Truck)	Passed	2010-11-03T14:16:13.085
7	TestCase:	UFT026_01 : Display time with tachograph	Passed	2010-11-03T14:27:07.148
8	TestCase:	UFT473_03 : Adjust Cruise control set speed	Aborted	2010-11-03T14:34:15.967
9	TestCase:	UFT473_01 : Enable Cruise Control	Aborted	2010-11-03T14:34:31.279
10	TestCase:	UFT013_01 : Instrument panel lighting, Activation	Passed	2010-11-03T14:34:36.450
11	TestCase:	UFT057_01 : Activation – ignition on (H4)	Passed	2010-11-03T14:38:09.562

Figure 37: Dynamic scheduler test run (2)

Test run (2) shows that we need to tune the scheduler for clustering similar test cases. This was achieved by adding a new pre-requisite variable that forces

execution of one test cases, based on the test case executed previously. This was discussed in Chapter 4. Test run (3) using a random selection conflict algorithm is given below.

TestSet: 193Truck_1215_A4_HPIilab2ready

Passed TC's: 6 (6 unique)
Failed TC's: 0 (0 unique)
Aborted TC's: 3 (3 unique)
Total: 9 (9 unique)

1	TestCase:	UFT123_01 : Parking brake indication, Indicate that parking brake is active in instrument cluster	Passed	2010-11-04T09:39:00.928
2	TestCase:	UFT060_01 : Main beam flash activation, Activation – ignition on	Passed	2010-11-04T09:41:13.398
3	TestCase:	UFT110_01 : Engine oil-pressure below the limit	Passed	2010-11-04T09:43:12.585
4	TestCase:	UFT119_01 : Engine speed control by accelerator pedal, Varying the acceleration pedal position	Passed	2010-11-04T09:49:11.085
5	TestCase:	UFT018_01 : Fuel level display, Display information (Truck)	Passed	2010-11-04T09:51:37.661
6	TestCase:	UFT057_01 : Activation – ignition on (H4)	Passed	2010-11-04T10:01:53.299
7	TestCase:	UFT099_20 : Enable cruise control	Aborted	2010-11-04T10:28:30.881
8	TestCase:	UFT099_22 : Adjust Cruise Control set speed	Aborted	2010-11-04T10:34:05.380
9	TestCase:	UFT099_21 : Engage cruise control, acceleration/retardation/resume. - Deactivate cruise control using off button and brake.	Aborted	2010-11-04T10:39:51.111

Figure 38: Dynamic scheduler test run (3)

We see considerable improvement in test scheduling. Cruise control tests are clustered, and the sequencing is according to the maneuver designed.

7.2 Modifies process Test efficiency measurements

This sections evaluates the new method in terms of test efficiency as explained in Section 3.3.1.

In Test run(2), we observe that test case UFT099_20 was aborted and UFT099_21 and UFT099_22 shared the same test result status. It was found that the virtual vehicle was not able to accelerate to the minimum speed required for engaging cruise control. During troubleshooting, test execution was interrupted, the virtual vehicle stopped, and started again. This removed the fault in acceleration. Further investigation of the fault is beyond the scope of this thesis.

In normal test execution, a fault of this kind might not propagate to the next test case. This is because the virtual vehicle is stopped after the previous test. But in a real test drive, faults propagate from one test case to other test cases. Hence, we observe that the new method reproduces fault propagation as in reality.

It was proposed that UFT099_22 and UFT099_21 can be prevented from being executed when UFT099_20 was aborted. This saves total test execution time. Test efficiency, which is an indicator of the time saving is calculated below.

Using efficiency measurement in Section 3.3.1, for Test run (2),

Before removing test cases UFT099_21 and UFT099_22:

$$\eta_{TEFF} = \left(1 - \frac{(NTCF+NTCA)}{NTCT}\right) * 100$$

NTCF = 0, NTCA = 3, NTCT = 9, η_{TEFF} = 66%

If tests UFT099_21 and UFT099_22 are removed:

NTCF = 0, NTCA = 1, NTCT = 7, η_{TEFF} = 85%

However for the purpose of failure analysis, it was decided to allow test failures due to fault propagation. A proposal to remove them will be considered in the future.

7.3 Challenges faced during implementation

Referring to Figure 16, in chapter 4, it was decided that the maneuver would drive the virtual vehicle to a pre defined state and the scheduler would select the most appropriate test case. The two modules, maneuver and scheduler, need access to the HIL simulator. They cannot access the simulator at the same time since the test framework is single threaded. That means, from a hardware point of view, only one process/module can be in charge of the simulator.

DSPACE Model desk based maneuver has higher flexibility and reality compared to Python based record and playback maneuver. However it takes full procession of HIL simulator until finished. It was not possible to run a scheduler in parallel to the maneuver. This was the reason why the maneuver and scheduler modules time share the HIL simulator. The maneuver would drive the vehicle to a state and then gives HIL simulator control to the scheduler. The scheduler would schedule a test case, test framework would execute it, and HIL simulator control would be given to the maneuver. It is recommended to explore a multithreaded test framework, or HIL simulator sharing techniques, other than time sharing.

Another challenge was using Python based speed controllers. In order to drive a vehicle to a certain speed, a Proportional-Integral-Derivative controller is used currently. At lower vehicle speeds this controller was not able to achieve

target speed. It was also observed that this controller takes longer to drive the vehicle to a new speed reference from one stable speed. It is recommended that the python speed controller be either optimized, or other speed control strategies be explored.

7.4 Proposals for future research activities

In dynamic test scheduler focus was on reusing existing test scripts, which influenced selection of the scheduler algorithm. It is proposed that one area of future research could be to explore a dynamic scripting technique. In dynamic scripting based testing there will be no pre-written test scripts. A scripter module directly reads test actions from design specifications and would translate these to python statements. The scripter module could be as simple as a look up table with Action statements linking to a corresponding python statement.

Another area worth exploring is the use of a dynamic scheduling algorithm in real test drive. At present, test drivers plan the test activity before using the list of test cases available to them. While driving, they make decisions on what test case has to be executed by observing their environment. This could be handled by a module that reads state variables directly from the vehicle CAN bus, and suggests to the driver which test case can be executed.

REFERENCES

1) Design and Implementation of HIL Simulators for Power train Control System Software Development - S. Raman, N. Sivashankar, W. Milam, W. Stuart, S. Nabi. Proceedings of the American Control Conference San Diego, California, June 1999.

2) Automation of Hardware-in-the-Loop and In-the-Vehicle Testing and Validation for Hybrid Electric Vehicles at Ford (2006-01-1448) - Sergey G. Semenov at 2006 SAE World Congress, Detroit, Michigan, USA.

3) Test set size minimization and Fault detection effectiveness: A case study in a Space application - W.E Wong, J.R. Horgan, A.P. Mathur, Alberto Pasquini. Journal of Systems and Software, Vol. 48, No. 2, October 1999, pp 79-89.

4) An introduction to Operating systems: concepts and practice, Second Edition, P. Bhatt, PHI Learning Pvt. Ltd., 2007.

5) Planning and Scheduling in Manufacturing and Services, Second Edition, Michael L. Pinedo, Springer, 2009 - Business & Economics.

6) Sequential testing algorithms for multiple fault diagnosis, M. Shakeri, V. Raghavan, K. R. Pattipati, and A. Patterson-Hine, IEEE Trans. Syst., Man, Cybern. A, Syst., Humans, vol. 30, no. 1, pp. 1–14, Jan. 2000.

7) Test Sequencing in Complex Manufacturing Systems, R. Boumen, I. S. M. de Jong, J. W. H. Vermunt, J. M. van de Mortel-Fronczak, and J. E. Rooda, IEEE Trans. Syst., Man, Cybern., Part A systems and humans, Vol. 38 No.1 , Jan. 2008.

8) Principles of Modern Digital Design, Parag K. Lala, Wiley- Inter science, 2007 - Computers.

9) Synthesis of finite state machines, Timothy Kam, Springer, 1997 - Mathematics.

10) Using Markov Chain Usage Models to Test Complex Systems, S. J. Prowell, Proceedings of the 38th Hawaii International Conference on System Sciences – 2005.

11) Markovian Analysis of Large Finite State Machines, Gary D. Hachtel, Fellow, IEEE, Enrico Macii, Member, IEEE, Abelardo Pardo, and Fabio Somenzi, Member, IEEE, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems Vol. 15, No. 12, Dec. 1996

12) Discrete Mathematics, Richard Johnsonbaugh, Seventh Edition, Prentice Hall. (January 8, 2008).

13) On sensor scheduling via Information Theoretic Criteria- Andrew Logothetis and Alf Isaksson at 1999 Proceedings of American Control Conference.

14) Introduction to Automata Theory, Languages and Computation: Hopcroft, Motwani, Ullman - Third edition, Pearson International. (2007)

15) Modeling software with Finite State Machines; A practical approach- Ferdinand Wagner, Schmuki, Thomas Wagner, Wolstenholme. (May 15, 2006)

16) The Development of a Real Time Hardware-in the Loop Test Bench for Hybrid Electric Vehicles Based on Multi-Thread Technology- Hu Zhong, Guoqiang Ao, Jiaxi Qiang, Lin Yang, Bin Zhuo. Institute of Automotive

Electronic Technology, School of Mechanical Engineering, Shanghai Jiao Tong University, Shanghai, 200030, China.

17) http://www.scania.com/

18) http://www.dspaceinc.com/en/inc/start.cfm

19) http://www.wikipedia.org/

20) Internship report at REST Ilab2- Volker Klink

21) Automotive Simulation Models guide – dSpace GmbH

22) Finite State Machines and Finite State Automata, Lecture notes in discrete mathematics, Prof. William Chen, Dept. of Mathematics, Macquarie University, Sydney, Australia.

23) Lecture notes, Introduction to Theoretical Computer Science, Prof. G. Grahne, Concordia University, Canada.

24) Interview with Mr. Tomas Backlund, REST, Scania AB, Sweden. (Oct 15, 2010).

APPENDIX

A1. Real time samples of vehicle state.

TimeHrs	TimeMin	TimeSec	VehicleState	Voltage	TAmbient	VehicleSpeed	EngineSpeed	RoadGrade	RoadSlip
10	23	55	0.0	23.69	19.0	0.0	0.0	0.0	1
10	24	30	1.0	26.58	19.0	0.0	0.0	0.0	1
10	25	15	2.0	28.01	19.0	0.0	520.625	0.0	1
10	26	1	2.0	28.03	19.0	6.92	1018.375	0.0	1
10	26	6	2.0	27.84	19.0	24.62	1518.75	0.0	1
10	26	18	2.0	27.82	19.0	30.27	1149.375	0.0	1
10	27	5	2.0	27.95	19.0	26.72	1012.625	0.0	1
10	27	21	2.0	28.04	19.0	30.00	1141.5	0.0	1
10	27	45	2.0	27.99	19.0	26.96	1004.625	0.0	1
10	28	6	2.0	27.88	19.0	30.33	1152.25	0.0	1
10	30	6	2.0	27.93	19.0	40.42	1230.375	0.0	1
10	30	41	2.0	27.90	19.0	47.98	1479.75	0.0	1
10	30	51	2.0	27.73	19.0	50.36	1214.625	0.0	1
10	35	47	2.0	27.84	19.0	60.51	1175.25	0.0	1
10	40	40	2.0	27.95	19.0	66.71	1330.875	0.0	1
10	40	45	2.0	27.93	19.0	74.12	1169.5	0.0	1
10	40	49	2.0	27.97	19.0	80.96	1257.875	0.0	1
10	42	22	2.0	27.84	19.0	74.88	1026.25	0.0	1
10	42	24	2.0	27.84	19.0	54.63	1269.375	0.0	1
10	42	26	2.0	27.88	19.0	13.29	552.625	0.0	1
10	42	31	2.0	27.93	19.0	8.12	519.5	0.0	1
10	42	43	2.0	27.95	19.0	1.82	519.5	0.0	1
10	42	55	2.0	27.95	19.0	0.04	519.75	0.0	1
10	42	57	2.0	28.01	19.0	0.0	519.875	0.0	1

A2. Test case pre-requisites.

Test Case	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10
K_{TL}	2	1	2	2	2	2	2	2	1	1
K_{TH}	2	1	2	2	2	2	2	2	1	1
V_{TL}	20	20	20	20	20	20	20	20	20	20
V_{TH}	30	30	30	30	30	30	30	30	30	30
θ_{TL}	x	x	0	x	x	x	x	x	x	x
θ_{TH}	x	x	x	x	x	x	x	x	x	x
v_{TL}	x	0	x	65	70	x	x	0	x	x
v_{TH}	x	0	x	x	x	x	x	0	x	x
Ω_{TL}	x	x	450	450	450	450	450	x	x	x
Ω_{TH}	x	x	x	x	x	x	1000	x	x	x
α_{TL}	0	0	0	0	0	0	0	0	0	0
α_{TH}	0	0	0	0	0	0	0	0	0	0
ζ_{TL}	1	1	1	1	1	1	1	1	1	1
ζ_{TH}	1	1	1	1	1	1	1	1	1	1

A3. Example of a test script. User function test: Enable cruise control.

```
# -*- coding: cp1252 -*-
# File name: UFT99_XX.py
"""
Testcase UFT 99.20:       Enable cruise control
"""

import time
from Scania_VT.StandardVehicle.CAN import SignalUtil
class UFT099_20:
    def __init__(self, vtExec):
        self._name = "UFT099_20 (MSC 1169): Enable and engage conventional
cruise control. Enables CC with CC switch."
        self._version = 1
        self._vtExec = vtExec
        self._vtExec.VtPrint.DebugPrint( "Initialising %s" %(self._name))
        # create battery
        from Scania_VT.StandardVehicle.Vehicle.ElectricSystem import Battery
        self._battery = Battery.Battery(self._vtExec)
        # create driver
        from Scania_VT.Driver import BasicDriver
        self._driver = BasicDriver.CreateBasicDriver(self._vtExec)
        # create engine (incl. exhaust brake)
        from Scania_VT.StandardVehicle.Vehicle.Drivetrain import Engine
        self._engine = Engine.Engine(self._vtExec)
        # create cruise control
        from Scania_VT.StandardVehicle.Vehicle.Cockpit import CruiseControl
        self._cc = CruiseControl.CruiseControl(self._vtExec)
        # create Dynamics
        from Scania_VT.StandardVehicle.VehicleState import Dynamics
        self._dynamics = Dynamics.Dynamics(self._vtExec)
```

```
        # create accelerator pedal
        from Scania_VT.StandardVehicle.Vehicle.Drivetrain import AccPdl
        self._accelerator = AccPdl.AccPdl(self._vtExec)
        # create Gearbox
        from Scania_VT.StandardVehicle.Vehicle.Drivetrain import Gearbox
        self._gearbox = Gearbox.CreateGearbox(self._vtExec)
        # create timer
        from Scania_VT.TaFramework.VtUtils import VtTimer
        self._timer = VtTimer.VtTimer(self._vtExec)
#---------------------------------------------------------------------
    def __del__(self):
        # Delete (unreference) objects
        self._vtExec = None
        self._battery = None
        self._driver = None
        self._engine = None
        self._cc = None
        self._dynamics = None
        self._accelerator = None
        self._gearbox = None
        self._timer = None
#---------------------------------------------------------------------
#Code modified using self.__vtExec.VtTCTestExecStatus to prevent start and stop
of the vehile
    def Execute(self, parameters = {}):
        try:
            self.__vtExec.VtTCTestExecStatus = 1
            self.Prerequisites()
            self.__vtExec.VtTCTestExecStatus = 0
            self.Act1()
        finally:
```

```
            self.__vtExec.VtTCTestExecStatus = 3
            self.Postrequisites()
            self.__vtExec.VtTCTestExecStatus = 0
#-----------------------------------------------------------------------
    def Prerequisites(self):
        self._act_name = "Prerequisites"
        self._vtExec.VtTracker.StartAction("Executing   %s"   %(self._act_name),
self._act_name)
        # Setting the battery voltage: 24 V
        self._vtExec.VtPrint.DebugPrint("Setting the battery voltage")
        self._battery.SetVoltage(24)
        # Start vehicle
        self._vtExec.VtPrint.DebugPrint("Start vehicle")
        self._driver.EngineOn()
        # Exhaust brake floor switch released
        self._vtExec.VtPrint.DebugPrint("Exhaust brake floor switch released")
        self._engine.ReleaseExhaustBrakeFloorSwitch()
        # Make sure that neither Acc (+) nor Ret (-) is pressed.
        self._vtExec.VtPrint.DebugPrint("Make sure that neither Acc (+) nor Ret
(-) is pressed")
        self._cc.ReleaseAccRetSwitch()
        # Cruise control enable switch off
        self._vtExec.VtPrint.DebugPrint("Cruise control enable switch off")
        self._cc.DisableCc()
        # Drive at 70 km/h
        self._vtExec.VtPrint.DebugPrint("Vehicle speed approaches 70 km/h")
        self._driver.DriveAtSpeed(70)
        #self._vtExec.VtXMLReport.Pass("Prerequisities  OK.  Vehicle  driving  at
70 km/h", self._act_name)
        self._vtExec.VtTracker.EndAction(True)
#-----------------------------------------------------------------------
    def Postrequisites(self):
```

```
        self._act_name = "Postrequisites"
        self._vtExec.VtTracker.StartAction("Executing   %s"   %(self._act_name),
self._act_name)
        # Disable CC
        self._vtExec.VtPrint.DebugPrint("Cruise control enable switch off")
        self._cc.DisableCc()
        # Stop vehicle
        self._vtExec.VtPrint.DebugPrint("Stop vehicle")
        self._driver.Stop()
        # Turn off vehicle
        self._vtExec.VtPrint.DebugPrint("Turn off vehicle")
        self._driver.TurnOff()
        self._vtExec.VtTracker.EndAction(True)
#-------------------------------------------------------------------
    def Act1(self):
        self._act_name = "Act 1: Press CC Enable Switch"
        self._vtExec.VtTracker.StartTest(self._act_name)
        # STIMULUS: Press CC enable switch
        self._vtExec.VtPrint.DebugPrint("Pressing    enable    Cruise    Control
switch")
        self._cc.EnableCc()
        self._timer.Sleep(1)
        # EXPECTED RESPONSE
        CCMUpdateCC    =    SignalUtil.ReadCANSignal('CCMUpdateCC',    'YELLOW_1',
'CCMD', 'CCM')
        self._vtExec.VtPrint.DebugPrint("CCMD_CCM.CCMUpdateCC       =       %s"
%(CCMUpdateCC))
        self._vtExec.VtXMLReport.AddSingleLimitComparison(CCMUpdateCC, 7, '==',
self._act_name, 'CCMUpdateCC should be 0x7 (NoCCUpdate)', 'Pressing CC Enable
Switch')
        CCM_CCActive   =   SignalUtil.ReadCANSignal('CCM_CCActive',   'YELLOW_1',
'CCMD', 'CCM')
```

```
        self._vtExec.VtPrint.DebugPrint("CCMD_CCM.CCM_CCActive       =       %s"
%(CCM_CCActive))
        self._vtExec.VtXMLReport.AddSingleLimitComparison(CCM_CCActive,       0,
'==',  self._act_name,  'CCM_CCActive  should  0x0  (NotActive)',  'Pressing  CC
Enable Switch')
        self._vtExec.VtTracker.EndTest()
#--------------------------------------------------------------------
if __name__ == '__main__':
       import sys
       from TestExecution import Start_ScaniaVT
       tcModule = 'UFT099_20'
       tcClass = 'UFT099_20'
       #Programming these ECUs:
       ecusToProgram = ''
       sopsPath                                                                =
r'C:\ta\ILab2\ScaniaTools\Main\SOPS\135Truck_13110_OPC5_CP_SC1_S8_3-
ped_EEC3.xml'
       fiuPath = None
       tc  =  Start_ScaniaVT.InitSingleTestScript(tcModule,  tcClass,  sys.argv,
ecusToProgram, sopsPath, fiuPath)
       vtExec = Start_ScaniaVT.vtExec
       try:
              vtExec.VtTracker.StartTestCase(tcClass)
              tc.Execute()
       finally:
              vtExec.VtTracker.EndTestCase()
              vtExec.VtTracker.EndTestSuite()
```

A4. Test scheduler implemented in python

```
"""
Script Name:   VTTestScheduler
Developed by:  Tenil Cletus
Last Revision: 2010/10/13
Description:   This module has two functions.
               1.Select a set of test cases best suited for that instance of
vehicle maneuver
               2.Based on a conflict resolve algorithm, select only one test
case from step 1.
"""


class VtTCScheduler:
    def __init__(self, vtExec):
        """
        Construct of scheduler instance. Initialize variables and download test
execution conditions
        """
        #Dictonary struct for keeping the tc states...
        #Download test execution requisites
        self._vtExec = vtExec
        from Scania_VT.TaFramework.VtUtils import VtTCExecReq
        self._NumOfTests = 15
        self._TestsExecuted = [0]*self._NumOfTests
        self._TestSelected = ''
        self._TestRank = {}
        self._tcStates = {}
        self._Environment = {}
        tcStates = VtTCExecReq.VtTCExecReq()
        self._tcStates = tcStates.TestCaseReq()
```

```
        self._ParamList = "VehicleState", "Voltage", "TAmbient",
"VehicleSpeed", "EngineSpeed", "RoadGrade", "RoadSlip")
    def GetTc(self,tcList):
        """
        Function for selecting a test case suitable for execution for an
instance of the environment.
        """
        #Obtain one sample of maneuver
        from Scania_VT.StandardVehicle.CAN import IO
        from Scania_VT.StandardVehicle.CAN import SignalUtil
        self._Environment['VehicleState']=IO.IO(self._vtExec, 'KeyCtrl').Read()
        self._Environment['Voltage']=IO.IO(self._vtExec,
'SupplyVoltageCtrlOut').Read()
        self._Environment['TAmbient'] = SignalUtil.ReadCANSignal(
'AmbientAirTemperature', 'YELLOW_1', 'AmbientConditions', 'ICL')
        self._Environment['VehicleSpeed'] = SignalUtil.ReadCANSignal(
'TCOVehSpeed', 'YELLOW_1', 'TCO1', 'TCO')
        self._Environment['EngineSpeed'] = SignalUtil.ReadCANSignal(
'EngineSpeed', 'RED_1', 'EEC1', 'E')
        self._Environment['RoadGrade'] = IO.IO(self._vtExec,
'SlopeLevel').Read()
        self._Environment['RoadSlip'] = 1
        for TC in tcList:
            #Compare vehicle state
            i = 1
            for Param in self._ParamList:
                if (self._Environment[Param] >= int(
self._tcStates[TC][Param][0] )) and (self._Environment[Param] <= int(
self._tcStates[TC][Param][1])):
                    self._TestRank[TC] = i
                    if (self._TestRank[TC] == 7):
                        self._TestSelected = TC
```

```
                    i+=1
        return self._TestSelected
if __name__ == '__main__':
    from Scania_VT.TaFramework import VtExec
    import time
    vtExec=VtExec.VtExec.getInstance()
    vtExec.InitVariableAccess()
    ts=VtTCScheduler(vtExec)
    tcList = ["TC_57_1", "TC_13_1", "TC_18_1", "TC_60_1", "TC_87_4",
"TC_30_28","TC_473_1","TC_473_2","TC_473_3","TC_119_1","TC_110_1","TC_123_1","T
C_57_2","TC_60_2"]
    while len(tcList) != 0:
        tcSelected = ts.GetTc(tcList)
        print tcSelected
        if tcList.count(tcSelected) == 0:
            print 'No TC selected'
            print tcList
        else:
            tcList.remove(tcSelected)
        time.sleep(5)
```

A5. Maneuver implemented in python

```
"""
Script Name:   VtTCManeuver
Developed by:  Tenil Cletus
Last Revision: 2010/10/19
Description:   This module has two functions.
                1. Record the environment (Vehicle speed, Engine speed etc. by
playing a predefined maneuver (designed using Control desk) in HIL simulator.
Data is stored in a .csv file
                2. Playback data recorded in HIL simulator when test suite has
to be run with event driven test scheduler
"""


class VtTCManeuverUtils:
    def __init__(self, vtExec):
        """
        Construct of init instance. Initialize variables.
        """
        import csv
        self._vtExec = vtExec
        self._vtExec.VtTCTestExecStatus = 0
        #Import user defined maneuver.
        #import matlablib
        #ManeuverMatFile = matlablib.Matfile()
#ManeuverMatFile.Open('C:\ILab2.development\Parametrization.current\SCANIA_ILAB
_2\Pool\Environment\Maneuver\Fishhook.mat','r')
        #self._MatlabSegmentMatrix = ManeuverMatFile.GetArray('SegmentMatrix')
        #self._MatlabTableData = ManeuverMatFile.GetArray('TableData')
        self._MatlabSegmentMatrix = 0
        self._MatlabTableData = 0
        #Initialize environment dataset
```

```
        #self._Environment = csv.writer(open(r'C:/Documents and
Settings/p53408/My Documents/ThesisTenil/Environment.csv', 'rb'),delimiter=";")
        self._EnvParamList = ("SampleTime", "VehicleState", "Voltage",
"TAmbient", "VehicleSpeed","EngineSpeed","RoadGrade","RoadSlip")
        self._SampleTime = 1
        self._VehicleState = 0
        self._Voltage = 0
        self._TAmbient = 0
        self._VehicleSpeed = 0
        self._EngineSpeed = 0
        self._RoadGrade = 0
        self._RoadSlip = 0
        #Define maneuver control variables
        self._PowerSwitch = 'scania_rack2/Model
Root/RACK2/CPT_IO/IO_PAR/Simulator/UI_PAR_CTRL_KL30/Value'
        self._KeyPos = 'KeyCtrl'
        self._GearSelector = 'scania_rack1/Model
Root/RACK1/CPT_IO/IO_PAR/COO/UI_PAR_GearSeletctor/Value'
        self._ParkingBrake = 'scania_rack3/Model
Root/RACK3/CPT_MDL/MDL_PAR_ParkingBrake_IO/Value'
        self._ASMMode
='scania_rack3/ModelRoot/RACK3/CPT_MDL/Environment/MDL_PAR/ManeuverControl/Sw_M
anualDriving[1ManSched|2Manual]/Value'
        self._SelManeuver =
'scania_rack3/ModelRoot/RACK3/CPT_MDL/Environment/MDL_PAR/MANEUVER_SELECT_SWITC
H/Sw_ManeuverSelect[1_6]/Value'
        self._SelRoad = 'scania_rack3/Model
Root/RACK3/CPT_MDL/Environment/MDL_PAR/ROAD_SELECT_SWITCH/Sw_Road_Select[1_6]/V
alue'
        self._StartManeuver =
'scania_rack3/ModelRoot/RACK3/CPT_MDL/Environment/MDL_PAR/ManeuverControl/Maneu
verStart[0|1]/Value'
```

```
        self._StopManeuver = 'scania_rack3/Model
Root/RACK3/CPT_MDL/Environment/MDL_PAR/ManeuverControl/ManeuverStop[0|1]/Value'
        self._ResetVehicle = 'scania_rack3/Model
Root/RACK3/CPT_MDL/Environment/MDL_PAR/ManeuverControl/Reset_VehicleStates[0|1]
/Value'
        self._SegmentMatrix = 'scania_rack3/Model
Root/RACK3/MDL/Environment/Maneuver/MANEUVER_SCHEDULER/ManeuverScheduler_SFcn/P
15'
        self._TableData = 'scania_rack3/Model
Root/RACK3/MDL/Environment/Maneuver/MANEUVER_SCHEDULER/ManeuverScheduler_SFcn/P
16'
    def VehicleStartIgnON(self):
        """
        Construct of Vehicle start instance. Start virtual vehicle and Engine
in idle state
        """
        from Scania_VT.StandardVehicle.CAN import IO
        from Scania_VT.Driver import BasicDriver
        vtDriver = BasicDriver.CreateBasicDriver(self._vtExec)
        import time
        #Switch on power
        IO.IO(self._vtExec, self._PowerSwitch).Write(1)
        time.sleep(3)
        #Ignition ON
        vtDriver.IgnitionOn()
    def VehicleStartEngON(self):
        """
        Construct of Vehicle start instance. Start virtual vehicle and Engine
in idle state
        """
        from Scania_VT.Driver import BasicDriver
        vtDriver = BasicDriver.CreateBasicDriver(self._vtExec)
```

```
        #Engine ON
        vtDriver.EngineOn()
    def VehicleStop(self):
        """
        Construct of Vehicle stop instance. Stop virtual vehicle and reset
        """
        from Scania_VT.StandardVehicle.CAN import IO
        from Scania_VT.Driver import BasicDriver
        vtDriver = BasicDriver.CreateBasicDriver(self._vtExec)
        import time
        #Turn OFF engine
        vtDriver.TurnOff()
        time.sleep(3)
        #Stop the vehicle
        vtDriver.Stop()
        time.sleep(3)
        #Switch OFF power
        IO.IO(self._vtExec, self._PowerSwitch).Write(0)
    def StartUserManeuver(self):
        """
        Construct of start user maneuver instance. Download and start a user
defined maneuver.
        """
        from Scania_VT.StandardVehicle.CAN import IO
        import time
        #Gear position to drive
        IO.IO(self._vtExec,self._GearSelector).Write(3)
        time.sleep(3)
        #Release parking brake
        IO.IO(self._vtExec,self._ParkingBrake).Write(0)
        time.sleep(3)
        #Select Maneuver controlled model
```

```
        IO.IO(self._vtExec,self._ASMMode).Write(1)
        time.sleep(3)
        #Select User Maneuver
        IO.IO(self._vtExec,self._SelManeuver).Write(6)
        time.sleep(3)
        #Select User Road
        IO.IO(self._vtExec,self._SelRoad).Write(6)
        time.sleep(3)
        #Load maneuver
IO.IO(self._vtExec,self._SegmentMatrix).Write(self._MatlabSegmentMatrix)
        IO.IO(self._vtExec,self._TableData).Write(self._MatlabTableData)
        #Start maneuver
        IO.IO(self._vtExec,self._StartManeuver).Write(1)
        time.sleep(3)
        IO.IO(self._vtExec,self._StartManeuver).Write(0)
        time.sleep(3)
    def StopUserManeuver(self):
        """
        Construct of stop user maneuver instance.
        """
        from Scania_VT.StandardVehicle.CAN import IO
        import time
        #Stop maneuver
        IO.IO(self._vtExec,self._StopManeuver).Write(1)
        time.sleep(3)
        IO.IO(self._vtExec,self._StopManeuver).Write(0)
        time.sleep(3)
        #Reset vehicle
        IO.IO(self._vtExec,self._ResetVehicle).Write(1)
        time.sleep(3)
        IO.IO(self._vtExec,self._ResetVehicle).Write(0)
        time.sleep(3)
```

```
        #Select Manual controlled model
        IO.IO(self._vtExec,self._ASMMode).Write(2)
        time.sleep(3)
    def StartManeuverPlayback(self,vtSpeed):
        self._vtSpeed = vtSpeed
        from Scania_VT.Driver import BasicDriver
        vtDriver = BasicDriver.CreateBasicDriver(self._vtExec)
        vtDriver.DriveAtSpeed(self._vtSpeed)
class VtTCManeuver:
    def __init__(self, vtExec):
        """
        Construct of init instance. Initialize variables.
        """
        self._vtExec = vtExec
    def DriveVt(self,vtKeyPosPrev,vtKeyPosCurr,vtSpeed,vtRPM):
        from Scania_VT.StandardVehicle.Vehicle.Cockpit import Key
        key = Key.Key(self._vtExec)
        vtTCManeuverUtils = VtTCManeuverUtils(self._vtExec)
        import time
        if (vtKeyPosPrev == 0) and (vtKeyPosCurr == 1):
            vtTCManeuverUtils.VehicleStartIgnON()
            time.sleep(3)
            print "Ignition ON"
        elif (vtKeyPosPrev == 1) and (vtKeyPosCurr == 2):
            vtTCManeuverUtils.VehicleStartEngON()
            time.sleep(3)
            print "Engine ON"
        elif (vtKeyPosPrev == 2) and (vtKeyPosCurr == 1):
            vtTCManeuverUtils.VehicleStop()
            time.sleep(3)
            print "Stopping the vehicle"
        elif (vtKeyPosPrev == 2) and (vtKeyPosCurr == 2):
```

```
            if key.GetKeyState() != 'IGNITION_ON' or 'STARTER_ON':
                vtTCManeuverUtils.VehicleStartEngON()
                time.sleep(3)
            if (vtRPM >= 400):
                vtTCManeuverUtils.StartManeuverPlayback(vtSpeed)
                print "Driving to speed"
                print vtSpeed
            else:
                print "Idling the vehicle"
        else:
            print "Vehicle idle"
if __name__ == '__main__':
    from Scania_VT.TaFramework import VtExec
    vtExec=VtExec.VtExec.getInstance()
    vtExec.InitVariableAccess()
    vtTCManeuverUtils = VtTCManeuverUtils(vtExec)
    vtTCManeuver = VtTCManeuver(vtExec)
```

VITA