Paulo Jorge Sena Figueira

Test Automation Framework for Embedded Systems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Costa Seco, Assistant Professor, Faculty of Sciences and Technology - NOVA University of Lisbon

Co-advisers: Carla Ferreira, Assistant Professor, Faculty of Sciences and Technology - NOVA University of Lisbon

Jorge Manuel Lopes, System Architect, Altran Portugal

Examination Committee

September, 2018

Test Automation Framework for Embedded Systems

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

First and foremost, I would like to thank Professor João Costa Seco, Professor Carla Ferreira, and Jorge Manuel Lopes from Altran Portugal for their support and guidance. They were vital for the development of this dissertation.

I also would like to thank all my colleagues from FCT/UNL and Altran Portugal for the companionship and support.

I am very grateful to my friends. Their support, motivation, and entertainment gave me the strength to continue.

Last but for sure not least, I would like to express my thanks and love to my family. There is no distance (not even the ocean that separates us) that can break the love, support, and encouragement that I receive every day.

Abstract

Embedded systems are everywhere! Electronic systems in just about every engineering market segment are classified as embedded systems, consumer electronics, medical, automotive, avionics, etc. Embedded systems differ from more conventional systems, such as computers, because they are limited to the embedded hardware, are designed to perform a dedicated function and have high quality and reliability requirements.

Due to these characteristics, this type of system is strongly related to critical systems. Critical systems are systems that in the event of a failure can cause damage to living beings or the environment. Thus, it is necessary to ensure a high level of correctness in this type of systems. One way to increase the correctness of a system is through the process of testing. However, testing embedded systems presents a degree of difficulty because they are typically closed systems and work with real-time data that is difficult to reproduce and are non-deterministic.

In this way, and with the collaboration of Altran Portugal, we intend to solve this problem by developing a framework that allows test automation for embedded systems. Automating the test data creation and execution of test case increases the quality of these systems by identifying defects to be fixed in a more efficient way.

To this end, a survey of automation tools is done and each tool evaluated according to a set of criteria defined when designing the solution. The selected tool is Robot Framework, which is a widely used tool in the web and desktop application. Thus, integrating such a proficient tool in the embedded environment elevates the test automation in the embedded systems context.

Then, we test the concept developed in this dissertation by executing functional tests in embedded systems that follow a model-driven development approach.

Keywords: Test automation framework, Embedded systems, Test automation framework analysis, Software testing, Evaluation of test automation frameworks, Robot Framework

Resumo

Os sistemas embutidos estão em todo o lado! Sistemas electrónicos em qualquer mercado de engenharia podem ser considerados sistemas embutidos, seja em electrodomésticos, dispositivos médicos, automóveis, aviões, etc. Os sistemas embutidos diferenciam-se dos sistemas mais convencionais, como os computadores, pelo facto de estarem limitados ao *hardware* embutido, são construídos para desempenhar uma função e são sistemas que é necessário um alto nível de qualidade e de confiabilidade.

Devido a estas características, este tipo de sistema está fortemente relacionado com os sistemas críticos. Sistemas críticos são sistemas que em caso de ocorrência de uma falha podem causar dano a seres vivos ou ao ambiente. Assim, é necessário garantir um alto nível de correcção neste tipo de sistemas. Uma forma de aumentar a correcção de um sistema é através do processo de testes ao sistema. Contudo, testar sistemas embutidos apresenta um grau de dificuldade, pois tipicamente são sistemas fechados e trabalham com dados a tempo real que são difíceis de reproduzir.

Desta forma, e com a colaboração da Altran Portugal, pretende-se resolver este problema desenvolvendo uma ferramenta que permita automatizar o processo de testes a sistemas embutidos. Nomeadamente, na criação de dados de teste e na execução dos testes, que irá permitir uma subida na qualidade dos sistemas embutidos através da identificação de falhas que podem ser posteriormente resolvidas.

Para isso, será feito um levantamento de ferramentas de automatização de testes que irão ser avaliadas segundo um conjunto de critérios definidos aquando o desenho da solução, resultando na ferramenta mais adequada para o nosso problema. A ferramenta resultante da avaliação foi a Robot Framework. Esta ferramenta é muito utilizada para fazer a automatização de testes em contexto **web** e em contexto *desktop*. Assim, utilizar uma ferramenta com um nível avançado de maturidade num contexto de sistemas embutidos, permitirá elevar a automatização de testes neste mesmo contexto.

Finalmente, para testar o conceito desenvolvido na dissertação, é efectuado um processo de teste a sistemas embutidos que sigam a metodologia de desenvolvimento *model driven development*.

Palavras-chave: Ferramenta de automatização de testes, Sistemas embutidos, Análise de ferramentas de automatização de testes, Levantamento de critérios, Robot Framework

Contents

List of Figures

List of Tables

Listings

Acronyms

ECU	Electronic control unit.
FSM	Finite state machine.
MBT	Model-based testing.
RLP	Remote Library Proxy.
RLS	Remote Library Server.
SUT	System under test.
TAFES	Test Automation Framework for Embedded System.

Chapter 1

Introduction

Embedded Systems are everywhere! Electronic devices in just about every engineering market segment are classified as embedded systems, consumer electronics, office automation, networking, medical, industrial control, automotive, avionics, etc [1]. Embedded systems differ from more conventional systems, such as computers, because they are limited to the embedded hardware, are designed to perform a dedicated function, and have high quality and reliability requirements.

Due to these characteristics, embedded systems are strongly related to critical systems. Critical systems are systems where malfunctions can cause harm to living beings or to the environment. So, an important factor in the development of such systems is ensuring their correctness, which means that they behave according to their specification.

We can verify a system correctness through the process of testing, where we identify defects that can later be fixed and consequently improve the system correctness. However, testing in embedded systems has a degree of difficulty. Firstly, an embedded system is restricted to their own context and when we are performing a test in such systems it is required to know their functionalities and how they are triggered. Secondly, given the wide usage of embedded systems in every engineering market segment, some communication protocols are adopted within each segment, for example in the automotive embedded systems the standard communication protocol is CANBus. Moreover, embedded systems are highly related to real-time events which are non-deterministic and not easy to replicate.

Given this difficulty of testing embedded systems, it is required to create a solution that automates the testing process in embedded devices. In particular, a solution capable of translating the embedded systems operations into test scripts that are callable and execute functionalities, for example, turn on the charging of a device, or turn on GPS.

Test automation means the automation of testing activities including the development and execution of test scripts, verification of testing requirements and use of test tools.

Choosing a test automation approach before manual, allows the testing to be less time consuming and perform it with more efficiency[2]. These justifications allow products to have an earlier time to market, which is vital for a manufacturer success. Nevertheless, software testing when applicable can reduce the software costs, which may take more than 50% off the overall software development effort[3].

Upon this problem and with the collaboration of **Altran Portugal**, we intend to solve the hardship of testing embedded systems through a Test Automation Framework for Embedded System (TAFES), providing the testers various benefits that help them to develop, execute and report test scripts efficiently and consequently improve the system correctness.

The automation of a test can only be achieved by a test automation tool. However, choosing the right one for our context is not trivial. In the field of web application and desktop application, test automation has achieved high levels of maturity, there are very complete solutions in the market. Bringing such technology for the context of the embedded systems will elevate the testing process of embedded devices.

Therefore, it is required an analysis of different tools and then an evaluation process that selects the most suitable tool. For example, if it has the capability of communicating with an embedded device, the ability to perform different types of automation testing and how easy is to specify test cases. After the selection of the test automation tool to base our solution on, we investigate how automatically create test data and how to communicate with the embedded device.

When developing an embedded system, a usual method of development is model-driven development. This methodology has a primary focus in the design of models that unambiguously describe the embedded device. Moreover, models that describe the points of connection to the embedded device are part of these artifacts, where it will explicitly show the different functionalities of the system and how can they be triggered. Thus, the challenge for our solution is to interpret these models and generate automatically the test scripts that later will be called in the test specification. By having this functionality, our framework is capable of generating test scripts of any kind of system.

To finalize, we developed embedded systems that follow the model-driven development and through TAFES we have ensured that these systems are behaving accordingly to their specification.

1.1 Contributions

The main contributions of this dissertation are:

- An analysis of different test automation frameworks;
- Evaluation of test automation frameworks for embedded systems;

- Design and development of a test automation framework for automotive embedded systems, having a mechanism for:
  - Interpreting the test script and executing it on the embedded device;
  - Storing common test scripts;
  - Communicating with the embedded systems;
  - Automatically generate test scripts from an embedded system's model;

1.2 Document structure

The rest of the document is structured as follows:

- Chapter 2 - This chapter explains the concepts of embedded systems, software testing, methodologies of test case generation and also testing automation. These concepts are important as they permit to design the solution in an effective way.
- Chapter 3 - In this chapter, we explore the concept test automation framework, and take a look at different approaches that have an impact on the current days.
- Chapter 4 - Here we analyse the problem and propose a solution design. Moreover, we go through a tool selection to find the suitable framework for our solution.
- Chapter 5 - This chapter explain the selected test automation framework.
- Chapter 6 - Correlates the generic solution from chapter 4 and the selected test automation tool described in chapter 5. Here we establish and detail the architecture of our Test Automation Framework for Embedded Devices.
- Chapter 7 - In this chapter we test embedded devices with the implemented solution.
- Chapter 8 - Finally, this chapter presents the conclusions regarding this dissertation, as well as future improvements' suggestions.

Chapter 2

Preliminary Concepts and Constraints

This chapter presents multiple concepts that are fundamental to the development of this dissertation. We start by presenting the concept of Embedded Systems, taking a closer look at the automotive context based on [4]. Next, the base concepts of software testing, where we are highly based on the concepts and methodology provided by the book *The Art of Software Testing* [5] and the work *The basics of embedded software testing* [38]. After this, we relate embedded systems (section 2.1) with software testing (section 2.2), based on concepts written in *Testing Embedded Software* [6]. Next, we present an overview of test data generation methods [7], a topic that has a strong impact on the effectiveness and efficiency of the whole testing process [8–10].Finally, we present a brief examination of automated testing, in order to understand the need for this type of testing.

2.1 Embedded Systems

In order to develop this dissertation we have to understand what is an automotive embedded system, and how do they operate. To reach this goal, we first take a look at the concept of embedded systems and then correlate the concept with the automotive context.

An embedded system is a combination of hardware and software with the objective of executing some specific function. It can either be an independent system or a piece of a large system, but always limited to the hardware specification that is embedded with. Typically embedded systems required high reliability and quality requirements because they're vastly used in critical systems.

These systems are in several engineering markets, for example[1]:

- Automotive - ignition system, engine control, brake system, etc.
- Consumer electronics - televisions, cameras, etc.

- Industrial control - robotics and control systems, etc.
- Medical - Infusion pumps, prosthetic devices, etc.

We can clearly see that embedded systems are everywhere in multiple forms, dimensions, and functionalities. Consequently, various architectures can be designed in order to promote the required functionality. However, it is possible to identify a basic embedded system architecture.

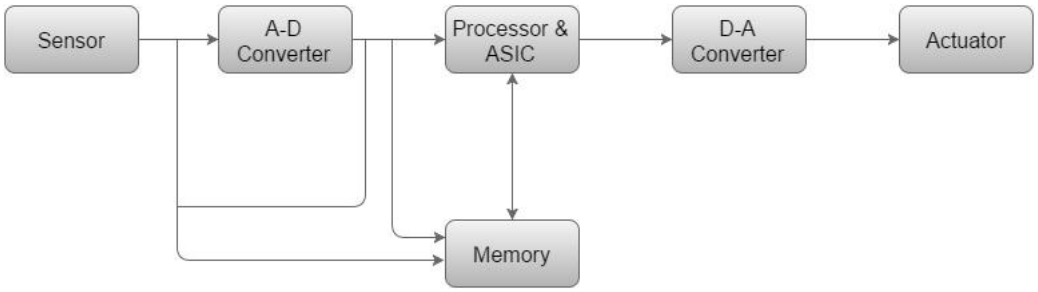


Figure 2.1: Basic embedded systems structure (in [39])

This structure is composed by the following elements[39]:

- **Sensor** - Collects data from the "outside world" and converts it to an electrical signal which can be read by the A-D Converter. The sensor can also store the data in the memory.
- **A-D Converter** - Converts analog signal by the sensor into a digital signal.
- **Processor & ASICs** - Process the data and store it in memory.
- **D-A Converter** - Converts digital data fed by the processor to analog data.
- **Actuator** - Given the data received from the D-A Converter they cause an event to the "outside world".

2.1.1 Automotive Context

With the rise of the capabilities and quality of these electronic components, the automotive world has begun to embrace embedded electronics in their products in order to provide improvements in functionalities, performance, comfort, safety, etc. If an embedded system controls any other electronic system in a vehicle it is called Electronic control unit (ECU).

According to (Navet et al., 2008), in-vehicle embedded systems are usually classified according to domains that correspond to different functionalities, constraints, and models. It is possible to divide them into functional domains, such as powertrain control, chassis control, active or passive safety systems and finally multimedia/telematics, body/comfort, and human-machine interface.

Power Train Domain - This domain is related to the systems that control the engine according to requests from the driver, for example speeding up or slowing down given the throttle position sensor or the brake pedal.

Chassis Domain - In the chassis domain, the ECU aim to control the interaction of the vehicle with the road. These ECU take as input the requests emitted by the driver (steering, braking, speed up), the road profile, and the environmental conditions in order to ensure the comfort and safety of the drivers. Example of elements that belong to this domain are systems like ABS, ESP, automatic stability control (ASC), and four-wheel drive (4WD).

Body Domain - The elements that belong to this domain are the wipers, lights, doors, windows, seats, and mirrors.

Multimedia, Telematic, and HMI - This domain is a "passenger-centric" functional domain. In this domain the ECU are responsible to control the interaction between the vehicle and other mechanisms outside the vehicle. For example, using systems that provide access to on-demand navigation, on-demand audio-video entertainment, Web surfing, etc.

Active/Passive Safety - In this domain, we have embedded systems that are in charge of the safety of the driver and passengers. To do so, it has two objectives: active safety and passive safety. The first refers to avoiding or minimizing an accident, an example of such systems are ABS, ESP, lane keeping, etc. Regarding passive safety systems, they help reduce the effects of an accident, for example, airbag, seat belts, etc.

2.1.2 Embedded Communication - CANBus

The CANbus was developed by BOSCH as a multi-master message broadcast system, which means that all nodes are capable of transmitting data and multiple nodes can request access to the bus simultaneously. It is widely used in embedded systems, with a higher impact in the automotive domain. This protocol as a maximum signaling rate of 1 megabit per second(bps) and unlike traditional networks such as USB or Ethernet, CAN does not send large blocks of data point-to-point from node A to node B under a supervision of a central bus master. In a CAN network, many short messages like temperature or RPM are broadcast to the entire network that will be received by all nodes or by none, which provides for data consistency in every node of the system.

The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP). CSMA means that each node must wait for a certain period of inactivity before attempting to send a message. CD+AMP means that collisions are resolved through a bit-wise arbitration, based on a previously programmed priority of each message in the identifier field of a message. Having a higher priority identifier will always grant bus access.

The ISO-11898:2003 Standard, with the standard 11-bit identifier, provides for signaling rates from 125kbps to 1Mbps. The standard was later amended with the extended

29-bit identifier. The Figure 2.2 illustrates the 11-bit identifier field in which is provides 2^{11} (2048) different message identifiers, meanwhile Figure 2.3 represents a 29-bit identifier meaning a total of 2^29 (537 million) different identifiers.

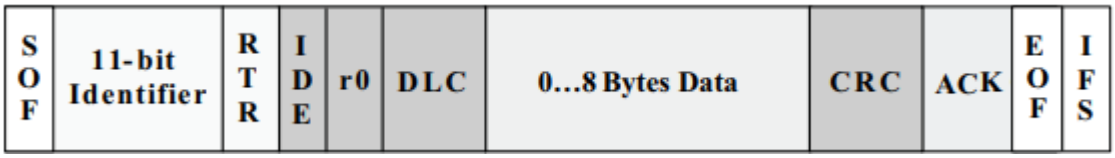


Figure 2.2: Standard CAN frame from [INTRODUTION to the CAN]

Regarding Figure 2.2 the meaning of the bit fields are:

- SOF - The single dominant start of frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.
- Identifier - The Standard CAN 11-bit identifier establishes the priority of the message. The lower the value, the higher is the priority.
- RTR - The single remote transmission request (RTR) bit is dominant when information is required from another node.
- IDE - A dominant single identifier extension (IDE) bit means that a standard CAN identifier with no extension is being transmitted
- r0 - Reversed bit
- DLC - The 4-bit data length code (DLC) contains the number of bytes of data being transmitted.
- Data - Contains the actual data values, up to 64 bits.
- CRC - The 16-bit (15 bits plus delimiter) cyclic redundancy check (CRC) contains the checksum of the preceding application data for error detection.
- ACK - Upon receiving an accurate message every node overwrites this recessive bit in the original message with a dominant bit, which indicates an error-free message has been sent. When a receiving node detects an error and leave this bit recessive, it discards the message and the sending node repeats the message after re-arbitration. Therefore, each node acknowledges(ACK) the integrity of its data, ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter
- EOF - The end-of-frame(EOF), is a 7-bit field that marks the end of a CAN frame.
- IFS - This is a 7-bit inter-frame space (IFS) that contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area.

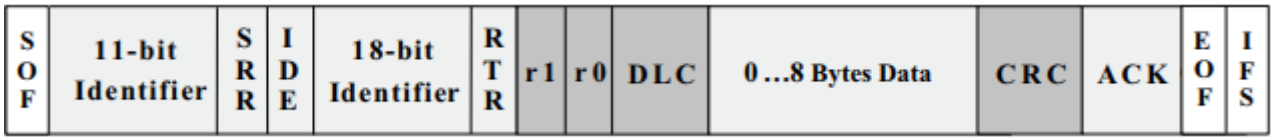


Figure 2.3: Extended CAN frame from [INTRODUTION to the CAN]

As described before the structure of extended CAN is an extension of the Standard CAN, Figure 2.3 illustrates it where we can see the addition of:

- SRR - The substitute remote request (SRR) bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- IDE - A recessive bit in the identifier extension (IDE) indicates that more identifier bits follow. The 18-bit extension identifier follows IDE.
- r1 - Additional reserve bit.

SockenCAN package is an implementation of CAN protocols for Linux. It uses the Berkeley socket API, the Linux network stack and implements CAN device drivers as networks interfaces. The CAN socket API has a similar design to TPC/IP protocol, allowing programmers that are familiar with network programming to easily learn how to use CAN sockets.

The Linux networking subsystem itself is highly flexible containing several networking protocols. Figure 2.4 illustrates different networking layers within the Linux kernel.

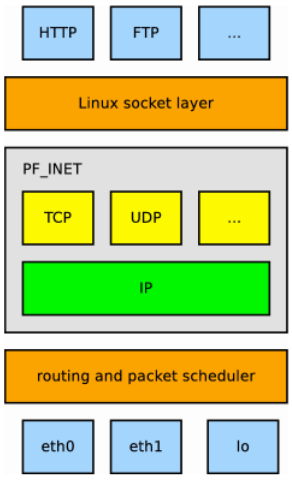


Figure 2.4: Linux networking subsystem in [11]

Starting at the application level exists a standard POSIX socket API defining the interface to the kernel. Underneath follows the protocol layer, which consists of protocol families, for example, PF_INET, that implement different networking protocols. Inside each family exists several protocols, in this case, TCP and UDP. Moreover, below this

level is where routing and packet scheduling layer takes place and finally it is followed by the layer containing the drivers for the networking hardware.

In order to have CAN networking to the Linux kernel, CAN support has been added to the existing networking subsystem. To do so, it has two crucial procedures:

- Have a new protocol family **PF_CAN** including a **CAN_RAW** protocol;
- Drivers for various CAN networking devices;

Figure 2.5 illustrates a network subsystem with CAN support.

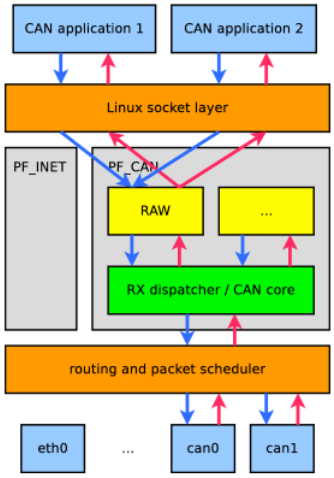


Figure 2.5: Linux socket based in [11]

Taking this approach brings several design advantages:

- Benefits from the existing and established POSIX socket API to assist the application developer.
- The new protocol family is developed against established abstractions layers, socket layer above and packet scheduler below.
- CAN network device drivers implement the same standardized networking driver model as Ethernet drivers.
- Communication protocols and complex filtering can be implemented inside the kernel.
- Support for multi-user and multi-application access to a single CAN interface is possible.

However, this design of SocketCAN has some drawbacks and limitations. Using a networking subsystem which was designed for a minimal Ethernet frame of 64-byte to the maximal 8 data bytes in a can frame bring memory overhead than simpler character device solution. Another problem is the fact that packet scheduler is a shared resource

among all networking devices(both Ethernet and CAN). Thus, heavy traffic on the Ethernet leads to delays in CAN traffic.

SocketCAN does not support hardware filtering of incoming CAN frames. Every CAN frames are received and passed to the CAN networking layer core, which processes the application specific filter lists. Having a hardware filter would lead to an overall reduction of the received CN traffic, but are a global setting. In a multi-user, multi-application scenario hardware filter is not feasible until the overall CAN design has been finalized and it is well known which CAN data is needed on the system.

SocketCAN's main goal is to provide a socket interface to userspace applications built upon the Linux network layers. In contrast to TCP/IP and Ethernet networking, CAN bus is a broadcast-only medium that no MAC-layer addressing like Ethernet. The CAN-identifier is used for arbitration of the CAN-bus, meaning that CAN-IDs have to be chosen uniquely on the bus. When designing a CAN-ECU network the CAN-IDs are mapped to be sent by a specific ECU.

2.2 Software Testing

When developing a software there's a considerable chance that it will have defects, and this chance increase with the complexity of the software. As it is impossible to produce defect-free software, we can resort to testing as a way of identifying the defects that can later be fixed. Hence, when testing a software we want to add value to it, which mean raising its quality or reliability through the identification of errors.

2.2.1 Introduction to the V-Model

To support this idea exists a development process known as V-Model.The V-Model represents a development process that follows the typical waterfall model with step-by-step stages. This model also illustrates the relationships between the development life cycle and testing, where every stage of the first is associated with the second.

Throughout the development life cycle occurs the software verification phase where every artifact produced(design documents and test cases) are reviewed.

1. **Requirements** - This is the initial phase of this process where the system requirements and analysis are performed. During this stage, it's designed the corresponding tests to be implemented later in the testing stages. In this phase occours the creation of the **acceptance tests**.

2. **Specification** - After defining the requirements for the system the next step is to generate a specification document. Also, in this phase the **systems tests** are written.

3. **Design** - Regarding the design phase of the model, it takes all the specifications written in the last stage and detail how the various components link with one another. During this phase, the **integration tests** are developed.

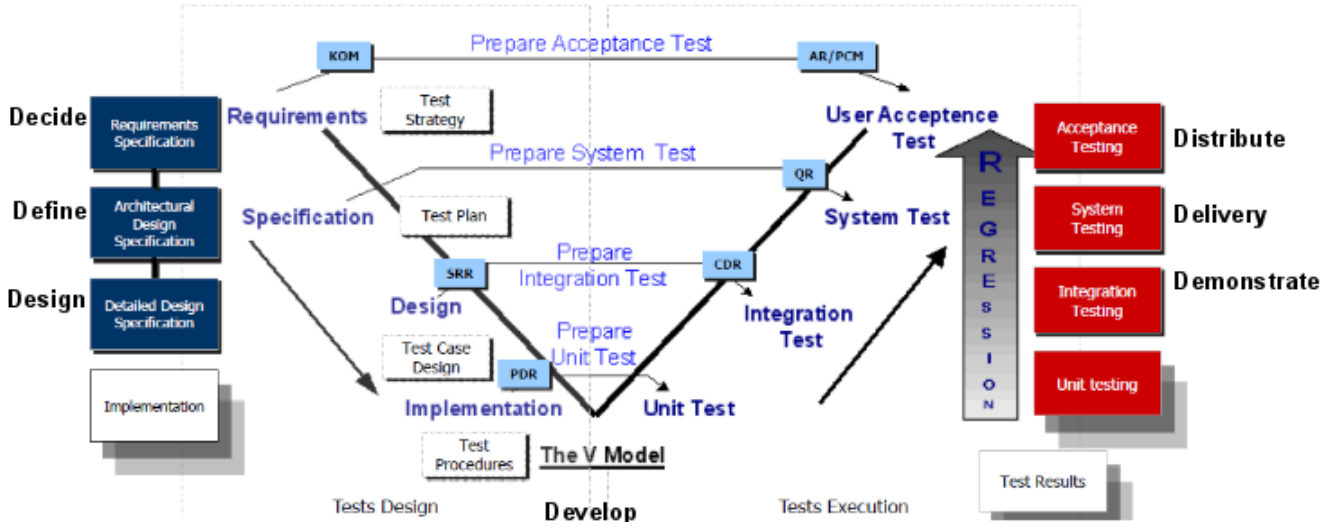


Figure 2.6: V-Model representation

4. **Implementation** - This is the stage where all the coding takes place. Furthermore, the developers also write down the **unit tests** that have to be executed on the code written.

After the implementation is the validation phase where the system is under multiple test executions to ensure that it meets the operational needs.

1. **Unit Testing** - The first phase of testing is the unit testing, where individual developers test at the module level by writing stub code to substitute for the rest of the system hardware and software. Tests focus on the logical performance of the code.

2. **Integration Testing** - After ensuring that a single module in logically correct, the next step is to test the combination of different models.

3. **System Testing** - This is the process of testing an integrated system to verify it meet the specified requirements.

4. **Acceptance Testing** - The last stage of the test execution process, in here it's done a formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

5. **Regression Testing** - When we are developing a system it isn't enough to pass a test once. Every time it is modified, it should be retested to assure the changes didn't break some unrelated behavior.

Besides the quality and reliability of the software, another motivation to test is to reduce cost on solving possible future defects. In a software development, the earlier

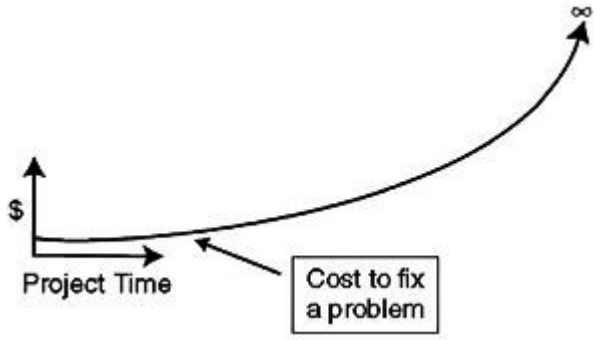


Figure 2.7: The cost to fix a problem as a function of the time in the product life cycle when the defect is found[38]

the bug is found the cheap it will be to fix it because the complexity is lower in the early stages. The figure 2.7, represents the relationship between the cost and the time of finding a defect.

2.2.2 Testing methods

Now that we have an idea of when and how to test a system, the next key point is which tests we use on the different stage of the testing process with the goal of having the highest probability of detecting defects. According to (Myers et al., 2011) it is impractical, often impossible, to find all errors in a program. This problem will have implications for the economics of testing, assumptions that the tester as to make about the program, and the manner in which test cases are designed. Therefore, to combat these challenges it is possible to assume some strategies, where **black-box testing** and **white-box testing** are the most prevalent.

Black-box testing, also known as functional testing or data-driven testing, views the program as a black box. The goal is to be completely unconcerned about the internal behavior. Instead, concentrate on finding circumstances in which the program does not behave according to the specifications. In this approach, test data are derived solely from the specifications (without taking advantage of the knowledge of the internal structure of the program)[5].

White-box testing, also known as coverage testing or logic-driven testing, allows you to examine the internal structure of the program. This strategy derives test data from an examination of the program's logic[5].

Regarding which test we use for a certain stage, the white-box testing is mainly used in the lower levels of testing(unit testing, integration testing). Consequently, black box testing is mainly applicable to the higher levels(system testing, acceptance testing, and also some integration testing).

2.2.3 Criteria for stopping the test process

Throughout this chapter many important questions related to software testing have been answered, leaving one last key point that needs to be addressed. The last key point is how to define a criterion where it is correct to assume that the system is fully tested and is bug-free. Unfortunately, it is impossible to claim such statement, because there's no way of knowing if the last error detected is the last remaining error. Nevertheless, the most common criteria are these [5]:

- Stop when the scheduled time for testing expires
- Stop when all the test cases execute without detecting errors

Although these are the most common criteria, they can be meaningless and counter-productive. Where in the first criterion you can satisfy it without doing any testing, therefore it doesn't measure the quality of testing. The second criterion is useless because it also is independent of the quality of the test cases.

To solve this problem [5] present three categories that are more useful than the described before.

The first is base completion on the use of specific test-case design methodologies. For example, in a unit test, we might define that a test is completed if it satisfies the multi-condition coverage, and a boundary value analysis of the unit specification and all resultant test cases found no error. However, this approach has some problems, one it isn't very helpful for a test phase in which specific methodologies are not available, such as systems test. Secondly, it is a subjective measurement, because there's no way to tell if the methodology was properly and rigorously used.

The second category is to state completion requirements in positive terms, this means setting a goal for the test. For example, when executing a test we set a goal of X number of errors, and we only stop testing when that goal is reached.

The third and last type of completion criterion is to plot the number of errors per unit time during the test phase. Obviously, it involves a lot of judgment and intuition when examining the shape of the curve in order to determine whether to continue the test phase or end it.

2.3 Software testing — Embedded context

In this section, we will discuss how the process of testing is managed when we want to test an embedded software. This is based on an approach called TEmb.

"TEmb is a method that helps to assemble a suitable test approach for a particular embedded system. It provides a mechanism for assembling a suitably dedicated test approach from the generic elements applicable to any test project and a set of specific measures relevant to the observed system characteristics of the embedded system." [6]

2.3.1 TEmb generic

As we have seen in section 2.2, when developing a new system different kinds of tests will be performed by multiple people, methods (automatic, manual), and multiple styles (unit, integration, systems, acceptence, regression). So at the start of the new project, we can produce a master test plan which will determinate who is responsible for which test and what are the relationships of the different tests. So, in each test level questions like "what", "when", "who", "which" and "by whom" show up.According to (Broekman et al.) to answer those questions the TEmb method introduce the following concepts, known as cornerstones:

- **Lifecycle** - In the lifecycle model, the main test activities are divided into five phases. The ordering of the phases is: planning & control(speed up and organize workflow of the testing process), preparation(testing techniques), specification(test cases), execution(testing) and finally completion.
- **Techniques** - Answers the question:"how", by defining standardized ways to perform certain activities. For example, techniques that design test cases, safety analysis, data-driven test automation, checklists, etc.
- **Infrastructure** - This will set all the facilities required for structured testing. It can be divided into three parts: facilities needed for executing the test (test environment), facilities that support efficient test execution(tools and test automation) and facilities for housing the staff(office environment). Regarding the test environment there are three important elements:
  - **Hardware/software/network** - A System under test (SUT) can have different appearances in different development stages. For instance a model, a prototype, isolated units connected to a simulator, production type. And for each of these stages, different test environment could be required.
  - **Test databases** - Tests can be repeatable, meaning that test data has to be stored.
  - **Simulation and measurement equipment** - If the SUT cannot run in the real world because it requires external signals, a solution is to simulate those signals. May also exist systems that produce output that requires special equipment for detection and analysis.
- **Organization** - Defines the roles and expertise required of those who perform the planned activities and the way they interact with others disciplines.

2.3.2 Mechanism for assembling the dedicated test approach

Each embedded system project has many specific concrete measures to achieve its goals and to solve its specific problems of testing. In the TEmb method, it is called *Mechanism*

for assembling the dedicated test approach, which is based on the analysis of risks and system characteristics. The measure based on risk analysis is a process of negotiating what should and should not be done, and set test priorities. Analysis of system characteristics is based on the characteristics of the SUT.

The system characteristics approach, as the name suggests, is the examination of the characteristics of the SUT, in order to help to answer the question "What makes this system special and what must be included in the test approach to tackle this?" [6]. Examples of system characteristics are:

- Safety critical systems - An embedded system is considered safety critical if a malfunction can cause serious damage to health. Examples of these systems are avionics, medical equipment, automotive. This type of systems requires a great risk analysis and rigorous techniques to analyze and improve reliability.

- Autonomous systems - Embedded systems that are designed to operate autonomously for an undefined period of time. Examples of these systems are traffics signaling systems. As this kind of systems is designed to work continuously and react to certain events without human intervention, a specific test environment with specific test tools is required in order to execute the test cases and analyze the results.

- Hardware restrictions - Towards this characteristic, an embedded system has limitations of hardware resources. Thus, it can compromise the embedded software, for example with the memory usage or power consumption. Systems with this characteristic require a significant amount of testing effort of a specialized and technical nature.

- State-based behavior - This characteristic tell us that the embedded system can be described in terms of a transition from a certain state to another state. So, with this characteristic, the output of a certain input depend on previous events. Upon a system with this behavior, we have to be careful with the planning of test cases and test automation.

- Hard real-time behavior - Real-time means that at an exact moment a certain input or output occurs, and it influences the system behavior. To test this kind of system the test cases must contain detailed information about the timing of input and outputs. Another point to have in consideration is that the result of test cases will usually be dependent on the sequence in which they are executed, having a huge impact on both test design and test execution.

- Control systems - Systems that interact with the environment by a continuous feedback mechanism. Between the system and the environment exists a relation where the system's output influences the environment, and consequently the environment influences the behavior of the system. Testing of such systems requires a simulation

of the environmental behavior. Examples of this system: industrial process control systems and aircraft flight control systems.

All these characteristics described above impact the testing process, because they raise issues that need to be solved by the test approach. Also, the specific measures of each embedded system help to solve certain issues related to its characteristic. With the TEmb method, we can attribute these specific measures to one of the four cornerstones referenced in 2.3.1. Thus, by connecting the embedded system characteristic with the special measure of every cornerstone, we have a LITO-Matrix, represented by the table 2.1.

System characteristic	***Lifecycle***	***Infrastructure***	***Techniques***
Safety critical	Safety test Load/stress test	Coverage analysers	FMEA / FTA Model checking Formal proof Rare event testing
Autonomous system	Hardware in the loop Software in the loop	Measurement probes	Rare event testing
Hardware restrictions		Host/target testing environment	Algorithm efficiency analysis
State-based behaviour		State modelling and testing tools	State transition testing Statistical usage testing Rare event testing
Hard real-time		Logic analyser Time measurement probes	Evolutionary algorithms
Control system	Hardware in the loop Software in the loop	Simulation of feedback-loop	Statistical usage testing Rare event testing Feedback control test

Table 2.1: LITO-Matrix, adapted from [6]

Table A.1, completes the table 2.1 by detailing the test levels and types performed in each technique.

With the representation of this matrix we conclude that systems with different characteristics asks for different test approaches. Thus, when testing an embedded system we need to know its characteristics to take full potential of the testing process and improve systems' correctness.

2.4 Methodologies for automated software test case generation

As we have seen in the previous sections, software testing is indispensable in the software development. However, testing can be expensive in terms of time and expense, and when the development isn't timely, testing will most likely be the phase where measures will be taken. So, reducing the time to test a system and increasing its efficiency is an enormous advantage. One solution to this problem is to automate the test case generation, so in this chapter, we will take a look at different approaches to reach this goal. These approaches are based by *An orchestrated survey of methodologies for automated software test case generation*[7].

2.4.1 Model-based test case generation

Model-based testing uses models of software systems in order to origin test suites. Here we take a look at the Model-based testing (MBT) regarding the behavior of the SUT, treating it as a "black-box" which accepts inputs and produces outputs. Thus, if the SUT has an internal state that changes when inputs are given and outputs produced, we can easily represent it as a model with the possible input/output sequences. Therefore, with a test selection algorithm, it is possible to generate test cases from the model by choosing a finite set from the potentially infinite set of sequences specified by the model. Regarding MBT exists three main schools:

- **Axiomatic approaches** - the Axiomatic foundation of MBT is based on some form of logic calculus. Given a conditional equation like $p(x) \implies f(g(x), a) = h(x)$ where f, g, and h are functions of the SUT, a is a constant, p a specified predicate, and x a variable, the objective is to find assignment to x such that the given equality is sufficiently tested [12].

- **Finite state machine (FSM) approaches** - In this approach, the model is formalized by a Mealy machine, where inputs and outputs are paired on each transition. Test selection derives sequences from that machine using some coverage criteria. We can consider the SUT as an FSM, is an "unknown" black-box, only exposed by its input and output behavior. Some of the practical FSM based MBT tools use structural coverage criteria, like transition coverage, state coverage, path coverage, etc. as a test selection strategy [13][14].

- **Labeled transition system approaches** - Regarding these approaches exist two ways to generate tests. The first is based on the input/output conformance (IOCO), defining a relation which describes conformance of a SUT with respect to a model [15]. The second approach is called interface automata [16], this approach makes testing conformance a two-player game, where inputs are the moves of the test generated from the model, with the objective to discover faults, and the outputs are the moves of the SUT with to objective to hide faults. Regarding test selection, IOCO is based

on coverage criteria[17], based on metrics[18] and based on test purposes [19]. Test selection in interface automata is based on state partitioning [20], based on traversal and coverage[21], and based on model slicing by model composition[22]

2.4.2 Combinatorial testing

Combinatorial testing is selecting a sample of input parameters that cover a prescribed subset of combinations of the elements to be tested. In this type of testing the parameters and their inputs (or configuration options and their setting) are modeled as sets of factors and values. For each factor, a set of values is values is defined. Test cases or specific programs configurations are generated by selecting a subset of the cartesian product of the values for all factors. As referenced in (Anand et al.,2013) if a program with five factors, each of them with three possible values, then 3^5 or 243 are the possible program configurations. Combinatorial interaction testing is the most common way of doing this sampling, where all t-way combinations of factors values are within the sample.

2.4.3 Adaptive random testing

According to (Anand et al.,2013), empirical studies have proved that failure-causing inputs tend to form contiguous failure regions. As a consequence, non-failure-causing inputs should form contiguous non-failure regions. That being said, if previously executed test cases didn't reveal a failure, the newer test cases should be far away from the already executed non-failure-causing test cases. Thus, test cases should be evenly spread across the input domain. This is the basic intuition for adaptive random testing, to generate a family of test case selection methods in order to improve the bug detection effectiveness of random testing by enforcing an even spread of randomly generated test cases across the input domain.

2.4.4 Search-based software testing

Search-based software testing is a process of generating test cases, or inputs of test cases using search-based algorithms, guided by a fitness function that captures the current test objective. SBST has been applied to a variety of testing goals:structural [23], functional[24], non-functional [25], and state-based properties[26]. This approach can also be applied in a wide and diverse range of domains: based on agents[27], aspects[28], interactions[29], integration[30], mutation[31], regression[32], stress[33] and web application[34]. The reason that this approach is widely applicable is that any test objective that can be measured is a candidate for this transformation into a fitness function.

2.5 Overview of testing automation

In this section, we look at what testing automation is and its importance in the software testing.

To begin with, testing automation is a technique where testers write some form of script that will be executed by some software with the goal of testing another software. It takes on a manual testing process and automate it through automation tools(allowing to write and execute test cases) [40], we detail the concept of automation tool in the section 3.1.

Performing test automation has the following benefits:

- Increase productivity;
- Increases software quality;
- Reduces testing time;
- Increase testing coverage;
- Reduction of repetitive work;
- Greater consistency;
- Human interaction is not required while execution

Regarding which test cases to automate, it should be the high-risk/critical, repeatedly executed, difficult or tedious to perform manually, and time-consuming [41].

Testing automation can be used to improve the software development, for example, in continuous integration. Continuous integration is a software development practice where team members integrate their work frequently. Each integration is verified by an automated build to detect integration errors as soon as possible, test automation has a strong impact on this phase of the continuous integration, allowing an efficient execution of test cases at the time of the new work integration. [42].

Chapter 3

Related Work

In this chapter, we expand the concepts referenced in the previous chapter and relate it to the main context of this dissertation. Therefore, we first take a look at the different approaches used in testing automation frameworks, mainly which types of frameworks exist. Next, we present an overview of distinct testing automation frameworks that have an impact in testing automation on the present days.

3.1 Test Automation Framework

As mentioned in section 2.5, in order to reach automation on test process we have to resort to some software. This third-party software is called **test automation framework** which allows the tester to develop, execute and report the automation test script in an efficient way [43]. Automation frameworks define automation guidelines, coding standards, concepts, processes, project hierarchies, modularity, reporting mechanism, test data injection, etc. to back up automation testing [44].

3.1.1 Types of Test Automation Framework

In this section, we explore the most popular [44]. Such as:

- **Module Based Testing** - This type of testing framework breaks the SUT into a number of logical and isolated modules. In each module, a test script is dedicated to a piece of the SUT and when all modules are grouped a larger test script is built representing more than one module. The separation between modules is done by an abstraction layer so that changes made in the SUT doesn't affect the whole module. Regarding this type of test automation framework, the main advantages are the high level of modularization and consequently how easy and cheap it is to maintain,

and scalability. However, it also presents disadvantages as the test data is directly embedded in the module, so if changes in test data are made it requires changing the test script of the corresponding module.

- **Library Architecture Testing** - Frameworks of this type have a similar approach to the module based ones. But instead of separating the application into different modules (test scripts), it divides the SUT into functions and creates a common function library for the SUT. Thus, test scripts in order to communicate with the SUT go through this common library. The advantages of this approach are high modularization, easy and low-cost maintenance, and finally reusability. The drawback is the complexity of this approach, and in case of changes in the test data, changes in the test script have also to be made.

- **Data Driven Testing** - Towards this approach, the test automation framework separates the logic and test data, storing the data in an external database(for example in XML, Excel, CSV files). The data is stored in "Key-Value" pairs and to access or populate the data it is used the key. The main advantage of this architecture is that it reduces the number of test scripts require to cover all scenarios. Changing the data on the external database will not affect the test scripts. Elevates the flexibility and maintainability, and a single test scenario can run with different test data values. The drawback is essentially the increase in terms of complexity because it needs new elements capable of reading data from an external source.

- **Keyword Driven Testing** - Much like data-driven testing this approach separates the test data from the test scripts, besides that it keeps the test scripts code in an external file. The code on this file is called keyword and is organized in a tabular fashion. Whenever in the main test script an specific code is needed and it belongs to the keywords file, the automation tool calls the corresponding keyword providing the respective arguments. One big advantage of this approach besides the ones referenced in data-driven testing is that the tester doesn't need scripting knowledge. Most of the cases the keyword is self-explanatory on its functionality. Another advantage is that as the keywords are external to the automation tool they can be used multiple times by multiple scripts. The drawback of this approach is that the tester has to have some idea of the keyword mechanism in order to take its full potential. Another drawback is that it becomes more complicated as the number of keywords grows.

- **Behaviour Driven Development** - Test frameworks associated with this type, have a test script language that is easy to read for various professionals. Behaviour-driven development describes the acceptance criteria in terms of scenarios which take the following form:

  Given some initial context,

When an event occurs,

Then ensure some outcomes.

A story behavior is simply its acceptance criteria, meaning if the SUT fulfill all the acceptance criteria then it's behaving correctly. These scenarios are written using the Given-When-Then template. The advantage of this approach is how easy it is to understand the test scripts by multiple team members, allowing a better communication between them. Another advantage is the possibility of writing the test scenarios before any development of the SUT. However, it is required some coding knowledge in order to execute the defined test scenarios in the SUT.

- **Hybrid Testing Framework** - A framework is considered a Hybrid Testing Framework if it has a combination of two or more approaches mentioned in this section.

Towards the description of these types of automation framework and the main objective of this dissertation, keyword-driven, library architecture, and data-driven are more relevant than the others. Keyword-driven is relevant given the fact that with this approach the tester doesn't need any scripting language to write the test cases and the keyword can be reusable. Library architecture for allowing the creation of a library with the most common functions of the SUT, from which we can execute commands for various embedded systems. Data-driven is also relevant because it separates the data from the test script giving a high flexibility on the test automation framework.

3.1.2 Examples of Automation Tools

Taking into consideration the analysis of a test automation framework and its most popular types, in this section, we present four different test automation tools. The reasons to choose these tool are: suggestion by Altran Portugal (Robot Framework section 3.1.2.1 and Unified Functional Testing section 3.1.2.2); comparable with Robot Framework (Gauge Framework section 3.1.2.3); popular in the embedded systems development (VectorCAST section 3.1.2.4).

3.1.2.1 Robot Framework

Robot Framework is a test automation framework for acceptance testing and acceptance testing development. This framework is python based and contains a keyword driven testing approach. It is possible to extend the capabilities by creating new test libraries implemented in Java or Python, or users can create new keywords from the existing ones.

There are many reasons to use Robot Framework, for example:

- Easy-to-use tabular syntax to create test cases.
- Reusability of keywords, and ability to create new keywords from the existing ones.
- Provides a library API for creating customized test libraries (Java or Python).

- Supports data-driven testing.
- Modular architecture.

Regarding the architecture of this framework, it is totally independent of the technology used by the SUT.

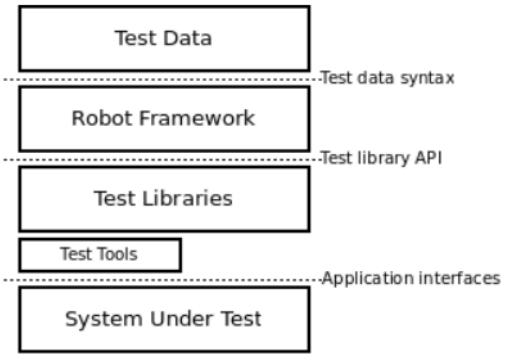


Figure 3.1: Robot Framework architecture (in [45])

As we can in the figure 3.1, Robot Framework presents a high level of modularity. Thus, to execute a test, the test data that is written in tabular syntax is processed by the core, executing the test case and generating logs and reports. The Robot Framework is unaware of the target under test, as the interaction with it is made by the test libraries.

3.1.2.2 Unified Functional Testing

The next tool we examine is Unified Functional Testing (UFT), which provides functional and regression test automation. This is a commercial tool developed and maintained by the company Micro Focus, and one of the most popular tools available in the market. In UFT the testing is highly focused on two aspects: GUI Testing and API Testing. With the possibility to combine both enabling to test functionality across multiple application layers, such as front-end GUI layer as well as back-end service layers. In our case, we are not interested in GUI testing, as in embedded systems there is no GUI. Regarding the API testing, it is interesting because embedded systems have an interface, and as our objective is to test the integration of different devices, testing their functionality through the interface is essential. However, an additional component has to be developed in order to enable UFT to embedded device communication.

Unified Functional Testing much like Robot Framework offers a keyword driven approach, besides that it also supports scripting of tests under the scripting language VBScript. Another approach that this tool allows is data-driven testing, being possible to have data in an external database and use it in the test case. Lastly, it is also possible to create test libraries [35].

3.1.2.3 Gauge Framework

Gauge framework is an open source test automation framework with the objective of writing tests that are comprehensible by all roles in a product development. In order to do so, exists the following Gauge Terminologies [46]:

- Specifications - Here will be the business layer test cases that can also represent the feature documentation. Basically, in this file will be described particular features of the SUT.
- Scenarios - They represent a single flow of a specification.
- Steps - These are the executable components of the specification. Every step has a code implementation(Java, C# or Ruby) which will be executed when the steps inside a specification are executed. They can also be parametrized, even with external files (Excel, CSV).
- Concepts - They provide the ability to re-use and combine steps into a single unit.

Regarding the type of automation, this framework follows behavior driven given the easy to read approach, and data-driven as it is possible to generate test cases with external data.

3.1.2.4 VectorCAST

VectorCAST is a family of products that automate the test process across the development life-cycle. These commercial tools are developed and maintained by Vector Software and have a high focus on the embedded systems.

Within this family of solutions we have:

- **VectorCAST/C++** - Automatically generate executable test harness for one or more units of application source code written in C/C++. Also, generate and execute test cases and report results.
- **VectorCAST/RSP** - This complements the VectorCAST/C++ allowing it to extend the testing one step further executing them in an embedded target environment.

When combining the two we have the following architecture:

When combining the two solutions the source code is parsed to generate the Test Harness through the VectorCAST/C++. Then, the VectorCAST/RSP automates the communication between the executable generated by the VectorCAST/C++ and the target embedded device, downloading the test harness to the target passing its test case data and retrieving results back from the target. Although this solution isn't ideal for the type of tests that we intend to execute, we still selected it as a relevant tool given its popularity in the embedded systems development.

3.2 Decision Analysis and Resolution

As we intend to select the most suitable framework for our test automation framework we have to resort to some evaluation process. That is the case of Decision Analysis and Resolution (DAR) that evaluates identified alternatives against established criteria.

In this process, we start by establishing the guidelines to determine which issues have to be subject to formal evaluation, in our case it's the tool selection. This formal evaluation process is represented in figure 3.2, and is constituted by the following actions:

1. Establish evaluation criteria for evaluating alternatives;
2. Identify alternative solutions;
3. Select evaluation methods for evaluating alternatives;
4. Evaluate the alternatives solutions using the criteria and methods previously established;
5. Select the recommended solutions from the alternatives based on the evaluation;

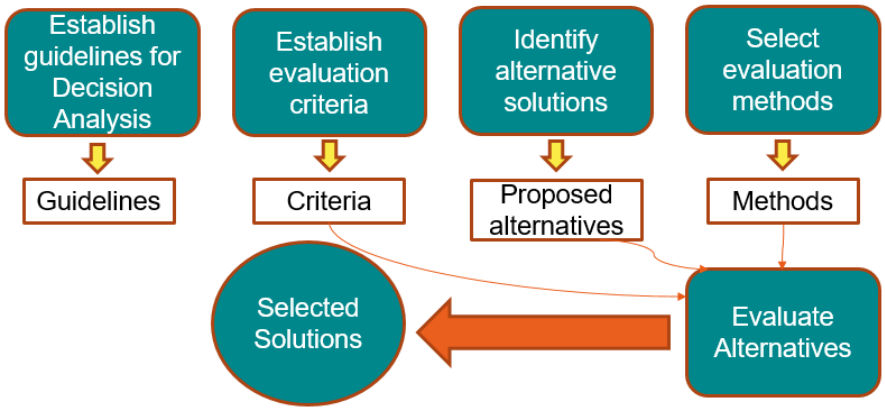


Figure 3.2: Formal evaluation process

After the execution of this process, the selected alternative is accompanied by documentation of selected methods, criteria, alternatives, and rationale for the recommendation.

Chapter 4

Solution Evaluation

This chapter presents the proposed solution to our problem. First, we present the design development of this solution and the rationale behind every step taken. Then, in section 4.2, we evaluate the different test automation tools referred in section 3.1.2 with the Decision Analysis and Resolution explained in section 3.2. In the end, we identify the most suitable framework to use in our proposed solution.

4.1 Architecture

As explained in the chapter 1, the objective is to create a framework able to execute tests in multiple embedded devices. Thus, at the start of this dissertation, the suggested solution is having a test automation framework that is capable of communicating with the embedded device in order to test its functionalities and integration with other components.

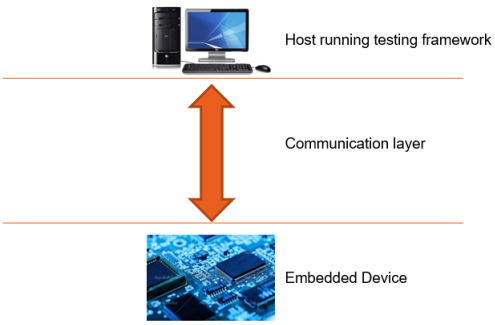


Figure 4.1: Suggested solution

Figure 4.1 is a representation of this suggested solution. However, it is too vague requiring more detail in each component. In order to reach a detailed solution, we resort

to the concepts of chapter 2 and the frameworks in chapter 3.

Taking into consideration the architecture of an embedded system and how do they communicate in the automotive context, we know that the ECUs can interact with each other through the communication methods described in the section 2.1.2. Hence, if two components are connected (one being the testing framework and the other the SUT) by a communication protocol with their interface exposed, it is possible to test the behavior of the SUT upon certain inputs.

Regarding host running testing framework we split it into two components, the testing tool, and an external component.

The **testing tool** component is in charge of the interaction between tester and framework. It is in this component where the tester specifies the test cases, that will be interpreted by the specification editor in order to create the test driver which executes the steps. During the test execution, the output values of the embedded device are given to the test driver and then used to generate a report by the test report.

The second component is called **external component** that is composed of library, communication and discovery mechanisms. The library is where all the common test scripts are located. The communication mechanism is in charge of setting up the communication between the test script and the embedded device. The discovery mechanism's functionality, as the name suggests, is discovering the functionalities of the embedded device. So, when testing an embedded system the discover will check for his functionalities publishing them to the library and then notify the tester. Following the discovery and under the assumption that a test case is running, every test step will call the library, then it will interact with the device through the communication mechanism.

The figure 4.2 presents the a more detailed view of the suggested solution (4.1).

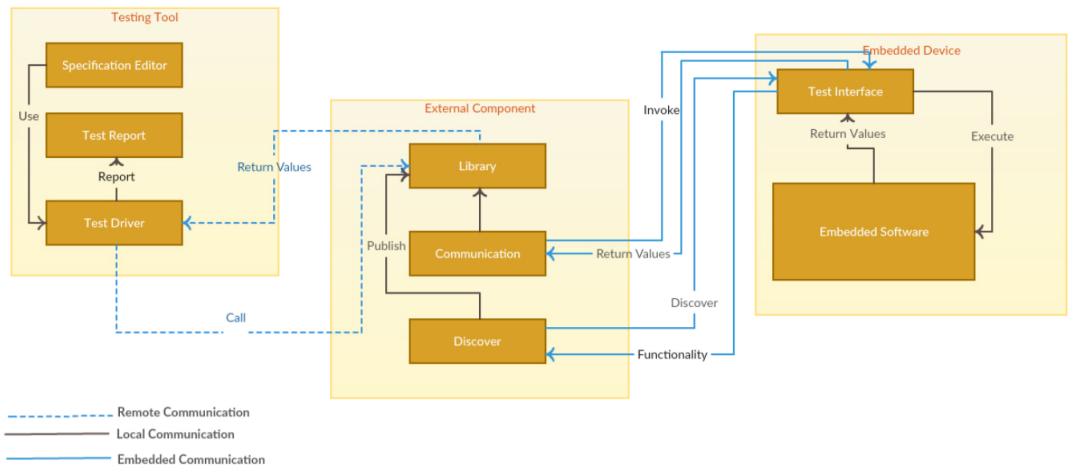


Figure 4.2: Proposed solution architecture

4.2 Tool Selection

Following the process described in section 3.2, the first step is to establish where and when should we use formal evaluation, in our case it is in the Tool Selection process. The second step is to establish the evaluation criteria, resulting in the following: test automation framework type; test driver; generation of test data; methods of communication; report; integration with other tools; learnability; supported platforms; code availability.

These are the main criteria and in each one of them are divided into sub-criteria in order to provide an objective evaluation. The third step of the tool selection process is identifying the alternatives solution which are the ones described in the section 3.1.2. The fourth is selecting the evaluation methods, consisting of a study of the framework and then an creation of a table that will illustrate the result of an alternative for a given criterion. The way that this table is built is by attributing a weight to each criterion and within that criterion set weights in each sub-criteria. Resulting in a sub-total value:

$$sub-total = \sum(sub-criteria \quad weight \quad * \quad sub-criteria \quad value) \tag{4.1}$$

Having the sub-total calculated we can calculate the total score for each alternative:

$$Total\ Score = \sum(sub-total \quad * \quad criteria \quad weight) \tag{4.2}$$

Regarding the criteria weight, the selected values are in table 4.1.

Table 4.1: Criteria and respective weight

Criteria	Weight(%)
Test automation framework type	18
Test Driver	10
Generation of Test Data	8
Methods of Communication	25
Report	15
Integration with other tools	2
Learnability	8
Supported platforms	4
Code availability	10
Total:	**100**

The reasoning behind the weights in table 4.1 is based on the importance of them in order to have a framework capable of reaching the main goals. As we intend to test embedded systems it is essential that the test automation framework interacts with the SUT, with the purpose of executing the test on the embedded system. Thus, the criterion regarding the **methods of communication** is the one with the highest weight.

Secondly, we want our solution to be capable of testing multiple devices with a little to no extra work needed in terms of configuration. Therefore, the criterion **Test automation framework type** is considered important, but not as important as the methods of communication criterion.

In third place of relevance for our solution we have the **report** criterion, the reason for this is that reporting and the way it is structured is critical to find any defect that might exist. With a good reporting, the tester can go to the root of the problem and solve it.

Next we considered **Test Driver** and **Code availability** equally important. The reason is that **Test Driver** criterion provides an easier approach to creating test cases which is something nice to have but not as important as the previous criteria, regarding the **Code availability** the reason for having a weigh of ten percent is that if an automation tool has this criterion it will be free of charge and access to the source code, therefore turning interesting when developing and new solution like in our case.

Regarding the **Generation of Test Data** criterion, it doesn't have much impact on the selection of the framework as in this solution we are highly concerned on the ability to test multiple embedded devices and **Generation of Test Data** is considered that helps the test case generation. **Learnability** is something important because when presenting the solution to new users if they can learn to operate in a very short time, the period of testing the devices will be larger and well explored.

The two left criterion **Supported platforms** and **Integration with other tools** are ranked lower because they don't interfere directly with the core functionality of the framework.

Now that the selection process is defined we present the result and its thought process for each criterion.

Test automation framework type, in this criteria we will evaluate the tool regarding its type. As we want our framework to be capable of test multiple embedded systems where the test steps are similar, the go-to type is Library Architecture, which will allow common functions to be called from multiple test scripts. Therefore, tools that follow this type are considered more relevant. Approaches like data-driven and keyword-driven are also interesting for our solution because they allow to cover all different possibilities of scenarios with different data and the tester doesn't need to possess scripting knowledge. Module based only approach is ranked lower because it relies on the test data being embedded into each module. Behaviour-driven is also ranked lower because the only value that it brings is the help in the engagement from multiple stakeholders towards the test process, it is something nice to have but not a must.

Evaluating each alternative against each test automation framework type criterion we have the table 4.2.

Table 4.2: Architecture sub-criteria evaluation

		Robot Framework		**UFT**		**Gauge**		**VectorCast**	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Keyword Driven	20	1	20	1	20	0	0	0	0
Data Driven	20	1	20	1	20	1	20	0	0
Behaviour Driven	10	1	10	0	0	1	10	0	0
Module Based	10	1	10	0	0	0	0	0	0
Library Architecture	40	1	40	1	40	1	40	0	0
	sub-total:		**100,00**		**80,00**		**70,00**		**0,00**

Test Driver, here we evaluate how much manual work has to be done to build a test case. In the best scenario, the alternative is managed through a GUI and no code is required. So, in this criteria what we value the most is the code required to generate a test case giving an evaluation of zero if it is required a lot of coding and one otherwise and 2/3 when work effort is average. Another criterion is if it has a GUI feature, which means if it is possible to create test cases through drag & drop functionality. To finalize we evaluate the framework in terms of parametrization of test cases, if it is possible to list input/expected values or over a range of values within the test case execution. The corresponding evaluation of each alternative against each test driver criterion is representable in the table 4.3.

Table 4.3: Test Driver sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
GUI feature	25	0	0	1	25	0	0	0	0
Amount of coding required to generate the drive	35	2/3	23,33	1/3	11,67	1/3	11,67	2/3	23,33
Parameterization:									
List of input values	15	1	15	1	15	1	15	1	15
List of expected values	10	1	10	1	10	1	10	1	10
Over a range of values	10	0	0	1	10	0	0	1	10
	sub-total:		**48,33**		**71,67**		**36,67**		**58,33**

Generation of Test Data, in this evaluation criteria we take in consideration the concepts of the section 2.4. The reason for selection this criterion is the ability to increase the testing process efficiency. Towards the different methods, the ones that are considered more important are combinatorial testing and random values, the reason behind this is the fact that both of these approaches are totally independent of the system and only take care of the input. Meanwhile, the model-based approaches (axiomatic, FSM, LTS) requires a model representation of the SUT. The table 4.4 represents the evaluation.

Table 4.4: Generation of Test Data sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Random Values	20	1	20	1	20	0	0	0	0
Axiomatic Approaches	15	0	0	0	0	0	0	0	0
Finite State Machines	15	1	15	0	0	0	0	0	0
Labeled Transition Systems	15	0	0	0	0	0	0	0	0
Combinatorial Testing	25	1	25	1	25	0	0	1	25
Search-Based Software Testing	10	0	0	0	0	0	0	0	0
	sub-total:		**60,00**		**45,00**		**0,00**		**25,00**

Methods of Communication, to test the embedded system our proposed solution has a host-target approach, where the framework is the host and the embedded system the target. Therefore, it is really important the ability that the alternative has to communicate with the embedded system (for example through CANBUS [4]). Another criterion that we also evaluate is the possibility of remote communication, with this feature it is possible to have the main test script in a computer an then, for example, through a raspberry PI communicate with the embedded system. Allowing a great portability and scalability

of the framework. In the table 4.5 we can find the evaluation made with respect to the communication criteria.

Table 4.5: Methods of Communication sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Remote Communication	30	1	30	1	30	1	30	0	0
Embedded Device Communication	70	1	70	1	70	1	70	1	70
	sub-total:		**100,00**		**100,00**		**100,00**		**70,00**

Report, as presented in the section 2.5 one of the fundamentals of a test automation framework is reporting the result of the test execution. Thus, in this criteria, we evaluate the alternatives against multiple forms of exporting the test reports and its logging. We classified the most important the ability to generate log files, with the full logging on the test execution. Then is the export format (HTML, CSV, TEXT, XML) and if they are generated automatically. Another criterion we evaluate the alternative is the possibility of customizing the test report. The table 4.6 summarizes the evaluation.

Table 4.6: Report sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
HTML	15	1	15	1	15	1	15	1	15
CSV	10	0	0	0	0	0	0	0	0
XML	15	1	15	1	15	1	15	0	0
Text	10	0	0	0	0	0	0	0	0
Logging	30	1	30	1	30	1	30	1	30
Automated	10	1	10	0	0	1	10	0	0
Customizable	10	1	10	1	10	1	10	1	10
	sub-total:		**80,00**		**70,00**		**80,00**		**55,00**

Integration with other tools, in this criterion we want to evaluate the alternative regarding its potential in integrating with different tools. For example, if we can integrate the framework with an IDE in order to increase the efficiency of the test cases development. When evaluating the alternative we find it relevant if it has an Open API, because if it does it allows a high integration with multiple tools through its API. Table 4.7 presents us the result of the evaluation.

Table 4.7: Integration with other tools sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Open API	70	1	70	0	0	0	0	0	0
IDEs	10	1	10	0	0	1	10	0	0
Other Tools	20	0	0	1	20	1	20	1	20
	sub-total:		**80,00**		**20,00**		**30,00**		**20,00**

Learnability, with this criteria we want to evaluate the alternative regarding how quick and easy it is to learn. The criterion with most weight is the learning curve, which means the rate of a person 's progress in gaining experience and skills. Then, it also took in consideration the documentation provided, if it presents a bad or unclear documentation

it will be classified with a zero and one if otherwise, in case of being neither really good or really bad it will be classified as 2/3. Finally, we examine how good the community is on each criterion. To illustrate the evaluation we have the table 4.8.

Table 4.8: Learnability sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Documentation	40	2/3	26,67	1	40	1/3	13,33	1	40
Learning curve	50	2/3	33,33	1/3	16,67	1	50	1/3	16,67
Large community	10	1	10	1	10	1	10	0	0
	sub-total:		**70,00**		**66,67**		**73,33**		**56,67**

Supported Platforms, it is convenient to have a solution that supports multiple platforms. Therefore we evaluate the alternatives against their supportability with two distinct platforms(Windows/Unix-like systems). The table 4.9 represents the evaluation done.

Table 4.9: Supported Platforms sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Windows	45	1	45	1	45	1	45	1	45
Unix Systems	55	1	55	0	0	1	55	1	55
	sub-total:		**100,00**		**45,00**		**100,00**		**100,00**

Code Availability, another nice to have characteristic of the framework is being open source. Typically means that it is free to use and its code is accessible. To summarize the evaluation done we have the table 4.10

Table 4.10: Code sub-criteria evaluation

		Robot Framework		UFT		Gauge		VectorCast	
Sub-criteria	Weightage(%)	Supports	**Result**	Supports	**Result**	Supports	**Result**	Supports	**Result**
Open source	100	1	100	0	0	1	100	0	0
	sub-total:		**100,00**		**0,00**		**100,00**		**0,00**

Now that all sub-criteria are evaluated the next step is to calculate the total score of each alternative. To do so, we first have to set weight on all criteria, which can be represented in the table 4.1. Then with the formula 4.2 we get the total score of an alternative, the sub-total is given in each table of sub-criteria evaluation.

Applying this method of evaluation we have the following result:

Table 4.11: Tool Selection result

	Robot Framework	Unified Functional Testing	Gauge Framework	VectorCAST
Total Score:	85,83	68,20	73,73	42,52

We can conclude that the most suitable test automation framework for our solution is the **Robot Framework**.

This framework has a test case editor (RIDE) delivering an easy write in the test cases, corresponding to the specification module presented in the figure 4.2. Robot framework has the functionality of remote library [45], in which it's possible to implement robot framework keywords with the programming languages Java or Python [45], corresponding to the library module of the proposed solution. The communication between the testing tool and the external component is ensured by the robot framework core functionalities (RemoteLibrary [45]).

Regarding the communication between the external component and the embedded device, it is ensured by existing Java or Python libraries for embedded communication protocol (for example python-can [47]).

Chapter 5

Robot Framework - Detailed

In this chapter we analyse Robot Framework to understand how to implement our proposed solution. We examine two major functionalities of Robot Framework, the creation of **test libraries** and the **remote library interface**. With both of these functionalities, we are able to create the test library in the Remote Library Proxy and execute its keywords remotely.

Robot Framework as previously described is a powerful test automation framework widely used in different fields of software testing. One of its many perks is the ability to extend the framework with new custom test libraries that are composed of keywords that interact with the SUT. As Robot Framework is developed in Python, test libraries can be easily implemented by this language. Besides Python it is also possible to implement libraries in Java, however it requires the usage of Jython to be interpreted by Robot Framework. We have selected Python given the high usage of the C programming language in embedded context and Python provides a well-constructed module (ctypes) to communicate with this language.

This framework has three different test library APIs:

- Static API - This approach takes in a module (Python) or class (Python or Java) and its corresponding methods and maps them into keyword names that later are used by the test case. Keywords generated also take the same parameters as the methods that implement them. It is also possible to report failures with exceptions, log by writing to standard output, and return values with the usual return statement.

- Dynamic API - This type of library must be a class that implements a method to get the names of the implemented keywords (*get_keyword_names()*) as well as a method to execute a named keyword with given arguments(*run_keyword(name,args)*). Keyword's names and how they are executed is determined dynamically at runtime.

```
*** Settings ***
Library     MyLibrary     10.0.0.1     8080
Library     AnotherLib    ${VAR}
```

Figure 5.1: Import library with arguments (in [45])

Regarding the reporting status, logging and returning values it takes the static API approach.

- Hybrid API - As the name suggests this is a hybrid between static and dynamic API. Here libraries are classes with a method who tells them which keywords they implement (*get_keyword_names()*), but unlike Dynamic API it does not have the method to run keywords *run_keyword(name,args)*. Other features besides discovering what keywords are implemented are similar as in the static API.

5.1 Test Library

5.1.1 Test Library Name

Test libraries are implemented as Python modules or Java classes. Thus, when creating a library it will take the name of the given Python module or Java class. For example, having a python module *MyLibrary.py* it will create a test library with name *MyLibrary*. Moreover, it is possible to have classes inside a Python module where it will implement a certain library. If this class name is the same as the module the library Robot Framework allows dropping the class name when importing the library. For example, class *MyLibrary* in the module *MyLibrary.py* can be used as a library with name *MyLibrary*. In another hand, if the class' name is different, for example *MyLib*, the library name will be *MyLibrary.MyLib*.

5.1.2 Configuring a Test Library

When implementing a test library as a class it is possible to provide arguments. The arguments are specified in the Setting table after the library name. When Robot Framework creates an instance of the imported library, it passes them into its constructor. Figure 5.1 represents an example of passing arguments to a library in the Settings section of the test case.

A representation of the implementation of *MyLibrary* can be found in Figure 5.2, where the constructor has two arguments that locates the library with a host and port.

5.1.3 Static keywords

5.1.3.1 What methods are considered keywords

In static library API, we have a library class/module and Robot Framework uses reflection to find its public methods. In libraries implemented with Python, all methods starting with an underscore will be excluded, regarding libraries in Java it will be the private

```python
from example import Connection

class MyLibrary:

    def __init__(self, host, port=80):
        self._conn = Connection(host, int(port))

    def send_message(self, message):
        self._conn.send(message)
```

Figure 5.2: Implementation of a library (in [45])

methods. The remaining methods are considered keywords and are callable by the Robot Framework. Figure 5.3 is a representation of a implementation of a test library in Python, where the method *my_keyword* is considered keyword and the method *_helper_method* is excluded.

```python
class MyLibrary:

    def my_keyword(self, arg):
        return self._helper_method(arg)

    def _helper_method(self, arg):
        return arg.upper()
```

Figure 5.3: Method considered keywords (in [45])

Methods that are in a base class and functions imported into the module namespace are considered keywords. For example in Figure 5.4 if the module is used as a library, it will have keywords: *Example Keyword*, *Second Keyword*, and *Current Thread*, the third comes from a Python's standard library and it returns the current *Thread* object.

```python
from threading import current_thread


def example_keyword():
    print 'Running in thread "%s".' % current_thread().name

def second_example():
    pass
```

Figure 5.4: Import methods to library (in [45])

5.1.3.2 Keyword names

When implementing a custom test library the keyword names are the method names. Thus, keywords written in the test data are compared with methods names to find the implementation method. This comparison is case-insensitive, moreover, spaces and underscores are also ignored. Figure 5.5 have an implementation of two methods(*hello*,*do_nothing*). Given the properties of robot framework, if we want to call method

hello, it is possible by having a keyword named *Hello*, *hello* or even *h e l l o*. Regarding method *do_nothing* we can call it with a keyword named *Do Nothing*.

```
def hello(name):
    print "Hello, %s!" % name

def do_nothing():
    pass
```

Figure 5.5: Set keyword names (in [45])

Figure 5.6 represents an example of a test case that uses the defined library *MyLibrary*, and keywords *Hello* and *Do Nothing* from that library.

```
*** Settings ***
Library    MyLibrary

*** Test Cases ***
My Test
    Do Nothing
    Hello    world
```

Figure 5.6: Example of a test case with defined keywords (in [45])

5.1.3.3 Keyword arguments

In static and hybrid APIs, keyword arguments are directly related to the method that implements that keyword. Meaning that the number of arguments needed for the keyword is the same as the number of arguments defined in the method. For example, if a method takes no argument the keyword will also take no argument, or if it takes one the keyword will also take one, and so on.

```
def no_arguments():
    print "Keyword got no arguments."

def one_argument(arg):
    print "Keyword got one argument '%s'." % arg

def three_arguments(a1, a2, a3):
    print "Keyword got three arguments '%s', '%s' and '%s'." % (a1, a2, a3)
```

Figure 5.7: Keywords with arguments (in [45])

Figure 5.7 represents the implementation of three keywords in which the first takes no arguments, the second one, and the third takes three, resulting on the keywords: *No Arguments*, *One Argument*, *Three Arguments*.

As the keyword arguments are dependent on keyword implementation, and in our case, it's in python. We can take advantage of this language having multiple types of arguments:

- Arguments with default values.

- Variable number of arguments(*varargs).
- Free keyword arguments(**kwargs).

Given that arguments in python do not hold any information regarding the type, there is no possibility to automatically convert the argument in the test case to the corresponding type in the library. Thus, calling a python method implementing a keyword with the correct number of arguments always succeeds. However, the execution fails later if the arguments are incompatible.

5.1.4 Communicating with Robot Framework

When executing a certain keyword we are expecting that it somehow communicates with the SUT and then sends messages to the Robot Framework, returning information about that execution and use it in log files, variables defined in the test case and most importantly report if a keyword has passed or not.

How to report a keyword status To report a keyword status we resort on exception, which means if an executed method raises an exception the keyword status is **Fail**, otherwise its status is **Pass**.

Thus, the error message shown in logs, reports, and console is created from the exception message type and its message.

Logging information Exception messages are not the only way of sending information to the users, another way to do so is setting the methods in order to send messages to the log file of Robot Framework by writing to the standard output stream (stdout) or to the standard error stream (stderr). Logging has different levels:

- Fail - Used when a keyword fails.
- Warn - Used to display warns.
- Info - This is the default level for normal messages. Messages below this level are not shown in the log file.
- Debug - Used for debugging purposes. Useful for logging what libraries are doing internally.
- Trace - More detailed debugging level. The keyword arguments and return values are automatically logged using this level.

Info is the default level when sending information to Robot Framework. To change the logging level it has to be set in the message itself following the format: *LEVEL* LOG MESSAGE. Thus, for example, if we want to have a warn log we just had to send

a message: *WARN* LOG MESSAGE. Figure 5.8 illustrates an example of different log levels.

```
print 'Hello from a library.'
print '*WARN* Warning from a library.'
print '*ERROR* Something unexpected happen that may indicate a problem in the test.'
print '*INFO* Hello again!'
print 'This will be part of the previous message.'
print '*INFO* This is a new message.'
print '*INFO* This is <b>normal text</b>.'
print '*HTML* This is <b>bold</b>.'
print '*HTML* <a href="http://robotframework.org">Robot Framework</a>'
```

```
16:18:42.123     INFO   Hello from a library.
16:18:42.123     WARN   Warning from a library.
16:18:42.123    ERROR   Something unexpected happen that may indicate a problem in the test.
16:18:42.123     INFO   Hello again!
                        This will be part of the previous message.
16:18:42.123     INFO   This is a new message.
16:18:42.123     INFO   This is <b>normal text</b>.
16:18:42.123     INFO   This is bold.
16:18:42.123     INFO   Robot Framework
```

Figure 5.8: Log levels (in [45])

Return Values Another way of communication to the robot framework is returning the information retrieved from the SUT or generated by other means. These values can be assigned to variables in the test case and used as inputs for others keywords.

To return these values it is used the *return* statement from Python method implementing the keyword.

```
from mymodule import MyObject

def return_string():
    return "Hello, world!"

def return_object(name):
    return MyObject(name)
```

```
*** Test Cases ***
Returning one value
    ${string} =    Return String
    Should Be Equal    ${string}    Hello, world!
    ${object} =    Return Object    Robot
    Should Be Equal    ${object.name}    Robot
```

Figure 5.9: Return values in test case (in [45])

Figure 5.9 is a example of returning values to Robot Framework core. As shown it is possible to send one value into one scalar variable, as well in a more complex variable in case of an object value.

Moreover, the return values can be assigned into several scalar variables at once, into a list variable, or into a scalar variable and list variable. To do so, the returned values represent Python lists or tuples, Figure 5.10 illustrates this functionality.

```python
def return_two_values():
    return 'first value', 'second value'

def return_multiple_values():
    return ['a', 'list', 'of', 'strings']
```

```
*** Test Cases ***
Returning multiple values
    ${var1}    ${var2} =    Return Two Values
    Should Be Equal    ${var1}    first value
    Should Be Equal    ${var2}    second value
    @{list} =    Return Two Values
    Should Be Equal    @{list}[0]    first value
    Should Be Equal    @{list}[1]    second value
    ${s1}    ${s2}    @{li} =    Return Multiple Values
    Should Be Equal    ${s1} ${s2}    a list
    Should Be Equal    @{li}[0] @{li}[1]    of strings
```

Figure 5.10: Retuning values in list or tuples (in [45])

5.1.5 Dynamic library API

In the upper sections we have seen how to create a test library, however, it is limited to the static and hybrid library API. Dynamic library API takes a few different approaches.

In most ways, it is similar to the static API, for example when reporting keywords status, logging or returning values. Moreover, the importing a dynamic library to the test case is equal to importing a static or hybrid library. Therefore, users who are writing test cases cannot treat all libraries in the same way.

The differences between static and dynamic libraries are the way of discovering what keywords are implemented, their arguments, documentation, and implementation. As we have seen in the static API this is done using reflection, but in dynamic libraries, it is done through special methods that will be used for these purposes.

Additionally, with dynamic library API, it is possible to implement a library so that it works as a proxy for an actual library possibly running on other process or even on another machine. Given that keywords names and all other information is obtained dynamically, there is no need to update the proxy when new keywords are added to the actual library.

Getting keyword names Dynamic libraries must have the method *get_keyword_names* responsible for telling what keywords are implemented. This method takes no arguments and returns a list with the names of the keywords that the library implements.

In case of being a dynamic library with methods meant to be keywords and methods which are meant to be private helper methods, keyword method should be marked facilitating the implementation of method *get_keywords_names*.

```python
from robot.api.deco import keyword

class DynamicExample:

    def get_keyword_names(self):
        return [name for name in dir(self) if hasattr(getattr(self, name), 'robot_name')]

    def helper_method(self):
        # ...

    @keyword
    def keyword_method(self):
        # ...
```

Figure 5.11: Get keyword names with decorator (in [45])

Figure 5.11 illustrates an implementation of *get_keywords_names* that checks if methods have the attribute *robot_name* which is setted by an decorator. If a method has this attribute it is considered an keyword and its name will be returned.

Running keywords For executing keywords of a dynamic library it is required to have a special method called *run_keyword*. Whenever a keyword from a dynamic library is used in the test data, Robot Framework resources to this method to execute the keyword. This method receives two to three arguments, the first being a string with the name of the keyword to be executed, the second a list of arguments given to the keyword, the last is an optional argument to use free keyword arguments (**kwargs).

With the keyword name and arguments, the library is free to execute the keyword. It must use the same mechanism to communicate with the framework as static libraries, which means that it must use exception for reporting keyword status, logging by writing to the standard output, and use the return statement to return values.

Getting keyword arguments Another method used in dynamic library is *get_keyword_arguments* which has the functionality of getting arguments of a certain keyword.

Unlike methods previously described(*get_keyword_names* and *run_keyword* this method is not require to execute keywords. However, this is problematic as its non-existence will create a doubt in the number of arguments of a keyword. As most keywords expect a certain number of arguments this will be problematic.

Therefore, it is recommended to have *get_keyword_arguments* implemented as it will tell Robot Framework what arguments is a keyword expecting to receive. This method has as input the name of a keyword and returns a list of strings containing the arguments accepted by that keyword.

Dynamic keywords can take any number of arguments, have default values, accept a variable number of arguments and free keyword arguments.

If *get_keyword_arguments* is missing or return a *None* for a given keyword, it means that keyword gets a specification on accepting all arguments. An example of a library that uses the dynamic API is Remote Library, section 5.2 explores this library.

5.1.6 Hybrid API

The last library API that we describe is the Hybrid API. As the name suggests this is a mix between the static API and the dynamic API.

Getting keyword names To get all keyword names in hybrid API it is used the same procedure as in dynamic API. So, the library needs a method called *get_keyword_names* and much like in dynamic API this method has the mission of returning a list of all keywords in the library.

Running keywords In getting keywords, hybrid library API follows the same policy of dynamic API. However, in executing the keywords it takes a different path from the dynamic API as it does not exist a method for running keywords. Instead, it uses reflection to find the implementation of keywords like static API.

With hybrid library API, it is possible to have the methods implementation directly on the library or dynamically.

As Robot Framework allows us to write libraries in Python it is easy to handle missing methods dynamically with the method *__getattr__*. Figure 5.12 illustrates an example of handling missing methods dynamically.

```
from somewhere import external_keyword

class HybridExample:

    def get_keyword_names(self):
        return ['my_keyword', 'external_keyword']

    def my_keyword(self, arg):
        print "My Keyword called with '%s'" % arg

    def __getattr__(self, name):
        if name == 'external_keyword':
            return external_keyword
        raise AttributeError("Non-existing attribute '%s'" % name)
```

Figure 5.12: Handling missing methods (in [45])

In this example, the method *__getattr__* is not executing the keyword, like *run_keyword* in dynamic API. Instead, it returns a callable object that Robot Framework can execute later.

Getting keyword arguments and documentation Regarding getting the keyword arguments and documentation it takes a route similar to static API. Robot Framework uses reflection to find the implementation methods of the keywords, thus searches for arguments and documentation.

5.2 Remote Library Interface

The remote library interface is a Robot Framework feature for having test libraries on different machines than where Robot Framework itself is running, furthermore, it allows us to implement libraries with other languages than Python and Java. Remote libraries look pretty much the same as any other test library, thus developing test libraries using the remote library interface is similar to create regular test libraries.

Remote library interface is provided by a special library of Robot Framework called Remote. This library does not have keywords of its own, instead, it acts as a proxy between the core framework and the implemented keywords elsewhere which are reached through remote servers. The communication between **Remote Library** and the servers are through a remote protocol on top of an XML-RPC channel. Figure 5.13 illustrates a high level architecture of Remote Library.

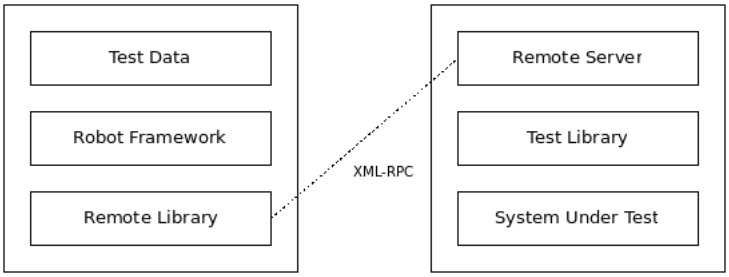


Figure 5.13: High level architecture of Remote Library (in [45])

When importing a Remote Library to the test case it is required to specify where the remote server is located by giving its address. Figure 5.14 shows three examples of importing remote libraries.

```
*** Settings ***
Library    Remote    http://127.0.0.1:8270       WITH NAME    Example1
Library    Remote    http://example.com:8080/    WITH NAME    Example2
Library    Remote    http://10.0.0.2/example    1 minute    WITH NAME    Example3
```

Figure 5.14: Import a Remote Library (in [45])

In the regular test library, the import of a libray was performed by using the **Library** command and then complement it with the module, for example, **Library.py**. However, in the remote library it is used the **Remote** and then the address of the remote library. In the example of the Figure 5.14 it is used a *WITH NAME* syntax that sets a name to the library. It is useful when using multiple libraries. Moreover, in the last line, it has an extra parameter regarding a time interval. This will be used to create a timeout to the remote library when performing the initial connection. Obviously, this time has to be set carefully to prevent interrupting a keyword execution.

Given that the remote library interface is written on top of XML-RPC, which is a remote procedure protocol using XML over HTTP, it is possible to implement it with

the most mainstream languages has they support XML-RPC. However, it has to follow some requirements in order to successfully communicate with Robot Framework. Remote server in XML-RPC has to have specific methods in its public interface similar to the dynamic library API, *get_keyword _names* and *run_keyword* are obligatory, but *get_keyword_arguments*, *get_keyword_tags* and *get_keyword_documentation* are also recommended. The explanation of these methods is in section 5.1.5.

Remote servers can act as wrappers for the real test libraries, or implement keywords. Additionally, method *stop_remote_server* should also be in the public interface to ease the server termination and to enable the possibility to call it in the test data regardless of the test library.

As the XML-RPC protocol does not support all possible object types, remote servers have to follow a set of rules in the returning of values as well as the keyword arguments received from the Robot Framework:

- String, numbers, and Boolean values are passed without modifications.
- Python *None* is converted to an empty string.
- All lists, tuples, and other iterable objects (except strings and dictionaries) are passed as lists so that their contents are converted recursively.
- Dictionaries and other mappings are passed as dicts so that their keys are converted to string and values converted to supported type.
- Returned dictionaries are converted to dot-accessible dicts allowing the access to keys as attributes like *${result.key}*
- Strings containing bytes in the ASCII range that cannot be represented in XML(e.g the null byte) are sent as Binary objects that internally use XML-RPC base64 data type. When received they are converted to byte strings.
- Other types are converted to strings.

Robot Framework already disposes of generic remote servers for various languages like Python, Java, Ruby, .NET, and so on.

Chapter 6

TAFES Architecture

In the proposed solution (chapter 4) we have selected three main components: testing tool, external component, and embedded device. These three components follow a generic approach which is totally independent of a test automation tool. However, we have selected the most suitable tool to implement this solution, the Robot Framework. Therefore the previously designed components will suffer changes and adapt to the selected framework. Figure 6.1 represents the end result of adapting the proposed solution to Robot Framework, consequently representing TAFES architecture.

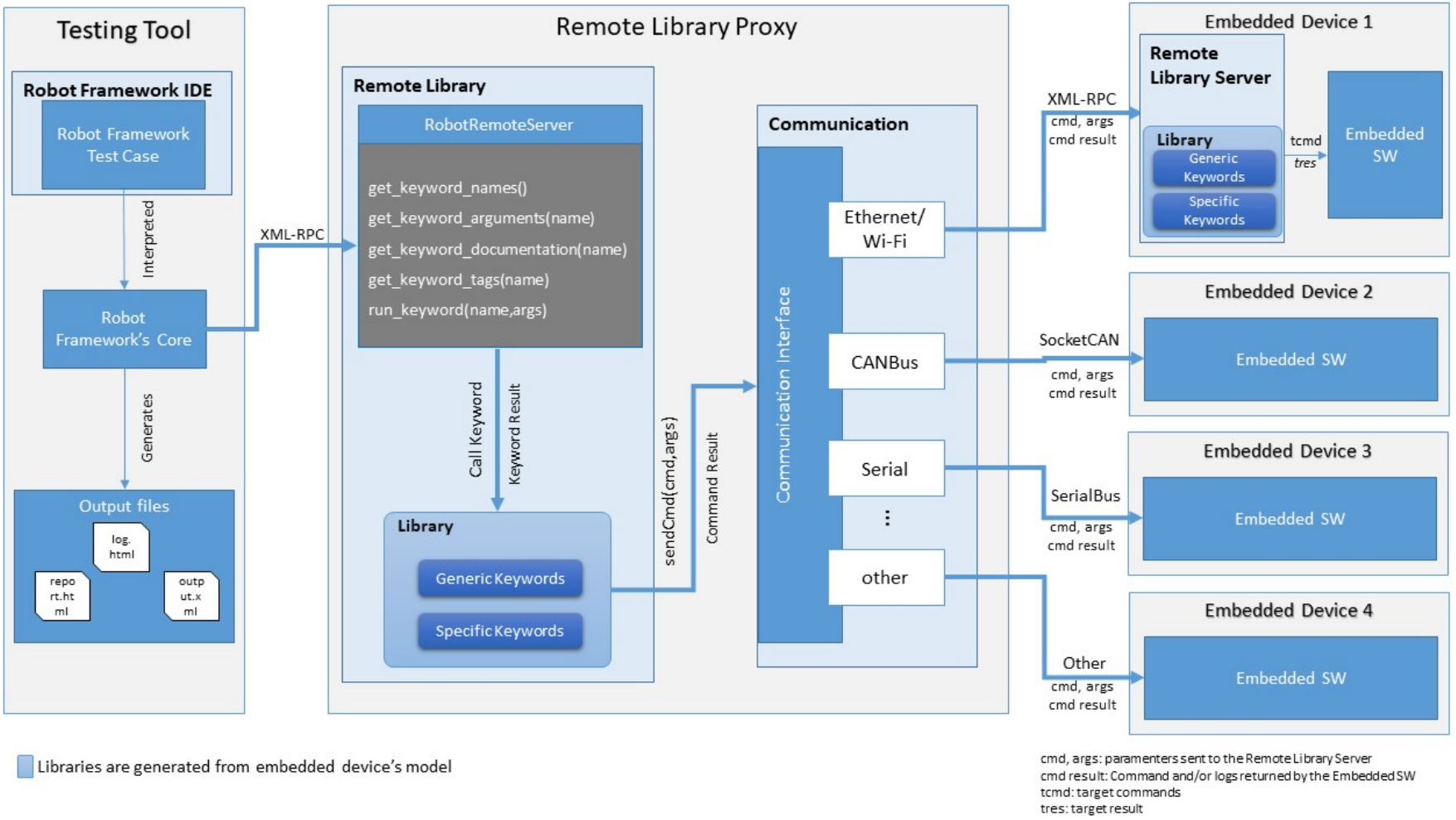


Figure 6.1: TAFES architecture

6.1 Testing Tool

The **Testing Tool** component is in charge of the interaction between the tester and the framework. It is in this component where the tester specifies the test cases, that will be interpreted by the specification editor and executed by the test driver. After a test case's execution, a test report is generated automatically stating if the test case failed or passed and the various outputs generated in each test step.

6.1.1 Robot Framework IDE (RIDE)

In this section, we explore Robot Framework IDE and within its various features, the one that we highlight is keyword completion, that allows search by keywords when typing into a test case.

Installation Robot Framework is totally independent of RIDE, meaning that in order to use this IDE we must install it externally from the Robot Framework. To run RIDE it is required Python with the minimum version of 2.6. Moreover, RIDE's GUI is implemented with wxPython toolkit, only version 2.8.12.1 is officially supported, however, it might work fine in other versions. RIDE is only used to edit test cases, therefore installing Robot Framework is required to run the created tests.[48]

Having completed and checked all the prerequisites, RIDE is easily installed using a manager for Python modules named pip. Upon the installation, RIDE can be started and Figure 6.2 show the result after starting up RIDE.

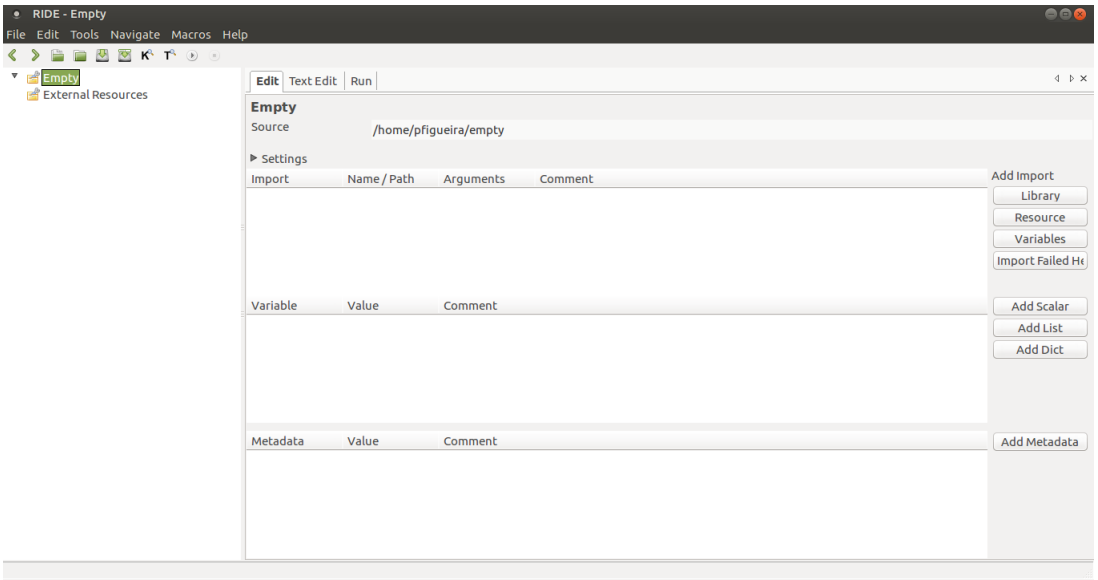


Figure 6.2: Ride startup

Test Case Specification Test specification is a detailed summary of what scenarios will be tested, how they will be tested, how often they will be tested, all this for a given feature.

Moreover, test specification gives the purpose of a specific test, identification of required inputs and expected results, step-by-step procedures for executing the test, and outline the pass/fail criteria for determining acceptance. [49]

A test plan is a collection of all test specification, it contains a high-level overview of what is tested, detailing the objectives, resources, and processes for testing. It normally contains a detailed understanding of the test workflow.

With Robot Framework IDE is it possible to specify multiple tests and later execute them. However, robot framework requires test suites which are collections of test cases, to show that it has some specified set of behaviors. Additionally, it is possible to organize these test suites in directories. For example, a system might have a smoke test[1] suite that consists only of smoke tests.

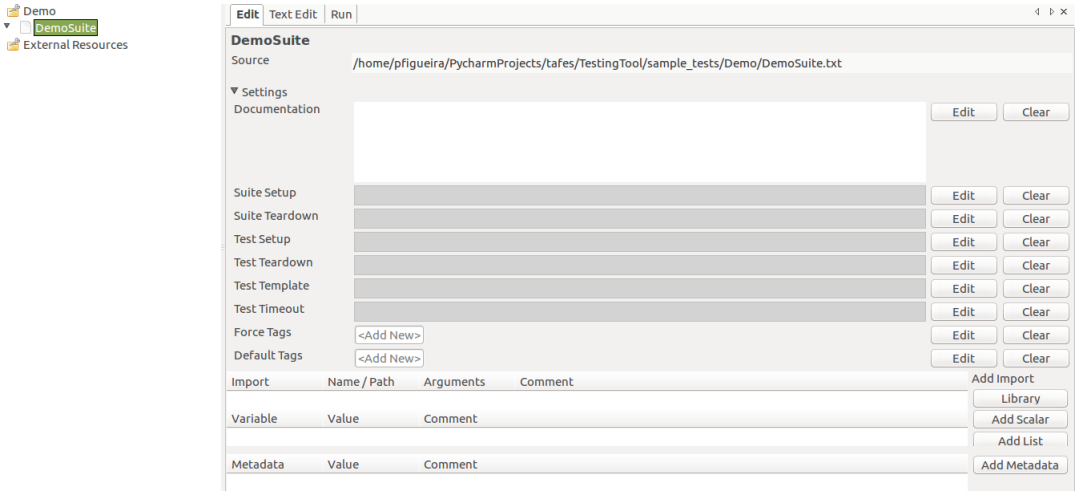


Figure 6.3: Ride suite settings

Figure 6.3 is a representation of a new test suite, where it is possible to customize the test suite, for example, setting a documentation, suite setup which is executed before the first test case execution, suite teardown which is executed after the last test case execution, it is also possible to import libraries and variables.

Having a test suite created, the next step is indeed to create test cases. Once again, RIDE is a perfect tool to work with robot framework, thus the creation of a test case is pretty straightforward.

As shown in Figure 6.4, it is possible to distinguish two features:

- Settings - Similarly to the creation of a test suite, it has a settings section where it is possible to set some options on the test case.

- Table - Robot Framework works as a tabular syntax, thus it is in this table where the test steps are specified using available keywords.

[1] Test that usually is executed before the commencement of more rigorous tests. The goal is to verify if SUT's main features are working properly.

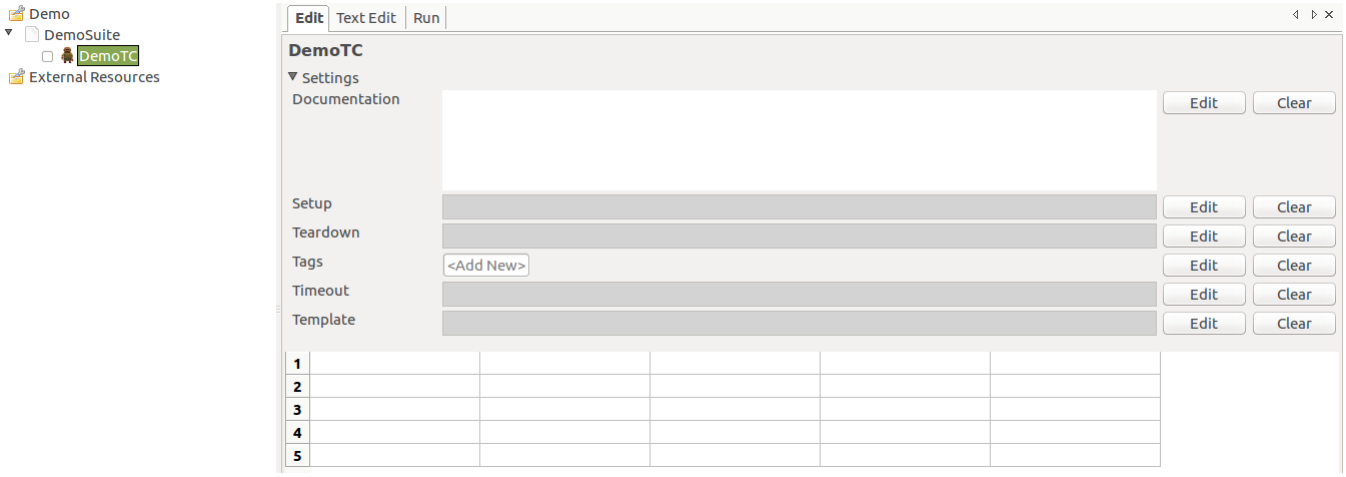


Figure 6.4: Blank test case

Keyword completion As previously described, one of the main perks of using RIDE is its efficiency and time consumption by using features like keyword auto-completion. When using this feature, RIDE user will be confronted with a list of available completions from imported Test Libraries or resource files.

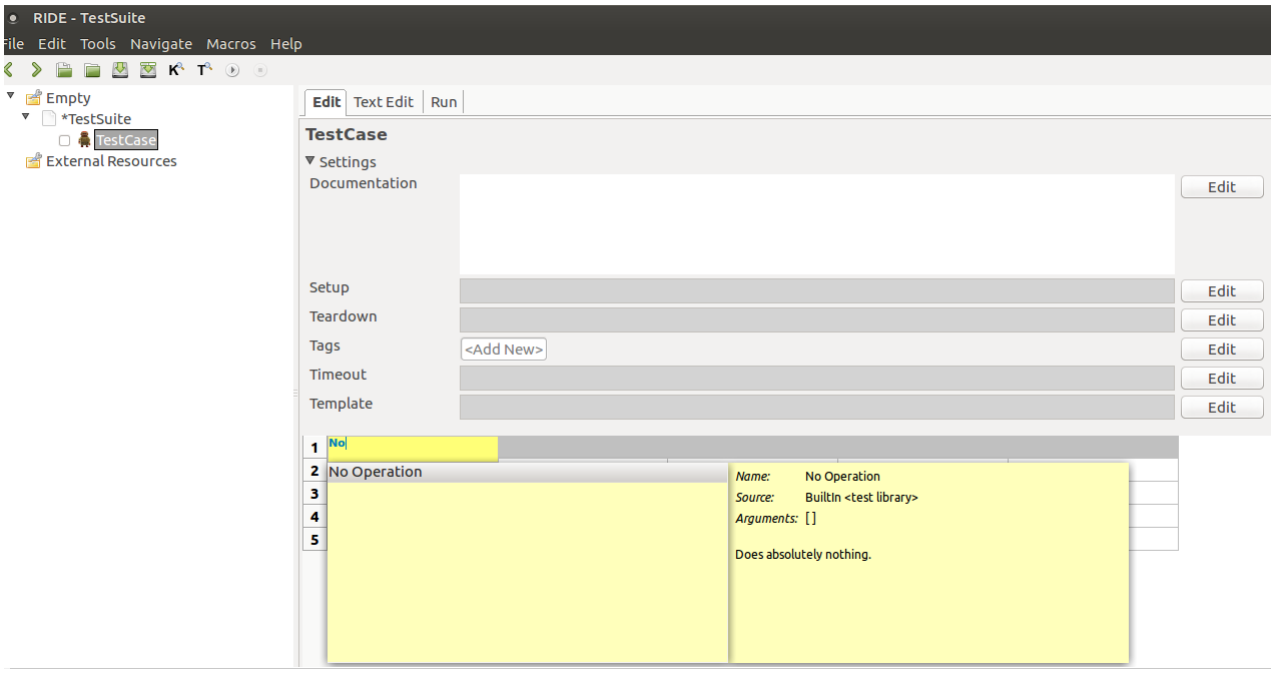


Figure 6.5: Keywords completion

Figure 6.5 shows us an example of keyword completion feature. In this example, the keyword expected is from the BuiltIn library of Robot Framework named: No Operation. Therefore, by typing "No" and then activating the auto-completion feature the suggestion box appears showing us the keyword as well as its arguments and documentation.

Obviously, if this feature was only available to Robot Framework's core libraries it

would have no use for our solution as we want to develop new test libraries, thus, it is also possible to use keyword completion for imported libraries or resources.

With all these features, the creation of test cases is efficient as it provides all the information that the tester needs regarding every keyword, and alerts for possible errors involving test steps.

Test execution RIDE allows adding additional functionality via plugins, several plugins are included in the basic installation and others may be available from other sources. One included plugin is the test runner plugin, which runs robot directly within RIDE. It supports multiple profiles, for instance *pybot*[2] and *jybot*[3] profiles come with the plugin. Moreover, with this plugin, it is possible to run an entire test suite or a subset by selecting a check-box next to individuals tests. Another approach is tagging test cases, which is a feature of Robot Framework, and then identify which tags are eligible to be executed in the *pybot* script command with *–include-tags* or *–exclude-tags* options.

Another feature of this plugin is the execution of profiles, as previously described it is possible to execute a test in Python or Jython with it respective profile (pybot,jybot). Additionally, exists an option of creating a custom profile. This is valuable as it allows to write a script and for example, use it if a test depends on other tools.

Executing a test to our framework only requires the usage of pybot, which is a Robot Framework command that interprets the test case and calls the test library in the Remote Library Proxy component and its keywords.

6.2 Remote Library Proxy - External Component

The testing tool is responsible for specifying test cases by writing keywords into its test steps. These keywords might be user keywords when written by the tester or belong to a keyword library. Keyword libraries can be native in the Robot Framework, for example, the BuiltIn library or custom made libraries. As our framework is intended to test embedded systems, Remote Library Proxy is designed to integrate Robot Framework's functionality and test these kind of systems.

As seen in Figure 6.1, Testing Tool, and Remote Library Proxy are totally independent. Separating these two elements allows test cases to be specified in different machines and in order to execute them it is only required to connect to the Remote Library Proxy. It also allows the capability of using multiples Remote Library Proxies within the same Testing Tool, this could be interesting when testing systems with a high complexity level. Additionally, the complexity to communicate with the embedded system is all on Remote Library Proxy and the tester does not need to take care of it, as his main responsibility is only to specify test cases and start its execution.

[2]Script to execute tests using Python
[3]Script to execute tests using Jython

Remote Library Proxy main objective is to store all possible keywords for a given embedded system, and consequently, have the capability to run them after the inspection of the test case done by Robot Framework. However, this is not an easy task because our framework has to have flexibility in communicating with the embedded system, that is because the communication towards the SUT might differ within systems. For example, one communicates via Serial and another via Ethernet (represented in Figure 6.1). Therefore, Remote Library Proxy is not only a proxy server storing the implementation of keywords, but it is also capable of communicating with the embedded system. Thus, to execute tests in the system under test successfully, Remote Library Proxy is composed of two modules, **Remote Library** and **Communication**.

6.2.1 Remote Library Module

As described in section 5.2, Robot Framework supports a powerful feature known as remote library interface that allows the Robot Framework's core to communicate with a remote library. In TAFES it is represented as the communication between the testing tool and the remote library of Remote Library Proxy.

A Robot Framework's remote library has to follow a set of rules (described previously in section 5.2), however, as we are developing in the programming language Python, Robot Framework has already developed a remote library server for this technology.[50]

Server configuration The python remote server for Robot Framework is configurable, meaning that it accepts various configuration parameters when initialized:

- library - Mandatory argument that corresponds to a test library instance or module with the intent of hosting and ultimately let the tester call its keywords.
- host - Represents the address on which the server is waiting for commands from the Robot Framework's core. Using "0.0.0.0" listens to all available interfaces.
- port - Here is designed the port that Remote Library Proxy is on. If given a value of "0", a free port is automatically selected.
- serve - This argument is a boolean where if set as "True" starts the server automatically and waits for it to be stopped when generated. Otherwise, the server instance is created and only will be up if a specific method is called.
- allow_remote_stop - Boolean argument if set as "True" allows the server being stopped remotely through the keyword *Stop Remote Server* and XML-RPC method *stop_remote_server*, if set otherwise disallow this functionality. We set this argument with the positive value, thus allowing the server to be stopped through a keyword.

Test Library The server configuration (section 6.2.1), is composed by multiple arguments. Among them exists *library*, which is a test library written in Python, the remote server accepts this test library (from a module or instance) by wrapping it and consequently allowing different machines to access that library via the protocol defined in 5.2.

In TAFES, the remote library is organized by two groups of keywords:

- Generic Keywords - These keywords are common among embedded systems. They are executable independently what is the system under test.

- Specific Keywords - Keywords that are specific to an embedded system. Thus, these keywords are only created for a certain system and consequently, when testing another system they have to be recreated. The creation of this kind of keywords is automatically from a system under test model specifying which are the points of observation.

6.2.2 Communication Module

When running a test case in TAFES, the end goal is to test its functionality by executing commands into the system and ensuring that it behaves accordingly. So, a vital part of the testing is the communication between the test driver (Robot Framework) and the embedded system. Consequently, if any error exists in this section, the testing of the SUT is not trustful and system errors might arise in the future.

TAFES is designed to perform tests automatically in any kind of embedded systems. Knowing that the method of communication in systems differs, it must have a communication module capable of communicating with multiple protocols. In our solution we have three methods of communicating to the embedded device, allowing TAFES to be executed in a wide variety of systems.

- **Ethernet/Wi-Fi** - In the recent years, we have seen an increase in the number of devices connected to a network by either Ethernet or Wi-Fi. Therefore, having TAFES prepared to communicate over these protocols is allowing our test platform for a broad number of devices.

  We selected the XML-RPC as the communication protocol between the Remote Library Proxy and the embedded device. As it is a remote procedure call it is required to have a RPC-Client and a RPC-Server, both with the exposed methods that ultimately represent keywords for the tester. In this context, the RPC-Client is the Remote Library Proxy and the RPC-Server is the Embedded Device. However, in order to have the Embedded Device acting like a RPC-Server is required additional software in its system (section 6.3.1.

- **Serial** - Serial communication is widely used in embedded devices, for example in computer peripherals or in printers. For that reason, we have chosen to have this communication method in TAFES.

  In serial communication the data is transferred in a sequential manner and in the form of binary pulses, this communication can take different modes such as:[51]:

  - Simplex - One-way communication technique, a node acts either as a sender or as a receiver. If a sender transmits the receiver can only accept.
  - Half Duplex - A node can be both sender and receiver but cannot be both at the same time, for example, if a sender transmits, the receiver can accept but cannot send and vice versa. An example of a half-duplex mode is when a browser sends requests for a web page, then the web server processes the application and sends back information.
  - Full Duplex - In this mode, a node can act as sender or receiver at the same time.

  Given the fact that when executing test steps into the embedded device it is expected some sort of output from the SUT, thus we require a Half or Full Duplex serial communication.

- **CANBus** - The last implemented communication method is CANBus, but it follows a different approach than the other two communication methods.

  When communicating with CANBus we broadcast a message through a BUS with a certain priority number that will affect the CANBus nodes that it is intended to affect. Through this process, it is not guaranteed that the SUT interpreted the corresponding message. Thus, in TAFES when using CANBus a message is sent to the BUS but there is no response like when using XML-RPC or Serial. Therefore, when executing a test step a message is sent and is assumed that it arrived at the SUT and the device acts accordingly to the message received.

  It is also assumed that the embedded device broadcast messages that represent its state, therefore, Remote Library Proxy communication module has to have the capability to read broadcast messages from the SUT.

6.3 Embedded Device

After describing the **Testing Tool** the and **Remote Library Proxy**, the next component is the embedded device that ultimately acts as a system under test. However, an embedded device is truly eligible to be a system under test in TAFES if it meets these conditions:

- It exposes an interface that is capable of connecting to the **Remote Library Proxy**.

- Has a model representation of all of its accessible operations and consequently follows a Model Driven Development.

Probably when analyzing the conditions one might think that it is too strict and it is against the motivation of having a generic framework capable of testing any embedded system, fortunately , the **Remote Library Proxy** component is constructed to have the most popular interfaces in the embedded context (Ethernet, Serial and CANBus), thus this requirement is easily achieved by an embedded system that shares one of these interfaces. Sometimes this interface can not exactly correlate to the systems operations, for that specific case TAFES has a special component that is implemented in the device with the objective of allowing an external component to execute commands. This special component is called **Remote Library Server** and is described in the section 6.3.1.

The second condition is the most restrictive as we cannot ensure that a device follows Model Driven Development, however, this approach has many benefits and is increasing in popularity among the embedded system development professionals [36]. Model Driven Development and automatic code generation are highly related and to improve the testing process efficiency, TAFES also generates test libraries through analyzing a model of the embedded device. Section 6.3.2 explains the process to automatically generate test libraries for a given embedded device.

6.3.1 Remote Library Server

Even if the system has an interface to communicate there might not exist a relationship between the communication interface and the system interface. To overcome this problem a special component of TAFES exists called **Remote Library Server**. As the name suggests it works as a server, and much like **Remote Library Proxy**, it also has a storage of the multiple operations that the system executes. With the generic and specific keywords that are generated automatically (section 6.3.2).Thus, it is possible to communicate between the Remote Library Proxy (RLP) and Remote Library Server (RLS) with remote procedure calls as the first works as a client and the second as a server that interacts directly with the software. This interaction is possible by the points of contact of the embedded software, for example in the .so files (shared libraries) when developing a software with the C language.

Given its characteristics, Remote Library Server is ideally used in a more complex embedded system that supports programming languages like Python and C. Regarding any other systems that do not support these languages they must have Serial, or CANBus communication interface to be tested with TAFES.

6.3.2 Library generator

In embedded systems, developers use traditional programming languages such as C and C++ and make use of the processes and techniques inherent by these languages to

improve reliability and reduce security flaws. However, model-driven development is another approach which can achieve this objective.

Model-driven development uses models as primary artefacts throughout the system engineering life cycle, elevating functional models to a central role in the specification, design, integration, and validation of software.[52]

It is important to model in an unambiguous way that can be understood by the domain experts that determine what the resulting system does as well as the software developers that implement the system. The major advantage is that models are expressed using concepts that have a lower bound to the underlying implementation technology [37], making easier to specify, understand, maintain and ultimately reduces flaws that might lead to system failures.

To design these models it is used a modeling language with a well-defined grammar and semantics, there are two groups of languages:[52]

- Vendor-specific languages - Languages developed and promoted by a specific vendor of a model-driven development platform, eg. MatLab and Simulink from MathWorks.
- Standardized languages - These languages are defined by industry groups of interested users and model-driven development platform vendors, most commonly based on the Unified Modeling Language(UML).

Between vendor-specific and standardized languages, the preferred one is the second as it is vendor-independent which offers the possibility for models to be exchanged between platforms. Thus, enabling to model in a broader set of systems than vendor-specific languages.

Model-driven development has the benefit of increasing the degree of automation applied to the development by generating code automatically, this indeed is one of the most powerful characteristics which assures correspondence between specification and implementation. However, it can only be achieved if the generation process from model to code is running flawlessly and models are previously validated.

TAFES takes advantage of this feature of model-driven development to generate automatically its test libraries. This process massively increases the efficiency of testing the embedded system because all the relation between the system's operation and the communication is automatically generated, leaving only the creation of the test cases to the tester.

Embedded System's Model As previously explained, the preferred language to model is UML, thus, to allow TAFES to interpret the model, it is required some annotations.

So, to have a viable model for library generation, it has to follow these steps:

1. First the supported communication method has to be explicitly defined by annotating the model with a specific tag that starts with the prefix **"#TAFES_RobotLib"**

followed by the communication method.**"#TAFES_RobotLibSERIAL"** is an example of an annotation when the embedded system allows a communication via Serial.

2. Upon annotating the whole model with the communication method, all of the interface's elements are examined and its operations are aggregated and form the test library used in the Remote Library Proxy and Remote Library Server. A specific method of communication like CANBus and Serial needs an extra annotation on the operation itself. For example, it is required the CAN ID and DATA that activates a given operation.

Configure TAFES for the embedded device To fully generate the libraries it is required some to fill some configuration files with information regarding the method of communication applied to the embedded system. By writing these files, the Library Generator not only will generate the keyword but also their content, translating it to the communication mechanism of Remote Library Proxy.

Therefore, the content required in the configuration for the given method of communication used are:

- CANbus - For communicating in CANBus we are using the Python library python-can[47]. The requirements needed to communicate via CAN are:
  - *INTERFACE* - Python-can's interface to use in the Remote Library Proxy, it has to be according with the controller area network adapter used by the hardware that the RLP is deployed at.
  - *CHANNEL* - Can interface name with which Remote Library Proxy connects to.
  - *BITRATE* - Channel's bit rate.
  - *RLP_HOST_IP* - The host and IP used to communicate with the Remote Library Proxy, it has to follow the syntax : *"hostip"*.
- Serial - For serial communication it is used the Python library pySerial[53] requiring the following configuration parameters:
  - *PORT* - Device name defined in the Remote Library Proxy.
  - *BAUDRATE* - Baud rate of the serial communication.
  - *TIMEOUT* - Read timeout.
  - *WRITE_TIMEOUT* - Write timeout.
  - *MESSAGE_START_MARKER* - To communicate with the embedded device it is required markers that define a message, this define the message start marker.
  - *MESSAGE_END_MARKER* - End of message marker.

  - *MESSAGE_CONTENT_SEPARATOR* - Marker that separated the message content, it is used when multiple values are passed in a message.
  - *RLP_HOST_IP* - The host and IP used to communicate with the Remote Library Proxy, it has to follow the syntax : *"hostip"*.

- XML-RPC - Regarding the XML-RPC protocol communication(through Ethernet/Wi-Fi) it is required the following configuration:
  - *SERVER_ADDRESS* - The Remote Library Server http server address.
  - *RLP_HOST_IP* - The host and IP used to communicate with the Remote Library Proxy, it has to follow the syntax : *"hostip"*.
  - *RLS_HOST_IP* - Host and IP to communicate with the Remote Library Server. Follows the syntax : *"hostip"*.

Figure 6.6 illustrates the process of generating test libraries automatically. It starts with the definition of a Model with annotations regarding the communication method supported by the embedded device. Additionally, it is required a configuration file that will configure the communication to the embedded device.

Then the Library Generator mechanism will have both of these files as input and will translate the operations set in UML format to keywords in Robot Framework. The content of these keywords is to call the communication module of RLP. For this reason, the configuration file is required, which allows opening a connection to the embedded device.

After creating the test library, the next step of Library Generator is to deploy this library in the corresponding components. Given that RLP is into a Raspberry Pi, it is used ssh in the communication between the Library Generator and the RLP. After storing the library, the remote server is updated. With an updated version of RLP, the testing tool is capable of discovering which new keywords are available, and consequently execute them.

6.4 Summary

In this chapter, we presented the architecture of our solution and restrictions to perform test automation in an embedded system. It is composed of three components that are totally independent and deployed in different systems. The testing tool component is located in the tester's computer, the Remote Library Proxy in a portable system (Raspberry Pi) and then the last component is the Embedded System itself.

First and foremost an embedded system is eligible to be tested with our solution if it has a model that exposes all its operations and if its communication method is one that TAFES supports (Ethernet/Wi-Fi, Serial, CANBus). Therefore, the generation of test scripts (Library Generator) requires two inputs: An UML file with all the corresponding

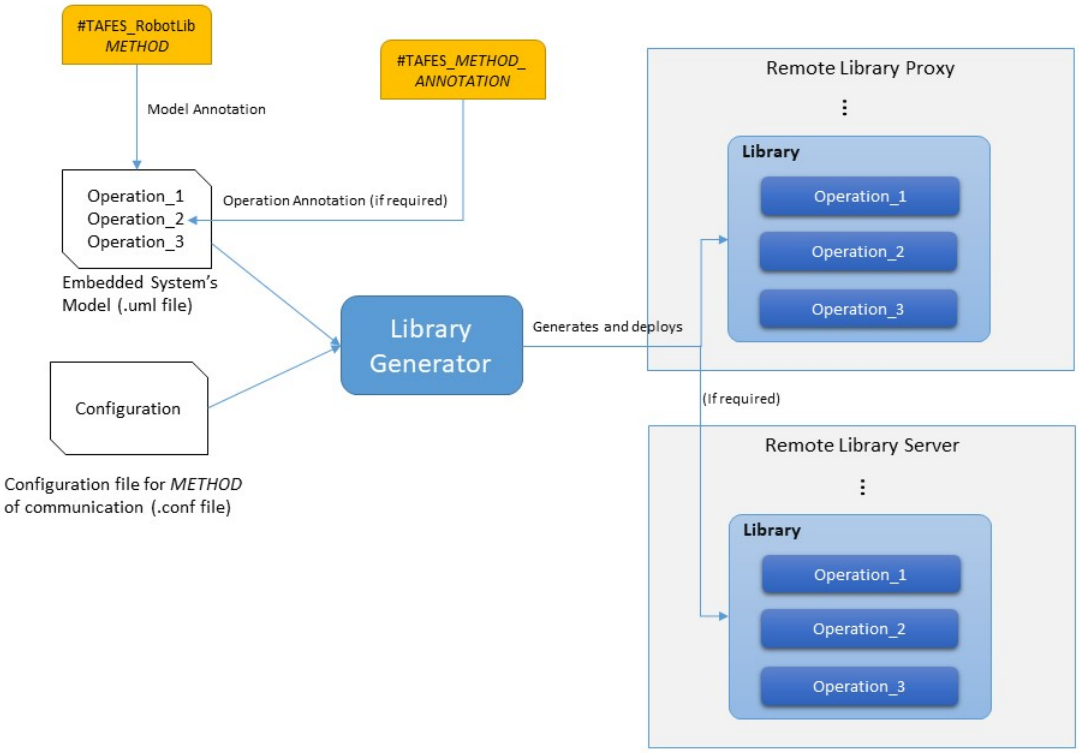


Figure 6.6: Generation of test libraries

annotations required to that method of communication; and a configuration file to set up the communication between components (mainly Remote Library Proxy and Embedded Device). As output, library generator automatically generates and saves the test library in the Robot Framework style into the corresponding components (Remote Library Proxy and in case of an Ethernet/Wi-Fi communication also in Remote Library Server). After storing the test library, the components then are accessed to the outside be starting a server that exposes the test library.

At this stage, the Remote Library Proxy contains all the operations that are possible to execute in the embedded device stored as test library. Thus, to access these operations it is required the Testing Tool component. In this component, the test case in specified and it imports the test library located in the RLP. Robot Framework has an IDE that turns the whole test specification process more efficient, where one of its key features is the auto-completion of keywords.

Finally, with the test specified it is possible to run it through Robot Framework. By doing so, it will generate output files that report the test cases' execution. In these files, it is possible to understand if the test cases have passed or failed, as well as analyze its behavior through the logging file.

Chapter 7

Use Cases

Having selected a test automation framework to base TAFES on, and already designed its architecture the next step is to analyse its functionality and ultimately use our solution to test an embedded system. Thus, in section 7.1 we deploy and setup the Remote Library Proxy into a hardware device (Raspberry Pi). Then we utilize it to test a proximity sensor, first a simulated version where it exposes communication via Ethernet/Wi-Fi (section 7.2), then with a real-world sensor that exposes operations via Serial communication (section 7.3).

7.1 Setup of Remote Library Proxy

Remote Library Proxy is a vital component for our solution has it stores the test library and sends instructions to the embedded device. Acting like a proxy server, it is required to be assigned to some kind of hardware. Given that the main property of our framework is to test a wide variety of embedded systems, we have selected to deploy the Remote Library Proxy into a **Raspberry Pi 3B+**.

Being deployed in a Raspberry Pi allows adaptability to different embedded devices interfaces, since it support Ethernet/Wi-Fi and Serial(USB) communication. Regarding CANBus communication, it is required extra hardware. Thus, deploying Remote Library Proxy in a Raspberry Pi not only allows flexibility in testing embedded devices, as well as portability. The Raspberry Pi has the Raspbian Stretch Lite operating system and the following files and folders stored:

- **/RemoteLibraryProxy/** - In this folder, it is located the code for the Remote Library and the communication to the embedded device:
  - **rlp.py** - Python module that contains a implementation of remote server from Robot Framework and the test library.

  - **specificKWProxyCan.py** - Implementation of keywords to CANBus communication embedded device.
  - **specificKWProxySerial.py** - Implementation of keywords to Serial communication embedded device.
  - **specificKWProxyXmlRpc.py** - Implementation of keywords to Ethernet/Wi-Fi communication embedded device, this represents the XML-RPC client.
  - **Utils/canutilities.py** - Python module that implements communication via CANBus with the Python library *python-can*.
  - **Utils/serialutilities.py** - Module with the implementation of communication via Serial using *pySerial* library.
  - **Utils/xmlrpcutilities.py** - Implementation of an XML-RPC client for communication with the XML-RPC server located in the embedded device.

- **/RLS Files/** - This folder contains files that have to be propagated to the embedded device when the communication is through XML-RPC:
  - **specificKWServerXmlRpc.py** - File corresponding the Remote Library Server on the embedded device, capable of listening and translating commands to embedded device functionalities.

- **scp_script.sh** - This is a script that propagates the content on the folder "RLS FILES" to the embedded device.

- **rlp_restart.sh** - Script that automatically restarts and updates Remote Library Proxy.

7.2 Simulated proximity sensor with Ethernet/Wi-Fi communication

In this section, we will walk through the process of testing in an embedded system that supports communication via Ethernet or Wi-Fi, being this method of communication the interaction to the embedded device is through XML-RPC protocol (section 6.2.2).

Nevertheless, it was not possible to acquire a device capable of exposing a connection via Ethernet or Wi-Fi, therefore we have simulated such device. The simulated proximity sensor is a program that:

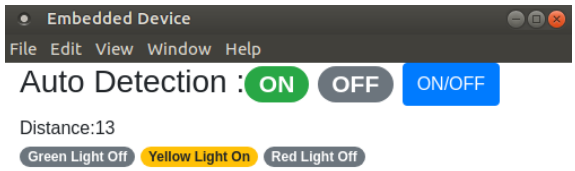
- Outputs the distance and activates a, exists three lights and each light is activated on when the detected distance belongs to one of the following ranges:
  - *Red light* - [0, 10[.
  - *Yellow light* - [10, 20[.

  - *Green light* - [20,∞[.

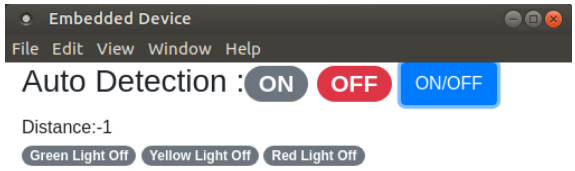
- Has a button that switches the sensor state:

  - *ON* - Sensor can detect objects and consequently output distance and consequently turning on the corresponding light.
  - *OFF* - Sensor can no longer detect objects, distance takes a negative value and lights are all off.

Figure 7.1 represents the embedded device's states, where in 7.1a the device is *ON* and detecting a object, and in 7.1b it is *OFF*, therefore, not detecting anything.



(a) *ON*



(b) *OFF*

Figure 7.1: Detection mode *ON* and detection mode *OFF* in simulated proximity sensor

To execute TAFES on the embedded device it is required to have a model that illustrates the operations available to the "outside". It was used the Papyrus Eclipse plug-in to design the respective model (Figure 7.2).

As described in section 6.3.2, models requires labelling, as this is a device that allows Ethernet/Wi-Fi communication, the data is transmitted through the XML-RPC protocol, therefore, the model has the following label "**#TAFES_RobotLibXMLRPC**" (Figute 7.3). After successfully design and annotate the model, the next step is to open the configuration file for the respective communication method (XML-RPC) and fill it with the information regarding the TAFES components following the topics in section 6.3.2. Figure 7.4 illustrates the end result of the configuration file.

Papyrus plug-in produces an XML representation of the designed model. It can easily be interpreted and that is exactly what the Library Generator does (section 6.3.2), then generates the test library and sends it to the Remote Library Proxy and Remote Library Server.

Figure 7.5 is a sample of the generated library for the Remote Library Proxy where we can find a Python class where the methods correspond the operations defined in the

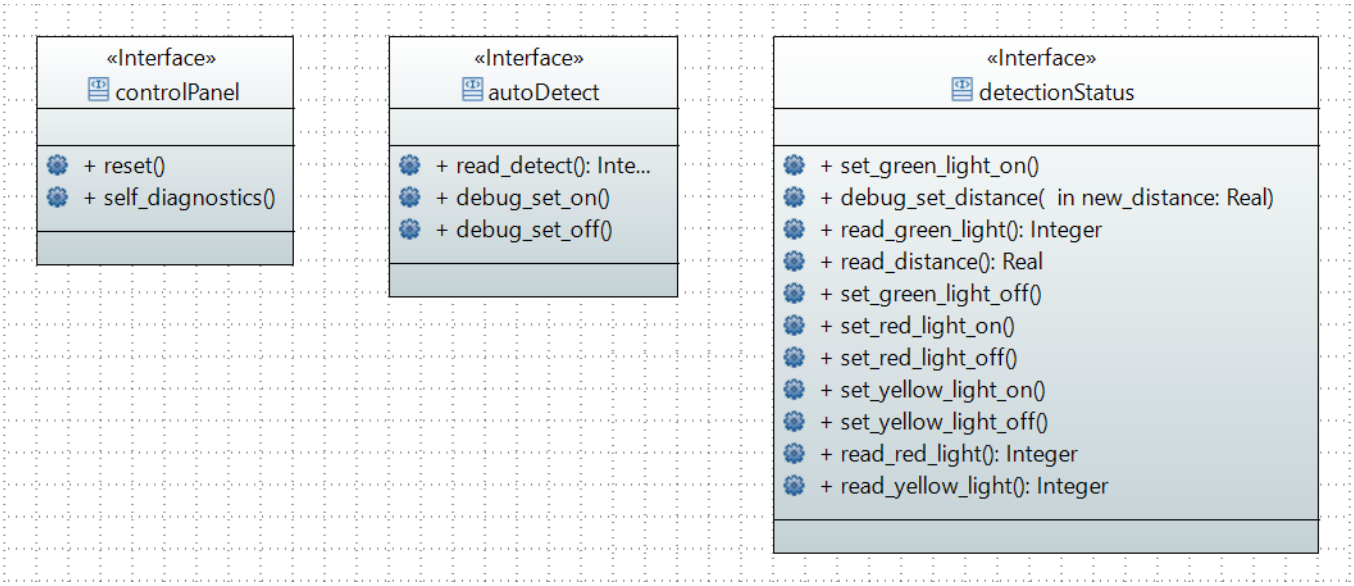


Figure 7.2: XML-RPC embedded device model

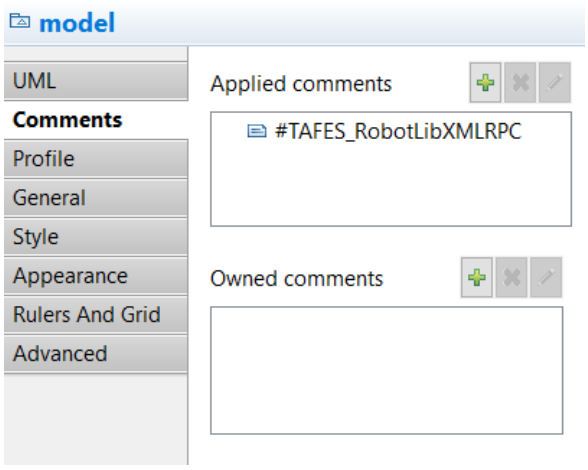


Figure 7.3: Model annotation

xml-rpc.conf 111 Bytes

```
1  [DEFAULT]
2  SERVER_ADDRESS = http://10.0.2.15:8271
3  RLP_HOST_IP = pi@10.12.15.50
4  RLS_HOST_IP = pfigueira@10.0.2.15
```

Figure 7.4: Configuration file for XML-RPC

specificKWProxyXmlRpc.py 2.16 KB

```
from Utils import xmlrpcutilities

class XmlRpcLibrary(object):

        def __init__(self):
                super(XmlRpcLibrary, self).__init__()
                self.client = xmlrpcutilities.connect_to_server("http://10.0.2.15:8271")

        def autoDetect_debug_set_off(self):
                log = self.client.autoDetect_debug_set_off()
                print(log)

        def autoDetect_read_detect(self):
                log, Integer__detect = self.client.autoDetect_read_detect()
                print(log)
                return Integer__detect

        def autoDetect_debug_set_on(self):
                log = self.client.autoDetect_debug_set_on()
                print(log)

        def controlPanel_self_diagnostics(self):
                log = self.client.controlPanel_self_diagnostics()
                print(log)

        def controlPanel_reset(self):
                log = self.client.controlPanel_reset()
                print(log)

        def detectionStatus_read_distance(self):
                log, Float__distance = self.client.detectionStatus_read_distance()
                print(log)
                return Float  distance
```

Figure 7.5: Remote Library Proxy's keyword library

model. Moreover, each method sends command that the embedded device will interpret and consequently execute. Additionally, a connection to the server is open where the address is the one defined in the configuration file.

The embedded device that we are testing only supports communication through Ethernet/Wi-Fi, therefore, the messages between to the device are sent via XML-RPC. As we have seen before this method of communication requires an additional software in the device that acts like a RPC-Server. This server, that is represented by the Remote Library Server shares the same methods as the RPC-Client, represented by the Remote Library Proxy. However, in the Remote Library Server the implementation of these methods differs, as it now calls direct operations of the embedded software,.

In this use case, the embedded device was developed in C language, thus exposing its operation through shared objects that can later be accessed by the keyword library of Remote Library Server using the Python's library ctypes (Figure 7.6.

```
class XmlRpcLibrary(object):

        def __init__(self):
                xmlrpcutilities.create_server("http://10.0.2.15:8271",self)

        def autoDetect_debug_set_off(self):
                lib = getlib("autoDetect")
                lib.debug_set_off.argtypes = []
                lib.debug_set_off()
                return log("autoDetect_debug_set_off", )

        def autoDetect_read_detect(self):
                lib = getlib("autoDetect")
                lib.read_detect.argtypes = []
                lib.read_detect.restype = c_int
                Integer__detect = lib.read_detect()
                return log("autoDetect_read_detect",), Integer__detect

        def autoDetect_debug_set_on(self):
                lib = getlib("autoDetect")
                lib.debug_set_on.argtypes = []
                lib.debug_set_on()
                return log("autoDetect_debug_set_on", )

        def controlPanel_self_diagnostics(self):
                lib = getlib("controlPanel")
                lib.self_diagnostics.argtypes = []
                lib.self_diagnostics()
                return log("controlPanel_self_diagnostics", )

        def controlPanel_reset(self):
                lib = getlib("controlPanel")
                lib.reset.argtypes = []
                lib.reset()
                return log("controlPanel_reset", )
```

Figure 7.6: Remote Library Server's keyword library

Now that the model has been interpreted and transformed into keyword libraries for the Remote Library Proxy and the Remote Libary Server, the embedded system is viable for testing. That is the purpose of Testing Tool component (Section 6.1), it uses Robot Framework IDE (RIDE) to create and run test cases. For this use case we have the following test cases:

- Set auto detection *OFF* and verify if the lights are off and detected distance is -1.
- Set auto detection *ON* and verify if the distance is positive.
- Set a detection distance and verify if the corresponding light is on and the others off.

Test Case 1 - Set auto detection OFF As described previously in this test case we look for setting off the detection of the embedded system and then ensure that the distance reads negative values and the light are all off. In the model, the operation that controls the auto-detection is in the interface *AutoDetect*. Thus the keyword for setting auto detection *OFF* is *AutoDetect Debug Set Off*, and to activate the detection the keyword is *AutoDetect Debug Set On*. In the model it is described by an operation that can return the value of the detection (1 when the detection is activated and 0 otherwise) this keyword is named *AutoDetect Read Detect*.

The keywords that correlate the light status are located in the interface *DetectionStatus*, thus the keywords to get the red, yellow and green light status are called *DetectionStatus Get Red Light*, *DetectionStatus Get Yellow Light* and *Detection Status Get Green Light* respectively.

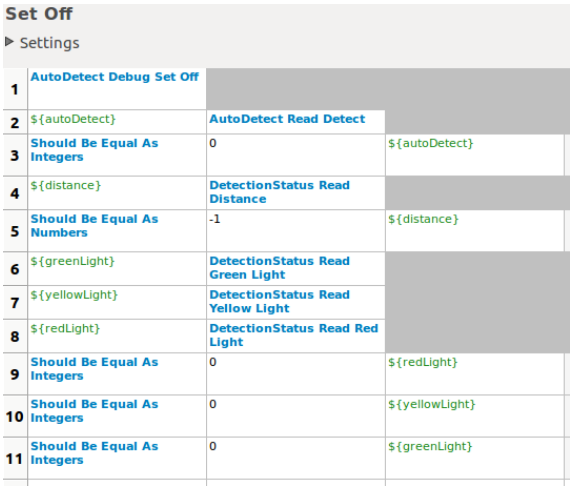
Figure 7.7a is the test case specification where we execute the set off operation and verify if the device if no longer detecting objects. Figure 7.7b is the device state after executing this test case.

Test Case 2 - Set auto detection ON This second test case is pretty similar to the previous test case, but instead of executing a command to set the detection *OFF* we execute the activation of detection. The keyword is also from the interface named *AutoDetect*, and the keyword originated is *AutoDetection Debug Set On*.

After setting the auto detection *ON*, the device detects an object on a random distance between 1 and 30. Therefore, the test case has to verify if a valid distance was detected.

Figure 7.1a is a representation of the specification of this test case in the RIDE. Figure 7.8b is the device state after executing test case 2.

Test Case 3 - Detection of an object and lights output The last test case selected has the purpose of checking if the device can correctly turn on the light of the detected distance. The embedded device's state is stored in files, thus changing these files will overwrite the state of the embedded system, another approach to change the detected

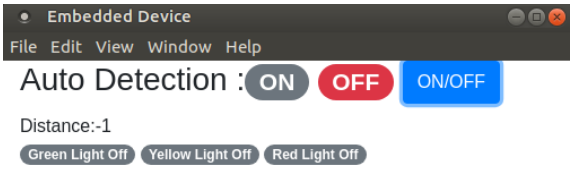


Set Off

▶ Settings

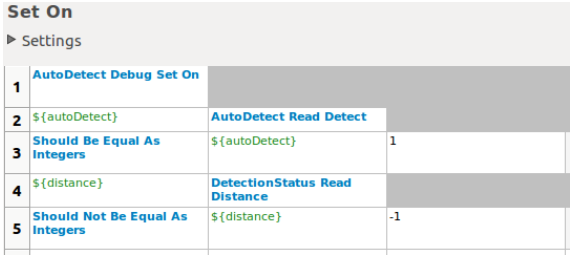
1	AutoDetect Debug Set Off		
2	${autoDetect}	AutoDetect Read Detect	
3	Should Be Equal As Integers	0	${autoDetect}
4	${distance}	DetectionStatus Read Distance	
5	Should Be Equal As Numbers	-1	${distance}
6	${greenLight}	DetectionStatus Read Green Light	
7	${yellowLight}	DetectionStatus Read Yellow Light	
8	${redLight}	DetectionStatus Read Red Light	
9	Should Be Equal As Integers	0	${redLight}
10	Should Be Equal As Integers	0	${yellowLight}
11	Should Be Equal As Integers	0	${greenLight}

(a) Test case specification



(b) Device state

Figure 7.7: Test case 1 specification and embedded device state after execution

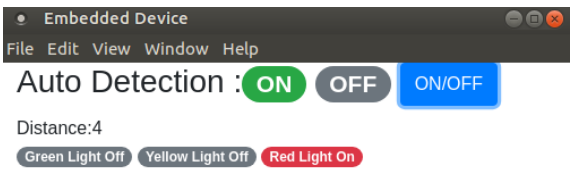


Set On

▶ Settings

1	AutoDetect Debug Set On		
2	${autoDetect}	AutoDetect Read Detect	
3	Should Be Equal As Integers	${autoDetect}	1
4	${distance}	DetectionStatus Read Distance	
5	Should Not Be Equal As Integers	${distance}	-1

(a) Test case specification



(b) Device state

Figure 7.8: Test case 2 specification and embedded device state after execution

distance is using the keyword *DetectionStatus Debug Set Distance* which have the same effect of changing the file.

When changing this distance it is expected that the light turns ON/OFF accordingly, thus to verify this we use the keyword to get the light status of each light.

In this test case, we change the distance to the following values, and it is expected only the following lights to be ON:

- 5 - Red light on.
- 10 - Yellow light on.
- 15 - Yellow light on.
- 20 - Green light on.
- 30 - Green light on.

Using Robot Framework functionality, we have created a user keyword that accepts a distance and the lights expected status. This keyword is used as a template.therefore, that in our test case we can call this keyword with different arguments. Figure 7.9 illustrates this keyword, where we call keyword *DetectionStatus Debug Set Distance* to change the detected distance. The new distance value comes as a argument of the user keyword. Moreover, the expected detection status of each light also comes from the user keyword parameter and it is used to check if it corresponds to the systems true value.

When using a user keyword as a template, the test case's specification is a set of values that act as arguments to the user keyword. Thus, for each row of the test case, the first column is the distance to detect and the following three the expected light status. Figure 7.10 is a representation of the test case specification on Robot Framework IDE, it is possible to see that user keyword *Check detection for a given distance* is a template and the test specification is the parameters for that keyword.

After executing the test cases, the report and log files from Robot Framework are generated automatically stating if the test case passed or failed and logging every test step. Appendix C.1 presents example of these files for this embedded device.

7.3 Proximity sensor with Serial communication

Based on the simulated proximity sensor (section 7.2) we have built a real-world proximity sensor that fully works as an embedded system that supports Serial communication. It was developed using the micro-controller board *Arduino UNO* and the ultrasonic sensor *HC-SR04*, Figure 7.11 represents the device connected to the Raspberry Pi that deploys the Remote Library Proxy.

The device's functionality is quite similar to the simulated version, it detects an object and depending on the distance (in centimetres) a light turns ON. The distance boundaries

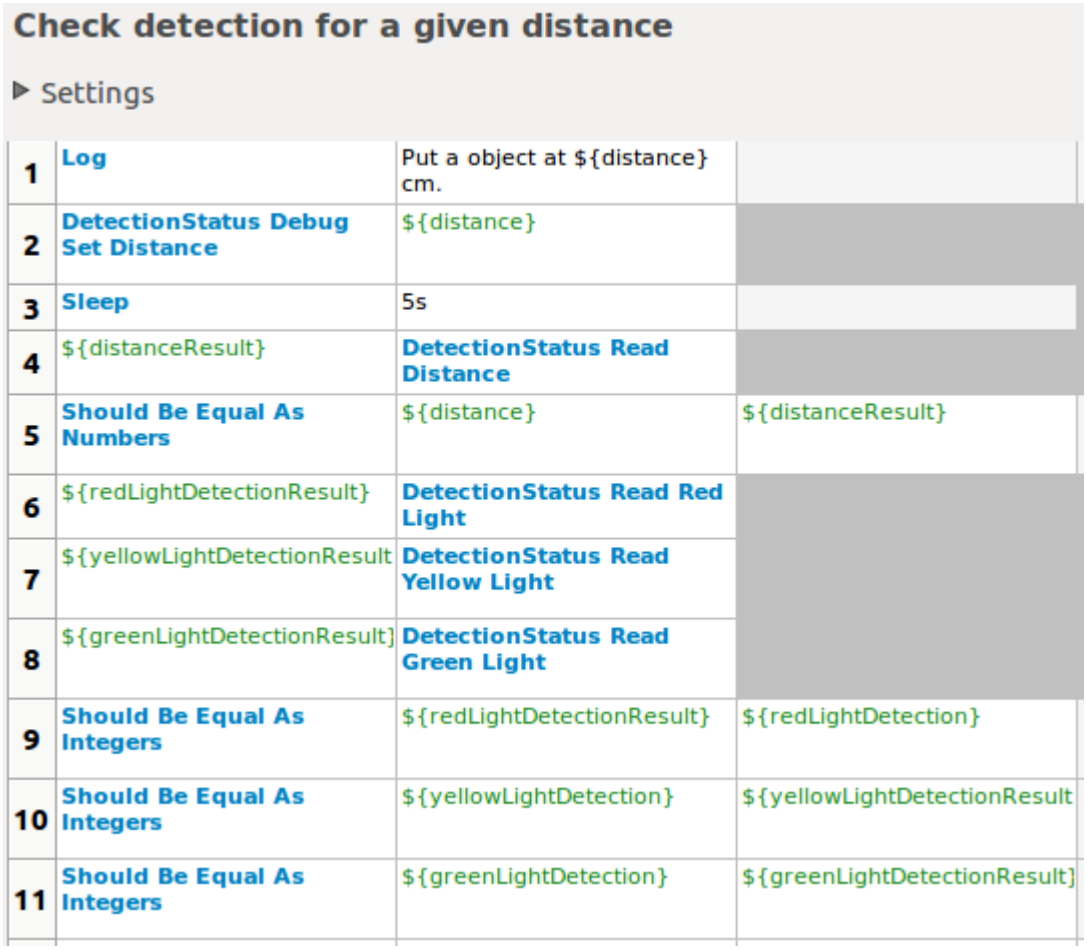


Figure 7.9: Use case XML-RPC test case number 3 - User keyword

are the same, meaning that a red light will light up if the distance is greater than zero and less than ten, yellow if greater than or equal to ten and less than twenty, and green if greater than or equal to twenty. Moreover, a blue light indicates if the detection of objects is activated or not.

Adding to the lights we have six toggles, located at the bottom right of the embedded device (Figure 7.11), which purpose is to inject bugs into the system. They are classified into two groups and have a toggle for each light:

- Detection Status toggles - With options *Working* and *Fail*. When selected as *Fail*, a bug is injected into the detection of an object, meaning that is not possible to detect in the range of the respective light, when selected as *Working* the detection of the respective light is working flawlessly.

- LED Health Status toggles - *OK* and *Not OK* are the options for these toggles. When selected as *Not OK*, a bug is injected when turning the light on.So, it stays OFF regardless if the sensor has detected an object that corresponds to that light or not. The *OK* option allows the light to be turned on.

Detection of object and LED output

▼ Settings

Documentation	
Setup	
Teardown	
Tags	<Add New>
Timeout	
Template	Check detection for a given distance

1	5	1	0	0
2	10	0	1	0
3	15	0	1	0
4	20	0	0	1
5	30	0	0	1

Figure 7.10: Use case XML-RPC test case number 3

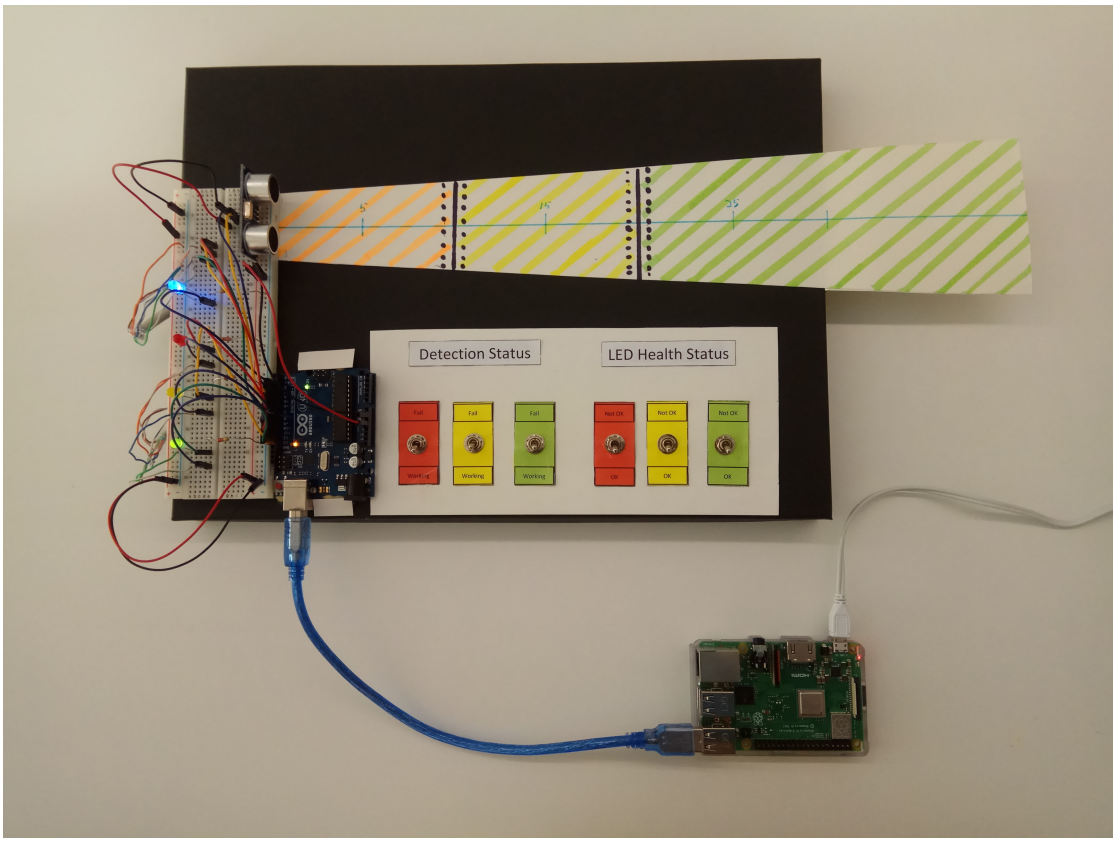


Figure 7.11: Proximity sensor with serial communication connected to the Raspberry Pi

Regarding the model for this embedded device it can be found in Figure 7.12 and it is composed by two interfaces. The first interface is *SystemControl* exposes operations that control the whole system, operations *set_on* and *set_off* activates and deactivate the detection of objects, the operation *get_auto_detection* returns the detection's status, if it is *ON* or *OFF*.

The second interface, *Detection*, exposes operations regarding the detection and the lights, it has an operation to get the detection distance (*get_distance*) and for each light there are three methods:

- get_**light colour**_light_detection : This operation returns one if it was detected an object in that light distance range, zero otherwise.
- get_**light colour**_light_detection_status : This operation correlates with the Detection Status toggles, it returns one if it is possible to detect objects within the light distance range, zero otherwise.
- get_**light colour**_light_health_status : Similar to detection status operation this is related to the LED Health Status toggles, return one if the light is health and zero otherwise.

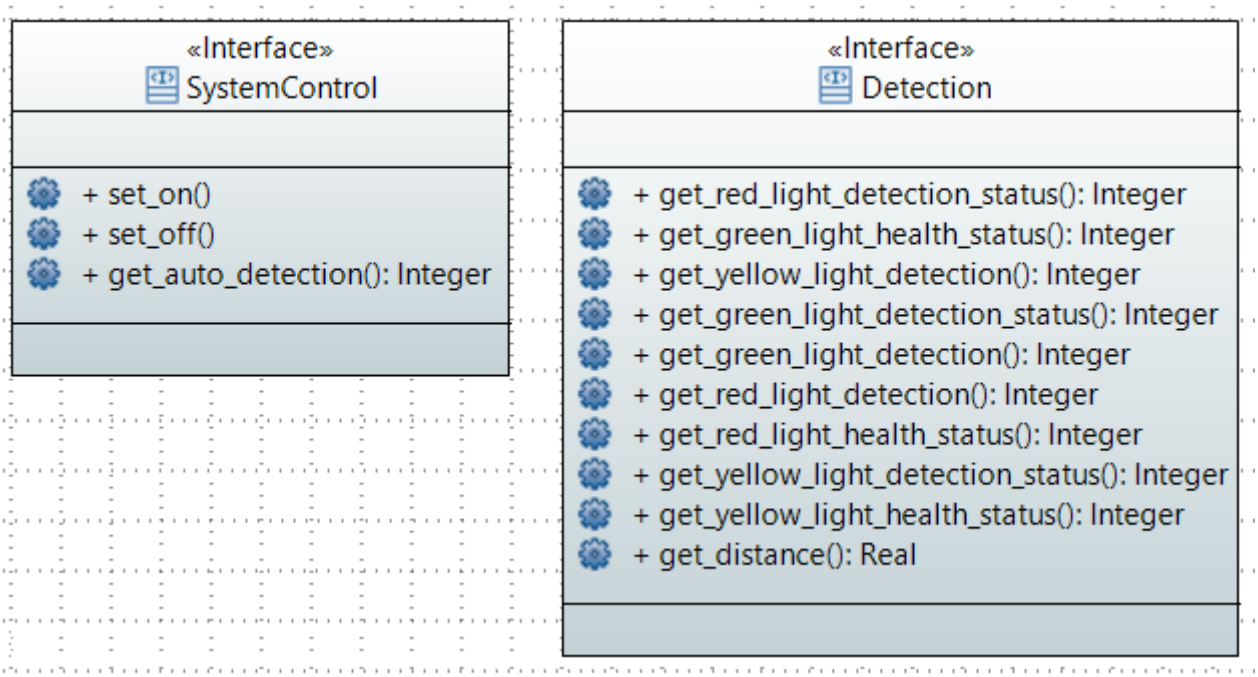


Figure 7.12: Serial embedded device model

As this embedded system communicates to the Remote Library Proxy via Serial, it is required to annotate the model with that information. As the communication is serial not only it requires the label **"#TAFES_RobotLibSERIAL"**, as well as annotating each operation with the command that the embedded device expects to receive with the prefix **"#TAFES_SerialCommand_"**. For example, the operation *set_off* is annotated with

"#TAFES_SerialCommand_1" as the command that the embedded device expects to receive to set the detection OFF is 1.

Following the design and annotation of the embedded device it is configuring necessary to configure TAFES to successfully generate the test library to test the SUT. As the communication required to interact with the device is Serial, we have to configure the file *serial.conf*. The end result of the configuration in this use case is presented in Figure 7.13.

serial.conf 183 Bytes

```
1  [DEFAULT]
2  PORT = /dev/ttyACM0
3  BAUDRATE =  9600
4  TIMEOUT = 1
5  WRITE_TIMEOUT = 1
6  MESSAGE_START_MARKER = <
7  MESSAGE_END_MARKER = >
8  MESSAGE_CONTENT_SEPARATOR = ,
9  RLP_HOST_IP = pi@10.12.15.50
```

Figure 7.13: Configuration file for Serial communication

With the .uml file originated from the eclipse plug-in **Papyrus**, and the configuration adapted to the embedded device, the test library can be generated automatically and deployed at the Remote Library Proxy. Figure 7.14 is a sample of the library, where is used the information from the configuration file to create a connection via Serial to the embedded device.

Next step is to use the .uml file generated from the model and the configuration file to automatically generate the test library and deploy it into the Remote Library Proxy through (section 6.3.2).

The core functionality of this embedded system is the same as the simulated version, which detects an object and turns on the light that corresponds to that distance. However, the developed embedded device has an extra functionality which is the capability of recognizing the status of detection and LED health for each distance range. Regarding the functionality of the embedded system we have selected the following test cases, their specification can be found in appendix B:

- Set auto detection *OFF* and verify if the detection state is *OFF*, and it is not detecting any object in any light distance range, and distance is -1.
- Set auto detection *ON* and verify if the detection state is *ON*, and the detection and LED status are positive.
- Place an object at a certain distance and verify if the sensor detection distance is correct.
- Place an object at a certain distance and verify if lights turn on accordingly.

specificKWProxySerial.py 2.51 KB Raw Bl

```
from Utils import serialutilities

class SerialLibrary(object):

        def __init__(self):
                super(SerialLibrary,self).__init__()
                self.serial = serialutilities.create_serial_connection("/dev/ttyACM0", 9600, 1, 1, "<", ">", ",")

        def Detection_get_red_light_detection_status(self):
                log, Integer__status = serialutilities.execute_command(self.serial, 7,"get_red_light_detection_status")
                print(log)
                return Integer__status

        def Detection_get_green_light_health_status(self):
                log, Integer__status = serialutilities.execute_command(self.serial, 11,"get_green_light_health_status")
                print(log)
                return Integer__status

        def Detection_get_green_light_detection(self):
                log, Integer__status = serialutilities.execute_command(self.serial, 8,"get_green_light_detection")
                print(log)
                return Integer__status

        def Detection_get_green_light_detection_status(self):
                log, Integer__status = serialutilities.execute_command(self.serial, 5,"get_green_light_detection_status")
                print(log)
                return Integer__status

        def Detection_get_yellow_light_detection_status(self):
                log, Integer__status = serialutilities.execute_command(self.serial, 6,"get_yellow_light_detection_status")
                print(log)
                return Integer__status
```

Figure 7.14: Remote Library Proxy's keyword library

In every test case we connect to the Remote Library Proxy to call the required keywords for specifying the test case. Additionally, Robot Framework generates the report and log files automatically, that can latter be accessed informing the tester the test case execution status. Appendix C.2 represent a example of output files for this embedded device

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Throughout this document, we have argued the importance of testing, particularly in embedded systems. In embedded systems given its complexity, and harm that can be inflicted to the real-word if the system does not behave accordingly to its specifications. Thus, we present a study on how to test embedded devices where we have seen that systems with different characteristics require different testing techniques.

To face the problem of testing embedded systems, we resort to an novel method of testing which relies on automating all the test process with the objective of increasing the efficiency. We selected a proficient test automation tool and explored how could it be used to test embedded systems. As we are ultimately performing black-box testing, our test automation tool has to be capable of communicating with the system under testing, and verify that it behaves as expected.

We have selected four test automation tools which we could potentially expand for testing embedded systems. These frameworks go through an evaluation process to select the most suitable. Upon the evaluation process we selected **Robot Framework** as the test automation tool. The reason behind this selection is the fact that this framework is quite rich and supports various types of test automation (Keyword-Driven, Behaviour-Driven, etc.). Furthermore, the creation of libraries can easily be done through Python programming language that itself has a huge capability on communicating with embedded devices.

We have designed a solution that requires three components. First, we have the **Testing Tool** component, this is where the tester can specify and run test cases. Robot Framework, together with its IDE (RIDE) belongs to this component, allows a clean and easy writing of test cases, execution of test cases and generation of report files. The second

component is responsible of storing the test library as well as executing the test steps that most likely to communicate with the embedded device. The introduction of Robot Framework in this component is done through the Robot Framework remote server that stores the test library. Moreover, Remote Library Proxy has a module that communicates with the embedded device in order to execute the test step. Given the fact that Remote Library Server is an external component from the others it can be deployed in a different system, for example in a Raspberry Pi. The last component is the **Embedded Device** that represents the system under test. A embedded system is considered viable to be tested with our solution if it is under development and follows a Model Driven methodology which will allow our framework to automatically generate test libraries by annotating the model with the specific labels and complete the configuration files regarding the method of communication that the embedded device allows.

To finalize and prove that our solution is viable we have set up two use cases with a proximity sensor as an embedded device. The development of these use cases followed a Model Driven methodology where the primary artifacts are models that describe each device and its operations reachable from the "outside world". These use cases were chosen given the fact that both have a different method of communication. The first is a simulated proximity sensor that communicates via XML-RPC , while the second a proximity sensor that communicates via Serial communication. We wrote test cases for these embedded devices and they were successfully tested with the Test Automation Framework for Embedded Devices (TAFES).

Therefore, we proved that indeed it is possible to use conventional test automation tools like Robot Framework, highly used in web testing, to test embedded devices, and take full advantage of its perks to elevate the efficiency of testing. Also, having an external component that acts as a proxy server between the Robot Framework and the embedded device allows a new approach to testing these devices. In particular, embedded devices can now be tested remotely using only a single hardware tool, for example, a Raspberry Pi. Finally, model-driven development is a respected development method for embedded devices and utilizing it to automatically generate test libraries enhances our testing framework in terms of flexibility among different embedded systems and time efficiency.

8.2 Future Work

The dissertation's goals were achieved, however, additional features and improvements can be made:

- TAFES model analysis is restricted, it can only analyze class diagrams and generate keywords from the operations in the Interface element. TAFES could be extended to be able to analyze different models and generate test data from them. For example, the analysis could generate test cases or high level keywords from activity or sequence diagrams.

- We have introduced the two types of libraries: Standard and Specific. However, setting standard keywords can add complexity to our problem as it is not possible to ensure that every embedded system has that keywords that we considered standard. Therefore, some analysis to establish which keywords are considered generic. Another approach to consider every keyword as an embedded device specific, which means it comes from the model. Ultimately the tester will not recognize any difference between both approaches.
- As the test libraries are saved in a remote server, an interesting feature is having the possibility of having multiple test libraries that can either be stored in a single external component or in multiple external components. This is quite useful when having a complex system under test.

Bibliography

[1] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2005. ISBN: 0750677929.

[2] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-43287-0.

[3] E. Kit and S. Finzi. *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. ISBN: 0-201-87756-2.

[4] N. Navet and F. Simonot-Lion. *Automotive Embedded Systems Handbook*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2008. ISBN: 084938026X, 9780849380266.

[5] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[6] B. Broekman and E. Notenboom. *Testing Embedded Software*. Addison-Wesley Professional, 2002. ISBN: 0321159861.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu. "An orchestrated survey of methodologies for automated software test case generation." In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[8] H. Zhu, P. A. V. Hall, and J. H. R. May. "Software unit test coverage and adequacy." In: *ACM Computing Surveys* 29.4 (1997), 366–427. DOI: 10.1145/267580.267590.

[9] A. Bertolino. "Software Testing Research: Achievements, Challenges, Dreams." In: *Future of Software Engineering (FOSE 07)* (2007). DOI: 10.1109/fose.2007.25.

[10] M. Pezze and M. Young. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008.

[11] M. Kleine-Budde. "SocketCAN - The official CAN API of the Linux kernel." In: (2012).

[12] M.-C. Gaudel. "Testing Can Be Formal, Too." In: *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*. TAPSOFT '95. London, UK, UK: Springer-Verlag, 1995, pp. 82–96. ISBN: 3-540-59293-8.

[13] J. Offutt and A. Abdurazik. "Generating Tests from UML Specifications." In: *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*. UML'99. Fort Collins, CO, USA: Springer-Verlag, 1999, pp. 416–429. ISBN: 3-540-66712-1.

[14] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. "Projected State Machine Coverage for Software Testing." In: *SIGSOFT Softw. Eng. Notes* 27.4 (July 2002), pp. 134–143. ISSN: 0163-5948. DOI: 10.1145/566171.566192.

[15] J. Tretmans. "Formal Methods and Testing." In: ed. by R. M. Hierons, J. P. Bowen, and M. Harman. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Model Based Testing with Labelled Transition Systems, pp. 1–38. ISBN: 3-540-78916-2, 978-3-540-78916-1.

[16] L. de Alfaro and T. A. Henzinger. "Interface Automata." In: *SIGSOFT Softw. Eng. Notes* 26.5 (Sept. 2001), pp. 109–120. ISSN: 0163-5948. DOI: 10.1145/503271.503226.

[17] R. Groz, O. Charles, and J. Renévot. "Relating conformance test coverage to formal specifications." In: *Formal Description Techniques IX*. Springer US, 1996, pp. 195–210. DOI: 10.1007/978-0-387-35079-0_12.

[18] L. M. G. Feijs, N. Goga, S. Mauw, and J. Tretmans. "Test Selection, Trace Distance and Heuristics." In: *Testing of Communicating Systems XIV*. Springer US, 2002, pp. 267–282. DOI: 10.1007/978-0-387-35497-2_20.

[19] C. Jard and T. Jéron. "TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-deterministic Reactive Systems." In: *Int. J. Softw. Tools Technol. Transf.* 7.4 (Aug. 2005), pp. 297–315. ISSN: 1433-2779. DOI: 10.1007/s10009-004-0153-x.

[20] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. "Generating Finite State Machines from Abstract State Machines." In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. Roma, Italy: ACM, 2002, pp. 112–122. ISBN: 1-58113-562-9. DOI: 10.1145/566172.566190.

[21] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. "Optimal Strategies for Testing Nondeterministic Systems." In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 55–64. ISBN: 1-58113-820-2. DOI: 10.1145/1007512.1007520.

[22] W. Grieskamp, N. Kicillof, and N. Tillmann. "Action Machines: a Framework for Encoding and Composing Partial Behaviors." In: 16 (Oct. 2006), pp. 705–726.

[23] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem." In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 182–191. DOI: 10.1109/ICSTW.2010.31.

[24] J. Wegener and O. Bühler. "Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System." In: *Genetic and Evolutionary Computation – GECCO 2004*. Springer Berlin Heidelberg, 2004, pp. 1400–1412. DOI: 10.1007/978-3-540-24855-2_160.

[25] J. Wegener and M. Grochtmann. In: *Real-Time Systems* 15.3 (1998), pp. 275–298. DOI: 10.1023/a:1008096431840.

[26] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. "Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs." In: *Comput. J.* 49.3 (May 2006), pp. 331–344. ISSN: 0010-4620. DOI: 10.1093/comjnl/bx1003.

[27] C. D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman, and M. Luck. "Evolutionary Testing of Autonomous Software Agents." In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '09. Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 521–528. ISBN: 978-0-9817381-6-1.

[28] M. Harman, F. Islam, T. Xie, and S. Wappler. "Automated Test Data Generation for Aspect-oriented Programs." In: *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*. AOSD '09. Charlottesville, Virginia, USA: ACM, 2009, pp. 185–196. ISBN: 978-1-60558-442-3. DOI: 10.1145/1509239.1509264.

[29] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. "Constructing Test Suites for Interaction Testing." In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 38–48. ISBN: 0-7695-1877-X.

[30] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, and A. Pozo. "Integration Test of Classes and Aspects with a Multi-Evolutionary and Coupling-Based Approach." In: *Search Based Software Engineering*. Springer Berlin Heidelberg, 2011, pp. 188–203. DOI: 10.1007/978-3-642-23716-4_18.

[31] M. Harman, Y. Jia, and W. B. Langdon. "Strong Higher Order Mutation-based Test Data Generation." In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 212–222. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025144.

[32] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. "TimeAware Test Suite Prioritization." In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 1–12. ISBN: 1-59593-263-1. DOI: 10.1145/1146238.1146240.

[33] C. Del Grosso, G. Antoniol, M. Di Penta, P. Galinier, and E. Merlo. "Improving Network Applications Security: A New Heuristic to Generate Stress Testing Data." In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. GECCO '05. Washington DC, USA: ACM, 2005, pp. 1037–1043. ISBN: 1-59593-010-8. DOI: 10.1145/1068009.1068185.

[34] N. Alshahwan and M. Harman. "Automated Web Application Testing Using Search Based Software Engineering." In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 3–12. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100082.

[35] H. P. Enterprise. *HPE Unified Functional Testing-User Guide*. English. Version Version 14.02. HPE. 859 pp.

[36] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. "What is the benefit of a model-based design of embedded software systems in the car industry ?" In: 2011.

[37] B. Selic. "The Pragmatics of Model-Driven Development." In: *IEEE Softw.* 20.5 (Sept. 2003), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146. URL: https://doi.org/10.1109/MS.2003.1231146.

Webography

[38] A. Berger. *The basics of embedded software testing*. 2011 (accessed at February 8, 2018). URL: https://www.embedded.com/design/other/4212929/1/The-basics-of-embedded-software-testing--Part-2.

[39] tutorialspoint.com. *Embedded Systems Overview*. 2018 (accessed at February 8,2018). URL: https://www.tutorialspoint.com/embedded_systems/es_overview.htm.

[40] *What is Automation Testing?* 2014 (accessed at February 18, 2018). URL: http://www.softwaretestingclass.com/what-is-automation-testing/.

[41] *AUTOMATION TESTING Tutorial: Process, Planning & Tools*. accessed at February 8, 2018. URL: https://www.guru99.com/automation-testing.html.

[42] *Continuous Integration*. 2018 (accessed at February 19, 2018). URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[43] G. Technologies. *How to Choose the Right Test Automation Framework*. 2017 (accessed at February 8, 2018). URL: https://www.glowtouch.com/blog/testing/how-to-choose-the-right-test-automation-framework/.

[44] S. Tutorials. *Most Popular Test Automation Frameworks with Pros and Cons*. 2017 (accessed at Frebruary 8, 2018). URL: http://www.softwaretestinghelp.com/test-automation-frameworks-selenium-tutorial-20/.

[45] *Robot Framework User Guide*. 2018 (accessed at September 20, 2018). URL: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

[46] *Gauge Framework's Documentation*. 2018 (accessed at February 9, 2018). URL: https://docs.gauge.org/index.html.

[47] *Python-can's Documentation*. 2018 (accessed at February 19, 2018). URL: http://python-can.readthedocs.io/en/latest/.

[48] *Robot Framework IDE Wiki*. 2018 (accessed at September 20, 2018). URL: https://github.com/robotframework/RIDE/wiki.

[49] *Test Case Specification*. 2018 (accessed at September 20, 2018). URL: http://toolsqa.com/software-testing/test-case-specification/.

[50] *Robot Framework's python remote server*. 2018 (accessed at September 20, 2018). URL: https://github.com/robotframework/PythonRemoteServer.

[51] *What is serial communication and how it works?* 2018 (accessed at September 20, 2018). URL: https://www.codrey.com/embedded-systems/serial-communication-basics/.

[52] *Using model-driven development to reduce system software security vulnerabilities*. 2018 (accessed at September 20, 2018). URL: https://www.embedded.com/design/programming-languages-and-tools/4429403/Using-model-driven-development-to-reduce-system-software-security-vulnerabilities-.

[53] *PySerial documentation*. 2018 (accessed at September 20, 2018). URL: https://pythonhosted.org/pyserial/.

Appendix A

Auxiliary tables

Table A.1: Relation between techniques and test levels and types, adapted from [6]

Techniques	**Test levels, types**
Failure mode and effect analysis (FMEA)	Detailed design verification
Fault injection (FTA)	Hardware/software integration test
Formal verification	Model integration test
Rare event testing	Software integration test
Fault tree analysis	Host/target test
State transition testing	System integration test
Statistical usage testing	Unit test

Appendix B

Proximity sensor test cases

In this appendix's chapter we present the specification of test cases executed in the proximity sensor (section 7.3). The following sections describe each test case specification.

B.1 Set Off

The first test case is called *Set Off*. The goal of this test case is to verify that the embedded device can successfully turn OFF the detection of objects. In this action the state of the embedded device change. The auto detect is set to *OFF*, it is not possible to detect in any distance range, thus the detection for each range is turn down to 0. Moreover, in this test case we verify the status of detection is *Working* and LED health *Ok* is for each range. Figure B.1 represents the test specification in the Robot Framework IDE.

B.2 Set On

Second test case executed in the proximity sensor is *Set ON*. For this test it is expected to activate the auto detection of objects. After activating the auto detection, we check if its state is *ON*. Then we verify if the status of detection is and the LED health status are *Ok*. Figure B.2 is a representation of the test case specification in the Robot Framework IDE.

B.3 Distance Reading

The third test case for the proximity sensor has the goal of verifying the detected distance. To perform this we have created a user keyword called *Check Distance* (Figure B.4). This keyword calls another user keyword name *Set Distance* (Figure B.5) which tells the tester

Set Off

▶ Settings

1	**SystemControl Set Off**		
2	${autoDetect}	**SystemControl Get Auto Detection**	
3	**Should Be Equal As Strings**	${autoDetect}	OFF
4	${distance}	**Detection Get Distance**	
5	**Should Be Equal As Numbers**	-1	${distance}
6	${greenLightDetection}	**Detection Get Green Light Detection**	
7	${yellowLightDetection}	**Detection Get Yellow Light Detection**	
8	${redLightDetection}	**Detection Get Red Light Detection**	
9	**Should Be Equal As Integers**	${greenLightDetection}	0
10	**Should Be Equal As Integers**	${yellowLightDetection}	0
11	**Should Be Equal As Integers**	${redLightDetection}	0
11	**Should Be Equal As Integers**	${redLightDetection}	0
12	${greenLightDetectionStatus}	**Detection Get Green Light Detection Status**	
13	${yellowLightDetectionStatus	**Detection Get Yellow Light Detection Status**	
14	${redLightDetectionStatus}	**Detection Get Red Light Detection Status**	
15	**Should Be Equal As Integers**	${greenLightDetectionStatus}	1
16	**Should Be Equal As Integers**	${yellowLightDetectionStatus	1
17	**Should Be Equal As Integers**	${redLightDetectionStatus}	1
18	${greenLightHealthStatus}	**Detection Get Green Light Health Status**	
19	${yellowLightHealthStatus}	**Detection Get Yellow Light Health Status**	
20	${redLightHealthStatus}	**Detection Get Red Light Health Status**	
21	**Should Be Equal As Integers**	${greenLightHealthStatus}	1
22	**Should Be Equal As Integers**	${yellowLightHealthStatus}	1

Figure B.1: Test case specification - Set Off

Set On

▶ Settings

1	**SystemControl Set On**		
2	${autoDetect}	**SystemControl Get Auto Detection**	
3	**Should Be Equal As Strings**	${autoDetect}	ON
4	${greenLightDetectionStatus}	**Detection Get Green Light Detection Status**	
5	${yellowLightDetectionStatus	**Detection Get Yellow Light Detection Status**	
6	${redLightDetectionStatus}	**Detection Get Red Light Detection Status**	
7	**Should Be Equal As Integers**	${greenLightDetectionStatus}	1
8	**Should Be Equal As Integers**	${yellowLightDetectionStatus	1
9	**Should Be Equal As Integers**	${redLightDetectionStatus}	1
10	${greenLightHealthStatus}	**Detection Get Green Light Health Status**	
11	${yellowLightHealthStatus}	**Detection Get Yellow Light Health Status**	
12	${redLightHealthStatus}	**Detection Get Red Light Health Status**	
13	**Should Be Equal As Integers**	${greenLightHealthStatus}	1
14	**Should Be Equal As Integers**	${yellowLightHealthStatus}	1
15	**Should Be Equal As Integers**	${redLightHealthStatus}	1

Figure B.2: Test case specification - Set On

to move the object to an certain distance from the embedded device. After executed *Set Distance* it is verified if the detected distance is approximately the expected.

Moreover, we have set the user keyword *Check Distance* as a Robot Framework template, allowing to parametrize the test execution. A set of values of expected distance values is selected (Figure B.3). Thus, when executing this test case it is required to change the object distance accordingly to the input parameters.

B.4 Detection of object and LED output

The final test that we perform in the proximity sensor has the purpose of verify if the embedded device can correctly correlate the detected distance to the output LED lights. Therefore, it uses a user keyword named *Check detection for a given distance* (Figure B.7).

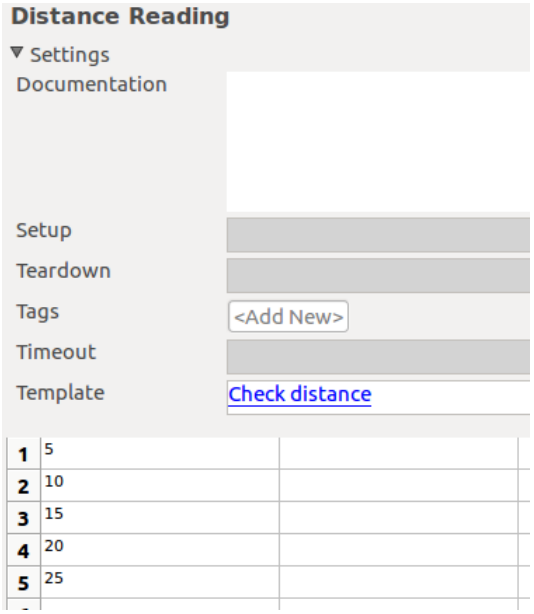


Figure B.3: Test case specification - Distance Reading

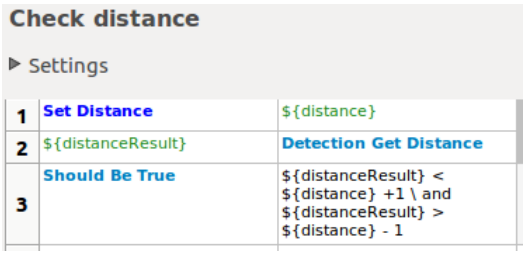


Figure B.4: Check distance user keyword

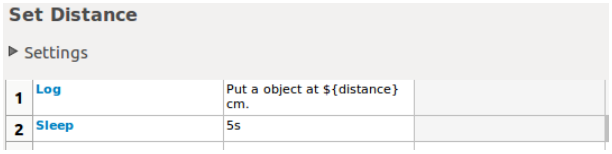


Figure B.5: Set distance user keyword

This keyword is the aggregation of the previously used user keywords, *Check Distance* (Figure B.4 and a new user keyword *Check Detection* (Figure B.8. The later is for verification of the detection values in the different ranges.

This test case uses *Check detection fora a given distance* as a template, where the input the expected distance and expected detection value in the three distance ranges. Thus, the execution of this test case requests to set a object in the selected distance and then it verifies if the detection values are the expected.

Detection of object and LED output

▼ Settings

Documentation	
Setup	
Teardown	
Tags	<Add New>
Timeout	
Template	Check detection for a given distance

1	5	1	0	0
2	10	0	1	0
3	15	0	1	0
4	20	0	0	1
5	30	0	0	1

Figure B.6: Test case specification - Detection of object and LED output

Check detection for a given distance

▶ Settings

1	**Check distance**	${distance}		
2	**Check detection**	${redLightDetection}	${yellowLightDetection}	${greenLightDetection}

Figure B.7: Check detection for a given distance user keyword

Check detection

▶ Settings

1	${greenLightDetectionResult}	**Detection Get Green Light Detection**	
2	${yellowLightDetectionResult	**Detection Get Yellow Light Detection**	
3	${redLightDetectionResult}	**Detection Get Red Light Detection**	
4	**Should Be Equal As Integers**	${greenLightDetectionResult}	${greenLightDetection}
5	**Should Be Equal As Integers**	${yellowLightDetectionResult	${yellowLightDetection}
6	**Should Be Equal As Integers**	${redLightDetectionResult}	${redLightDetection}

Figure B.8: Check detection user keyword

Appendix C

Test Case's output files

In this appendix, we present the automatically generated output files after execution of a test case. In chapter 7, we selected two use cases for our framework and in the end result, we specified a set of test cases to execute. Thus, section C.1 indicates the output files for the simulated proximity sensor, which communication method is XML-RPC. Section C.2 exposes the generated output files from the second use case (Proximity sensor with Serial communication). For each case, we present a successful and a failed test case execution.

C.1 XML-RPC proximity sensor output files

Regarding the simulated proximity sensor, we have executed three test cases. Figure C.1 represents the test report when the test has passed. Moreover, it presents the status of each test case and its elapsed time.

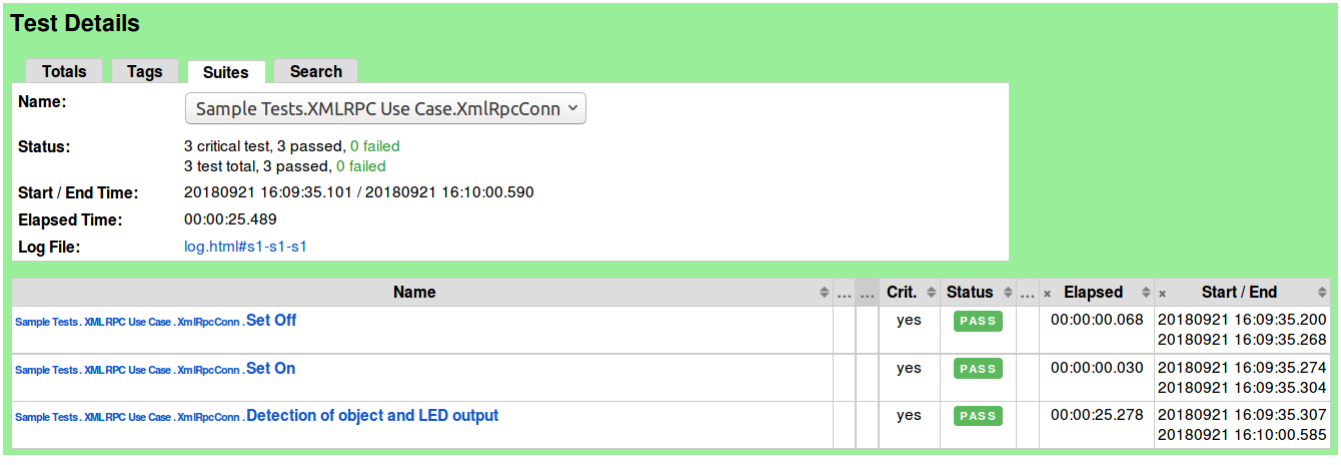
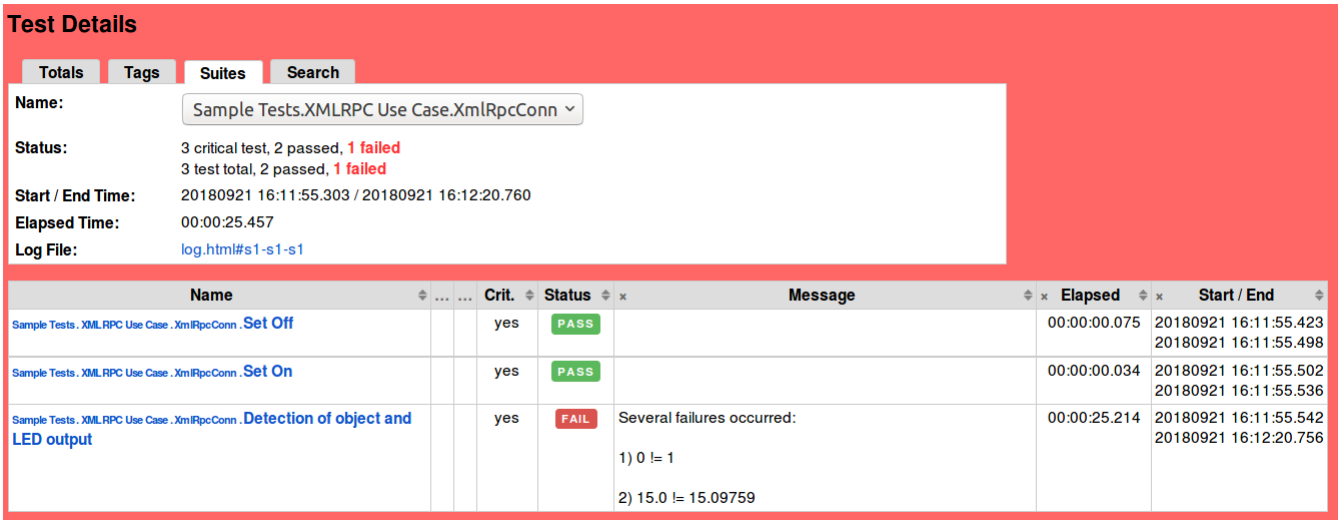


Figure C.1: XML-RPC proximity sensor test report - PASS



Test Details

Totals | Tags | Suites | Search

Name:	Sample Tests.XMLRPC Use Case.XmlRpcConn
Status:	3 critical test, 2 passed, 1 failed 3 test total, 2 passed, 1 failed
Start / End Time:	20180921 16:11:55.303 / 20180921 16:12:20.760
Elapsed Time:	00:00:25.457
Log File:	log.html#s1-s1

Name	Crit.	Status	Message	Elapsed	Start / End
Sample Tests . XMLRPC Use Case . XmlRpcConn . Set Off	yes	PASS		00:00:00.075	20180921 16:11:55.423 20180921 16:11:55.498
Sample Tests . XMLRPC Use Case . XmlRpcConn . Set On	yes	PASS		00:00:00.034	20180921 16:11:55.502 20180921 16:11:55.536
Sample Tests . XMLRPC Use Case . XmlRpcConn . Detection of object and LED output	yes	FAIL	Several failures occurred: 1) 0 != 1 2) 15.0 != 15.09759	00:00:25.214	20180921 16:11:55.542 20180921 16:12:20.756

Figure C.2: XML-RPC proximity sensor test report - FAIL

Together with the development of the embedded device, we also developed a software that causes the system to fail. Thus, by running such software the test execution has failed. By analysing the test report it is clear that the failed test case is the third, *Detection of object and LED output*, figure C.2 represents the test report. Along with the status comes an execution message and in the failed test case the message gives us the raised exceptions. It is now clear on each stage of testing the embedded device did not behave accordingly. Thus, we have to analyze the test case execution with deeper detail, to do so, we resort on the generated log files. Figure C.3 show us the two keywords that have raised exception. In Figure C.3a, the reason that resulted in a test case failure is the fact that with the expected distance values of 10 and detection values of 0/1/0 corresponding to the red, yellow and green light. However, when verifying if the green light had a detection value of 0 the test case failed, meaning that the embedded device was detection in a distance range that it was not supposed to, then a bug exists.

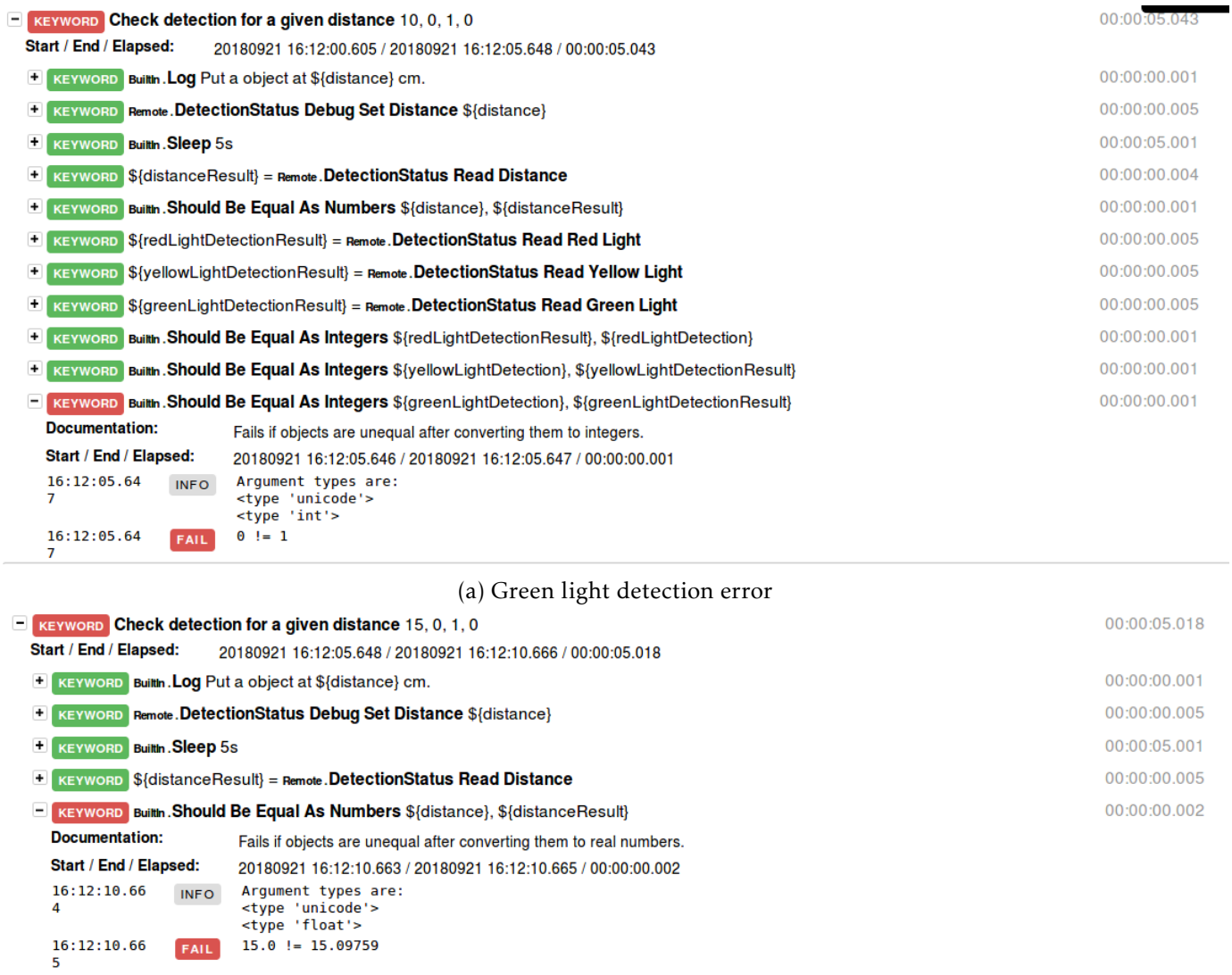
Regarding the second failure on the test case (Figure C.3b), it was provoked in the next iteration, and it is related to a detected distance. It was expected to detect an object at 15 but instead, it detected a hundredth higher.

C.2 Serial proximity sensor output files

For the embedded device with the support of communication via Serial we have also presented a set of test results and test logs. Figure C.4 show us the test report when the execution of the test case is successful.

We later performed the same test cases, but this time they had the test result of *FAIL*. Figure C.5 is the test report for that execution.

The test report has failed in two different test cases of the same test suit. However, error messages have the same structure suggesting that the failure might be the same. To



(a) Green light detection error

(b) Detected distance error

Figure C.3: XML-RPC proximity sensor log file with failed test case

analyse with more detail we resort on the generated log files (Figure C.6). In this file, we have a failure in the test execution on both test cases. For the test case *Distance Reading* (Figure C.6a) the failure occurs in the user keyword *Check Distance*. The first iteration of this keyword is executed with success, however, the remaining test execution results are failures. The bug is encounter in the verification of detected distance. It is required a certain distance for each iteration, however, the distance detected in all iterations is similar.

Regarding test case *Detection of object and LED output*, the error messages are similar to the earlier test case. The reason for this is the fact that *Detection of object and LED output* also calls the user keyword *Check Distance* and the errors are in the same iterations.

With the analysis of the test report and log files of this test execution we can conclude

Test Details

LOG

Totals | Tags | Suites | Search

Name: Sample Tests.Serial Use Case.Arduino

Status: 4 critical test, 4 passed, 0 failed
4 test total, 4 passed, 0 failed

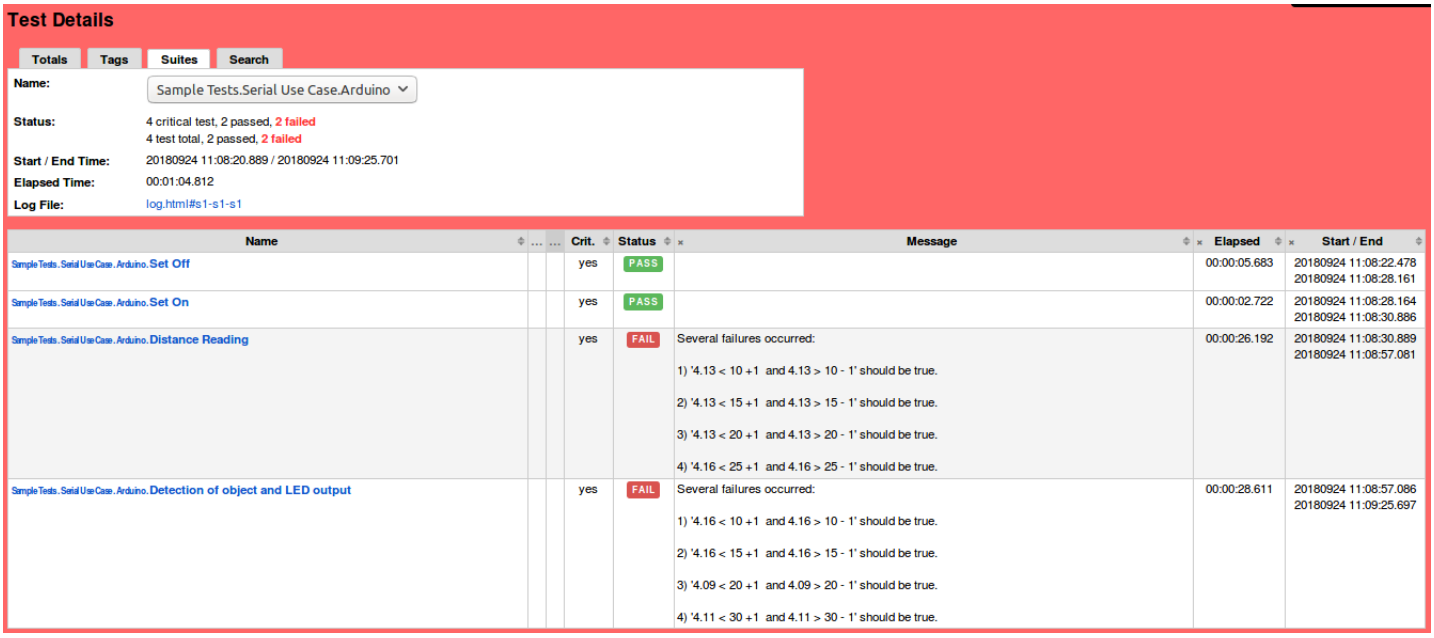
Start / End Time: 20180921 17:23:54.886 / 20180921 17:25:00.428

Elapsed Time: 00:01:05.542

Log File: log.html#s1-s1

Name	Crit.	Status	Elapsed	Start / End
Sample Tests . Serial Use Case . Arduino . Set Off	yes	PASS	00:00:03.894	20180921 17:23:57.400 20180921 17:24:01.294
Sample Tests . Serial Use Case . Arduino . Set On	yes	PASS	00:00:02.386	20180921 17:24:01.297 20180921 17:24:03.683
Sample Tests . Serial Use Case . Arduino . Distance Reading	yes	PASS	00:00:26.249	20180921 17:24:03.686 20180921 17:24:29.935
Sample Tests . Serial Use Case . Arduino . Detection of object and LED output	yes	PASS	00:00:30.485	20180921 17:24:29.939 20180921 17:25:00.424

Figure C.4: Serial proximity sensor test report - PASS



Test Details

Totals | Tags | Suites | Search

Name: Sample Tests.Serial Use Case.Arduino

Status: 4 critical test, 2 passed, 2 failed
4 test total, 2 passed, 2 failed

Start / End Time: 20180924 11:08:20.889 / 20180924 11:09:25.701

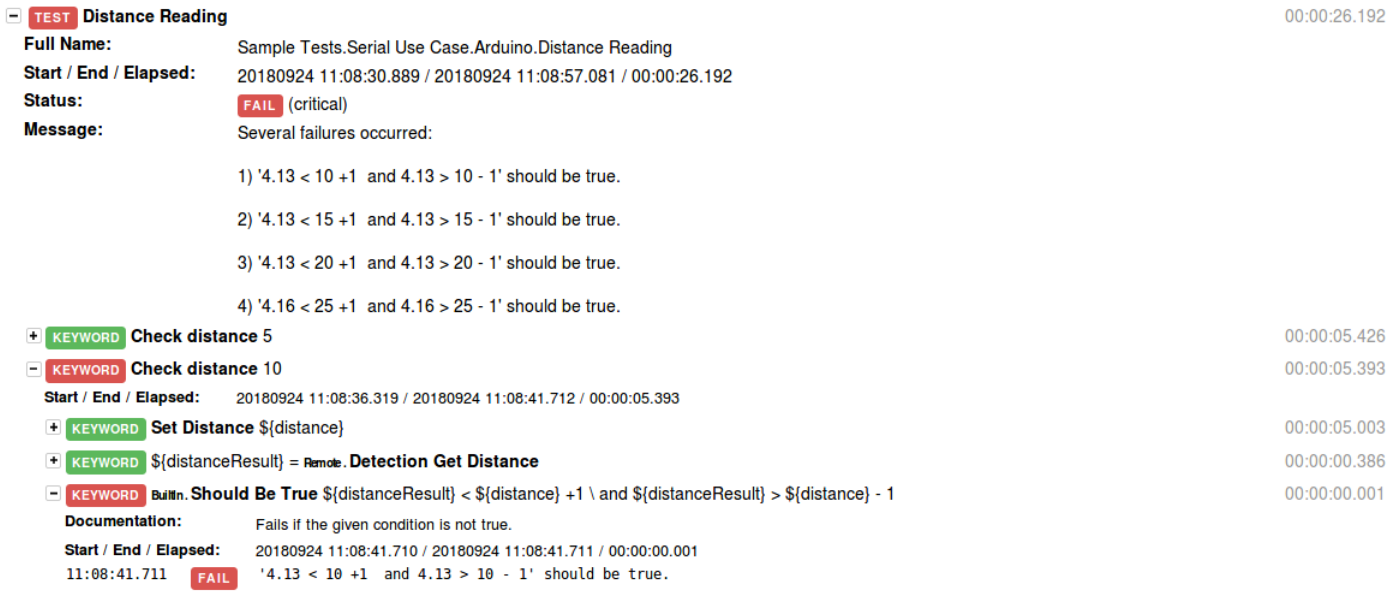
Elapsed Time: 00:01:04.812

Log File: log.html#s1-s1

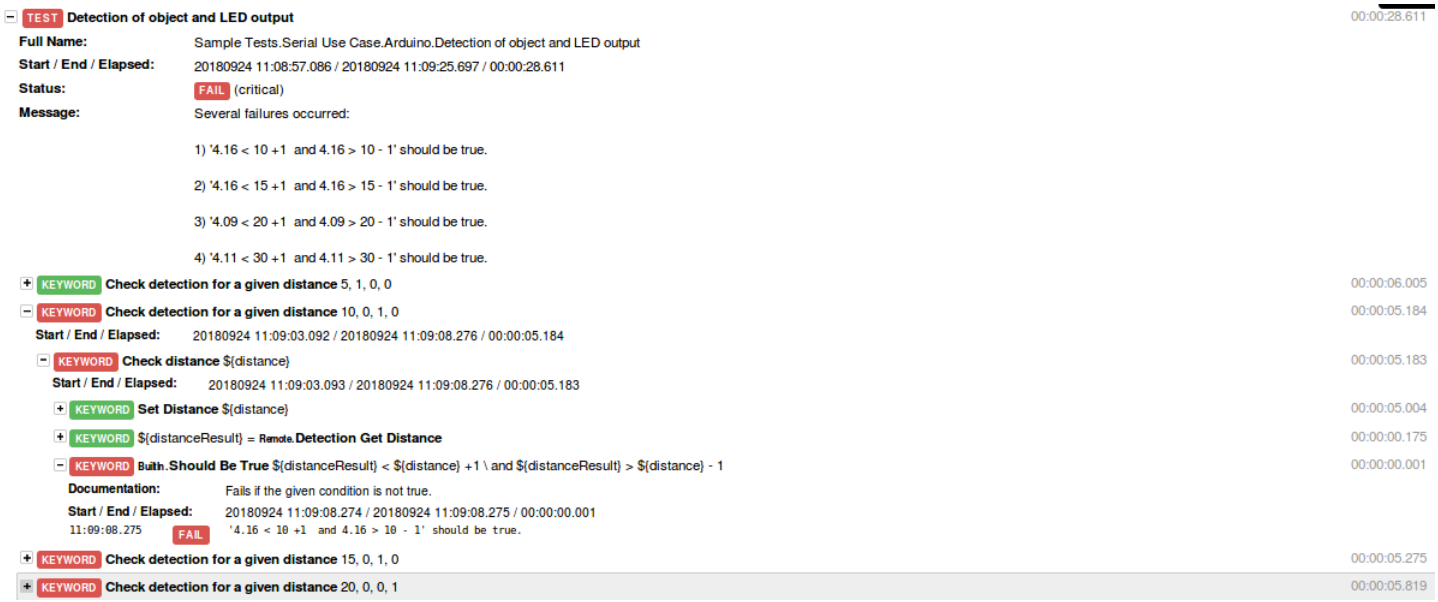
Name	Crit.	Status	Message	Elapsed	Start / End
Sample Tests . Serial Use Case . Arduino . Set Off	yes	PASS		00:00:05.683	20180924 11:08:22.478 20180924 11:08:28.161
Sample Tests . Serial Use Case . Arduino . Set On	yes	PASS		00:00:02.722	20180924 11:08:28.164 20180924 11:08:30.886
Sample Tests . Serial Use Case . Arduino . Distance Reading	yes	FAIL	Several failures occurred: 1) '4.13 < 10 +1 and 4.13 > 10 - 1' should be true. 2) '4.13 < 15 +1 and 4.13 > 15 - 1' should be true. 3) '4.13 < 20 +1 and 4.13 > 20 - 1' should be true. 4) '4.16 < 25 +1 and 4.16 > 25 - 1' should be true.	00:00:26.192	20180924 11:08:30.889 20180924 11:08:57.081
Sample Tests . Serial Use Case . Arduino . Detection of object and LED output	yes	FAIL	Several failures occurred: 1) '4.16 < 10 +1 and 4.16 > 10 - 1' should be true. 2) '4.16 < 15 +1 and 4.16 > 15 - 1' should be true. 3) '4.09 < 20 +1 and 4.09 > 20 - 1' should be true. 4) '4.11 < 30 +1 and 4.11 > 30 - 1' should be true.	00:00:28.611	20180924 11:08:57.086 20180924 11:09:25.697

Figure C.5: Serial proximity sensor test report - FAIL

that a defect exists with a high probability in *Check Distance* keyword.



(a) Distance Reading - FAIL



(b) Detection of object and LED output - FAIL

Figure C.6: Serial proximity sensor log file with failed test cases

Appendix D

Proximity sensor's embedded software - Arduino

```
#include "Ultrasonic.h"

////////Setup up////////

//create a ultrasonic object defined by the pins. Trigger - 6, Echo - 7
Ultrasonic ultrasonic(6,7);

//declare digital pins
const int greenLight = 13;
const int yellowLight = 12;
const int redLight = 11;
const int lightSwitch = 8;
//////////////////////////

// Receive with start- and end-markers combined with parsing
const byte numChars = 64;
char receivedChars[numChars];
char tempChars[numChars];        // temporary array for use by strtok() function

      // variables to hold the parsed data
char commandCode[numChars] = {0};

boolean newData = false;

boolean isOn = true;
float distance = 0;
long microsec = 0;

const char startMarker = '<';
```

```
30 const char endMarker = '>';
31 const String messageOk = "OK";
32 const String messageFail = "FAIL";
33 //=============
34 
35 void setup() {
36 
37     Serial.begin(9600);
38 
39     pinMode(greenLight, OUTPUT);
40     pinMode(yellowLight, OUTPUT);
41     pinMode(redLight, OUTPUT);
42     pinMode(lightSwitch, INPUT);
43 
44 
45 }
46 
47 //=============
48 
49 void loop() {
50 
51 
52     if(isOn){
53 
54       //read sensor
55       microsec = ultrasonic.timing();
56 
57       //distance in cm
58       distance = ultrasonic.convert(microsec, Ultrasonic::CM);
59 
60       updateStatus();
61     }
62 
63     recvWithStartEndMarkers();
64     if (newData == true) {
65         strcpy(tempChars, receivedChars);
66             // this temporary copy is necessary to protect the original data
67             //   because strtok() replaces the commas with \0
68         parseData();
69         //showParsedData();
70         newData = false;
71     }
72     delay(500);
73 }
74 
75 void setOff(){
76 
77   isOn = false;
78   digitalWrite(greenLight,LOW);
79   digitalWrite(yellowLight,LOW);
```

```
80    digitalWrite(redLight,LOW);
81    distance = -1;
82    //string out = startMarker + messageOk + endMarker;
83    Serial.write("<OK,>");
84    //Serial.println("Recebi");
85  }
86
87  void setOn(){
88
89    isOn = true;
90    //string out = startMarker + messageOk + endMarker;
91    //Serial.println("Recebi");
92    Serial.write("<OK,>");
93  }
94
95  void getAutoDetection(){
96    if(isOn){
97      //string out = startMarker + messageOk + ",ON" + endMarker;
98      Serial.write("<OK,ON,>");
99    }else{
100     Serial.write("<OK,OFF,>");
101   }
102 }
103
104
105 void getGreenLightStatus(){
106   int greenLightStatus = 0;
107
108   //if(digitalRead(lightSwitch)){
109     greenLightStatus = digitalRead(greenLight);
110
111   //}
112   Serial.write("<OK,");
113   Serial.print(greenLightStatus);
114   Serial.write(",>");
115 }
116
117
118 void getYellowLightStatus(){
119   int yellowLightStatus = 0;
120
121   //if(digitalRead(lightSwitch)){
122     yellowLightStatus = digitalRead(yellowLight);
123   //}
124   Serial.write("<OK,");
125   Serial.print(yellowLightStatus);
126   Serial.write(",>");
127 }
128
129
```

```
130 void getRedLightStatus(){
131   int redLightStatus = 0;
132 
133   //if(digitalRead(lightSwitch)){
134     redLightStatus = digitalRead(redLight);
135   //}
136   Serial.write("<OK,");
137   Serial.print(redLightStatus);
138   Serial.write(",>");
139 }
140 
141 void updateStatus(){
142   digitalWrite(greenLight,LOW);
143   digitalWrite(yellowLight,LOW);
144   digitalWrite(redLight,LOW);
145 
146   if (distance >= 20){
147     digitalWrite(greenLight,HIGH);
148   }
149 
150   if( distance < 20 && distance >= 10) {
151     digitalWrite(yellowLight,HIGH);
152   }
153 
154   if( distance < 10 ){
155     digitalWrite(redLight,HIGH);
156   }
157 }
158 
159 //============
160 
161 void recvWithStartEndMarkers() {
162     static boolean recvInProgress = false;
163     static byte ndx = 0;
164 
165     char rc;
166 
167     while (Serial.available() > 0 && newData == false) {
168         rc = Serial.read();
169 
170         if (recvInProgress == true) {
171             if (rc != endMarker) {
172                 receivedChars[ndx] = rc;
173                 ndx++;
174                 if (ndx >= numChars) {
175                     ndx = numChars - 1;
176                 }
177             }
178             else {
179                 receivedChars[ndx] = '\0'; // terminate the string
```

```
                recvInProgress = false;
                ndx = 0;
                newData = true;
            }
        }

        else if (rc == startMarker) {
            recvInProgress = true;
        }
    }
}


void parseData() {

    // split the data into its parts
    char * strtokIndx; // this is used by strtok() as an index

    strtokIndx = strtok(tempChars,",");      // get the first part - the string
    strcpy(commandCode, strtokIndx); // copy it to commandCode

    switch (atoi(commandCode)) {
      case 1:
        setOff();
        break;
      case 2:
        setOn();
        break;
      case 3:
        getAutoDetection();
        break;
      case 4:
        getGreenLightStatus();
        break;
      case 5:
        getYellowLightStatus();
        break;
      case 6:
        getRedLightStatus();
        break;
    }
}
```