**Text

Description automatically generated**

**Assignment - Part 2 (of 3)**

Isaac Cutajar (703H)

B.Sc. (Hons) Computing and Business

Study-unit: **Fundamentals of Software Testing**

Code: **CPS3230**

**FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the assignment submitted is my work, except where acknowledged and referenced.

I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).



Isaac Cutajar

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

Assignment - Part 2 (of 3)

CPS3230

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Course Code Title of work submitted

02/01/2023

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date

Contents

[1. Task 1: Specify the system 1](#_Toc123583979)

[1.1 Finite automation 1](#_Toc123583980)

[2. Task 2: Runtime Verification 2](#_Toc123583981)

[2.1 Implementation 2](#_Toc123583982)

[2.2 Video of monitor execution 2](#_Toc123583983)

[3. Task 3: Model Based testing 3](#_Toc123583984)

[3.3 Implementation 3](#_Toc123583985)

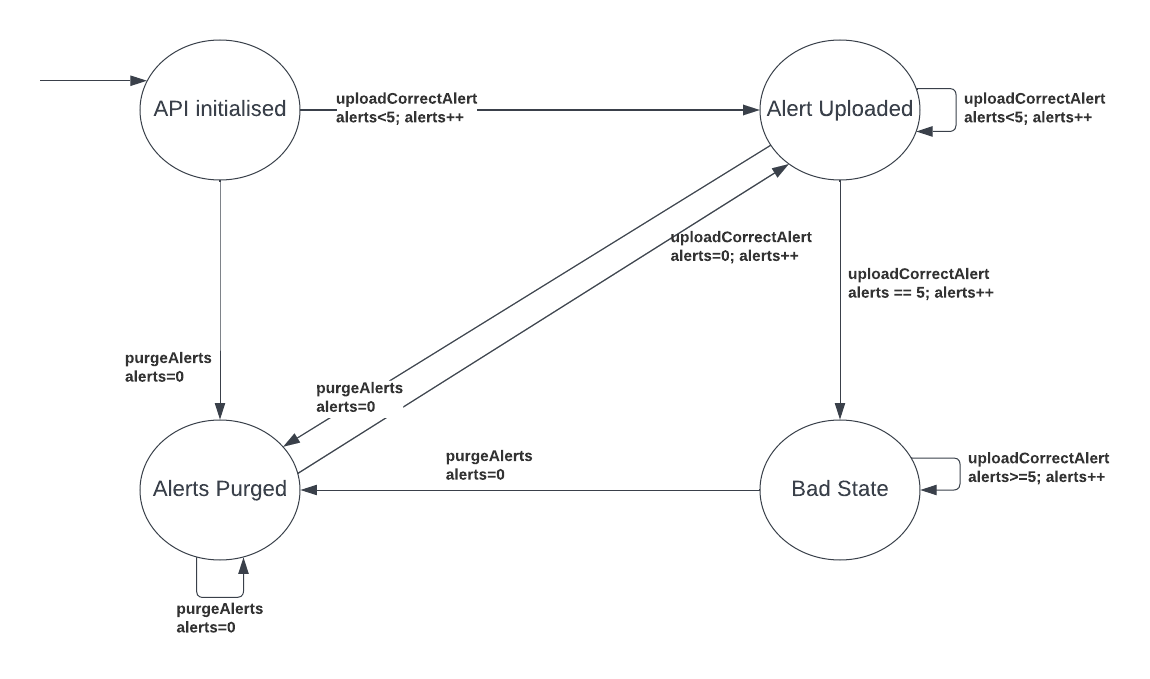
[3.4 Experimentation & Results 4](#_Toc123583986)

[3.5 Video of model-based test suite execution 5](#_Toc123583987)

# Task 1: Specify the system

## Finite automation

The system simulates uploading alerts using API calls.

The model is as follows;

The created automaton looks like this. The API Initialized stage is where the system begins. Firstly, the API for uploading and deleting alerts is initialised. The user can upload alerts or purge them from this point. The alert will move to the Alert Uploaded stage if the user uploads one. The user has the option to either continue submitting alerts or remove those that have already been sent. If more than five alerts are posted in a row, the system will go to a bad state. When alerts are deleted, the system enters the Alerts Purged state. Only after an alert is uploaded can the system move from the state of ‘Alerts Purged’ to ‘Alert Uploaded’.

# Task 2: Runtime Verification

Runtime verification is a technique for checking the correctness of a system while it is running, by monitoring its execution and comparing it to a specification. This can be used to detect errors or deviations from the desired behaviour, and can be applied to various kinds of systems, including software, hardware, and hybrid systems.

## Implementation

In Task 2, using the symbolic automation specified for task 1, a monitor was developed using Eclipse, with .jar extension files (okhttp3 & GSon), to verify the system at runtime. A black box testing approach was taken.

Eclipse was used to implement the FSM model, in Task 1, as part of a runtime verification implementation using LARVA. Properties were defined based on the program’s execution (and therefore the model). If a violation is observed an alert can be triggered. The system drives the monitors, not vice versa.

## Video of monitor execution

<https://drive.google.com/file/d/1tBF8D5VsO9wKihNXHFSPxvfibF63T6io/view?usp=share_link>

# Task 3: Model Based testing

Model-based testing is a technique for designing and executing tests for a system, in which a model of the system is used to generate test cases and test inputs. The model represents the behaviour of the system and defines the expected outcomes of the tests.

## Implementation

In Task 3, using the symbolic automation specified for task 1, a model-based test suite was developed using IntelliJ, with the okhttp3 & GSon dependencies, to verify the system at runtime. A black box testing approach was taken.

IntelliJ was used to implement the FSM model in Task1. A ApiFunctionalityModelTest

class implementing the FsmModel interface model was created to test and to be used as

the place where to transform a graphical interpretation into a programming one. Apart

from this class, an ApiFunctionality class and a ApiFunctionalityStates enum class were created, see Figure 3.1.

Text

Description automatically generated

*Figure 3.1*

The ApiFunctionality class is composed of the following methods:

* int getNumberOfAlerts() – Returns number of alerts.
* boolean isUploaded() -Returns uploaded.
* boolean isPurged() – Returns purged.
* containsAlertType() – Returns alert type.
* boolean containsHeading() – Returns heading.
* containsUrl() - returns url.
* boolean containsImageUrl() - returns image url.
* boolean containsPostedBy() - returns posted by.
* boolean containsPriceInCents – returns price in cents.
* void startWithPurgeAlerts() – deletes alerts.
* void purgeAlerts() – deletes alerts and returns purged and number of responses, if response is of event log type 1 and not null.
* void uploadCorrectAlert() – uploads json alert using and returns GetResponse object.

In the ApiFunctionalityModelTest class, the SUT is the ApiFunctionality class. Each transition was depicted as an @Action, having a guard which checks whether the transition can be performed. This guard checks the current state and any other necessary conditions Upon entering each transition, the SUT is updated and also the model. Then, to make sure that the SUT and model are in the same order, the respective values were stated to make sure that they match and a transition is completed successfully. For example, the void purgeAlerts() method in Figure 3.3.

Graphical user interface, text, application

Description automatically generated

Figure 3.3

## Experimentation & Results

The ApiFunctionalityModelTest class is a test runner, making sure that all states and transitions are being utilised and accessed.

There exist various different test generation algorithms, but for this task a Random Tester approach is taken. This **Random Tester** tests a system by making random walks through the FSM model of the system. The results obtained using this tester can be seen in Figure 3.4. The results show that the Random Tester has full action, state and transition-pair coverage and thus suggests that all states and transitions of the FSM model are being reached. 250 tests are generated.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 3.4

## Video of model-based test suite execution

https://drive.google.com/file/d/1FGEK-sXO2XPP1sjneCTIuRu8DMi434pk/view?usp=sharing

Please note that in this video the test fails, unlike the result seen in Figure 3.4.