

Capstone Project Phase B

Feasibility analysis and performance testing of collision detection algorithms for satellites

23-2-D-17

Supervisor: Mr. Ilya Zeldner

Hallel Weil 316484153 Hallel.Weil@e.braude.ac.il

Omer Shamir 314763970 Omer.Shamir@e.braude.ac.il

[Abstract 3](#_Toc165560666)

[Keywords 3](#_Toc165560667)

[1. Introduction 4](#_Toc165560668)

[2. General Description 5](#_Toc165560669)

[2.1. Background 5](#_Toc165560670)

[2.1.1. The Algorithms 5](#_Toc165560671)

[2.1.2. The Propagator 8](#_Toc165560672)

[2.2. Goals 8](#_Toc165560673)

[2.2.1. Implementing the Algorithms 8](#_Toc165560674)

[2.2.2. Creating a Testing System 8](#_Toc165560675)

[2.2.3. Feasibility Testing and Analysis 9](#_Toc165560676)

[2.3. Users 9](#_Toc165560677)

[3. The solution 10](#_Toc165560678)

[3.1. Algorithms Analysis 10](#_Toc165560679)

[3.1.1. Algorithms Complexity 10](#_Toc165560680)

[3.2. Algorithms implementation 14](#_Toc165560681)

[3.2.1. ANCAS Implementation 14](#_Toc165560682)

[3.2.2. SBO-ANCAS Implementation 15](#_Toc165560683)

[3.2.3. CATCH Implementation 16](#_Toc165560684)

[3.3. Testing system 17](#_Toc165560685)

[3.3.1. Testing Station App 18](#_Toc165560686)

[3.3.2. Tested OBC App 24](#_Toc165560687)

[3.3.3. Communication Protocol and Channels 28](#_Toc165560688)

[3.3.4. Feasibility Testing Environments 30](#_Toc165560689)

[3.4. Research and Development process 31](#_Toc165560690)

[3.4.1. Algorithms Analysis and Implementation 31](#_Toc165560691)

[3.4.2. The Development Process 31](#_Toc165560692)

[3.4.3. Unit Tests and Debug 31](#_Toc165560693)

[3.4.4. Testing the System 31](#_Toc165560694)

[3.5. Development tools 32](#_Toc165560695)

[3.5.1. Development environment 32](#_Toc165560696)

[3.5.2. Languages 32](#_Toc165560697)

[3.5.3. Additional Tools 32](#_Toc165560698)

[3.6. Problems and solutions 33](#_Toc165560699)

[3.7. Feasibility Analysis and Test Results 34](#_Toc165560700)

[3.7.1. SBO-ANCAS Errors Analysis 34](#_Toc165560701)

[3.7.2. Run Time Comparison 36](#_Toc165560702)

[3.8. Results and conclusion 37](#_Toc165560703)

[4. User guide 38](#_Toc165560704)

[4.1. Testing Station App 38](#_Toc165560705)

[4.2. Tested OBC App 39](#_Toc165560706)

[5. Maintenance Guide 40](#_Toc165560707)

[5.1. Testing Station App 40](#_Toc165560708)

[5.2. Tested OBC App 41](#_Toc165560709)

[5.2.1. Installing and Running the Application 41](#_Toc165560710)

[5.2.2. Error Detection and Debugging 42](#_Toc165560711)

[5.2.3. Implementing Changes 43](#_Toc165560712)

[6. REFERENCES: 45](#_Toc165560713)

# Abstract

The project we present is based on algorithms for finding the minimal distance between two objects in space, and the time of occurrence, which will have to work on satellite on-board computer (OBC). The project has two goals, the first goal is to implement the algorithms and deliver them ready to integrate and use for future testing and projects, and the second goal is to create a system for testing the algorithms performance on a satellite OBC or an emulated environment and conduct a feasibility study.

The project is based on Dr. Elad Denenberg's research papers that introduced the algorithms **[**[**1**](#Reference1)**][ ]**.

# Keywords

Satellite, Space debris, Minimal distance, Approximations algorithms, Feasibility test, Collision detection, Algorithm testing, Distance approximation, Orbiting objects, Satellite on-board computer.

# Introduction

One of the things that concerns satellite operators during a mission, is the risk of colliding into other objects. There is a significant amount of space debris orbiting Earth, including decommissioned satellites or parts of them, rockets, and other human-made objects, and there are also natural celestial bodies in space we should be aware of like asteroids. In order to avoid these threats, we start by keeping track of them, then we identify possible collisions and recalculate our path. Doing so is done by calculating the future orbit of 2 object and finding the point in time where the distance between them is the smallest, this time is called **Time of Closest Approach** **(TCA)** and the TCA and the respective distance is the values we are looking for.

With the increasing number of objects in Earth orbit, around 27,000**[**[**3**](#Reference3)**]** and the shift to cluster of smaller satellites instead of a single big one **[**[**1**](#Reference1)**]** the cost of calculating the orbit of objects and finding the TCA for our satellite is only growing. To solve this problem a few cheaper algorithms were developed. The algorithms are supposed to be cheap and fast enough to run on the satellite’s own ***on-board computer (OBC)***. These algorithms have not been tested and we need to prove the task feasibility, to implement the algorithms and to show the calculation time and memory requirements in an environment with limited calculation power and memory simulating an actual satellite on-board computer. We are working with Dr. Elad Denenberg, who created the **Conjunction Assessment Through Chebyshev Polynomials (CATCH)** algorithm **[**[**1**](#Reference1)**]** and the **SBO-ANCAS** algorithm[ ]. Dr. Elad is working on creating an autonomous satellite and as part of his work he need our help. An autonomous satellite needs to calculate the possible collisions by itself and for doing that a fast algorithm for finding the TCA is needed, faster calculations are possible using approximations like the algorithms CATCH **[**[**1**](#Reference1)**],** SBO-ANCAS[ ] and **Alfano\Negron Close Approach Software (ANCAS)** **[**[**2**](#Reference2)**]**. These algorithms were never tested on an actual satellite on-board computer, and proved fitting to run on an autonomous satellite and this is we come in.

# General Description

## Background

### The Algorithms

In this project we implemented and tested three algorithms, all three can be used to calculate the TCA. The following is a description of the algorithms.

#### ANCAS

The first algorithm, ANCAS **[**[**2**](#Reference2)**]** uses cubic polynomial as an approximation of a function over an interval. Given n points in time and the respective location and velocity vectors for 2 objects, we can find the TCA by:

1. ANCAS on n points, (the original algorithms description can be found at [[2](#Reference2)])

**Input**:

**Output**:

**for** each set of 4 points **do:**

Map the time points to on the interval

Calculate using **Eq.([5](#eq5)/**[**2**](#eq2)**)** with the points

Fit cubic polynomial to according to **[**[**2**](#Reference2) **,Eq.1f-1j]** over

Find the cubic polynomial real roots in the interval

Fit cubic polynomials for in the interval

**for** each root **do:**

calculate the distance using **Eq.([6](#eq6))**

**if** **:**

**end**

**end**

**end**

The cubic polynomial coefficients calculations described in article **[**[**2**](#Reference2)**]**.

Finding the roots of a cubic polynomials can be done by solving the 3rd degree equation and we will find between 1 to 3 real solutions. There is a problem with the algorithm, the point in time must be relatively close because the algorithm can only find up to 3 extrema points, so working on a large time interval means we can miss possible points and even miss the actual point of the TCA. Because the root finding can be done fast using the 3rd degree equation the algorithm run relatively fast but the result can be inaccurate.

#### SBO-ANCAS

The second algorithm, SBO-ANCAS **[**[**2**](#Reference2)**]** is based on the ANCAS algorithm, still using cubic polynomial as an approximation of a function over an interval. But uses additional points to get better results. Given an initial set of n points in time, the respective location and velocity vectors for 2 objects, tolerance in time and tolerance in distance we can find the TCA by:

1. SBO-ANCAS on n points, (the original algorithms description can be found at **[**  **]**)

**Input**: ,,

**Output**:

**for** each set of 4 points **do:**

**Do**

Map the time points to on the interval

Calculate using **Eq.([5](#eq5)/**[**2**](#eq2)**)** with the points

Fit cubic polynomial to according to **[**[**2**](#Reference2) **,Eq.1f-1j]** over

Find the cubic polynomial real roots in the interval

Fit cubic polynomials for in the interval

**for** each root  **do:**

calculate the distance at  using **Eq.([6](#eq6))**

**if** **:**

**end**

**end**

Sample and at using a propagator

**While**  OR

**end**

The cubic polynomial coefficients calculations described in article **[**[**2**](#Reference2)**]**.

Finding the roots of a cubic polynomials can be done by solving the 3rd degree equation and we will find between 1 to 3 real solutions. In each iteration after finding the minimum point we use the propagator to sample the location and velocity vectors at , we use the new and more accurate values and create a polynomial to find a more accurate minimum distance and so on until we reach the desired tolerance. This algorithm can give the best results, we can get the same results as checking every point with a small time-steps if we use small enough tolerance but with high cost in run time. SBO-ANCAS have an additional loop in each iteration and sampling points with the propagator is an expensive operation.

#### CATCH

The third algorithm, CATCH **[**[**1**](#Reference1)**]**, uses **Chebyshev Proxy Polynomial (CPP)[[1](#Reference1)]** to approximate the functions. The CPP can give more accurate result, depending on the degree of polynomial we want to use. We can choose high enough degree to get the size of error we want. The algorithm work on time interval from 0 to , each iteration searches the minimal distance in an interval with size . The degree of the CPP is part of the algorithm input and appear as N.

1. CATCH the original algorithms description can be found at **[**[**1**](#Reference1)**, algorithm 2]**

**Input**: 

**Output**: 









**While** **do**:

Fit CPP  with order N to  according to **[**[**1**](#Reference1) **, Eq.15]** over the interval 

Fit CPP with order N to  over the interval 

Find the roots of 

**for** each root  **do:**

calculate the distance  at  using Eq.([6](#eq6))

**if** **:**



**end**

**end**

****

**end**

The algorithm needs N+1 points in time in each Gamma interval in order to create CPP of order N. After calculating the CPP coefficients we can use them to create a special NxN matrix called the companion Matrix **[**[**1**](#Reference1)**,Eq.18]** and the eigen values of this matrix are the polynomial roots. Using the roots, we found and creating CPP for  we can calculate the minimal distance in each interval and eventually the TCA and respective distance in . The problem with CATCH is the cost of finding the roots, which is the cost of finding eigen values for an NxN matrix, to deal with it, Dr. Elad describe in his article**[[1](#Reference1),part 4]**  that we can get sufficient results for both runtime and error size, using degree of 16 for the polynomial. Using a constant degree give us deterministic run times and the size of the error is small enough for the required result.

### The Propagator

Using the current location and velocity of an object in orbit (for example our satellite) and a point in time (for example ten minutes from now) the Propagator can calculate the object location and velocity in the given point in time. The propagator uses a forces model to find the future orbit, this force model is different between different propagators and can affect the result’s accuracy and the calculation time. In this project we use a propagator called **Standard General Perturbations Satellite Orbit Model 4(SGP4)** **[**[**10**](#Reference10)**]**, SGP4 is old and famous propagator used for research that known for its fast run time and there are many available implementations we can use **[**[**4**](#Reference4)**]**. SGP4 get the object data using format called **Two Lines Elements (TLE)** which consist of two lines of data, including the object location, velocity, the corresponding time, the average number of revolutions per day (Mean Motion) and more.

We used the propagator for two task, the first is creating the data for each of the algorithms runs. Given a set of TLE from the user we can create a set of point in time for two satellites and run the algorithms with it. The second task is using a propagator as part of the SBO-ANCAS algorithm. SBO-ANCAS needs to sample new points as part of the algorithms so a propagator is needed.

## Goals

In this project we have three main goals.

### Implementing the Algorithms

Our first goal is to implement the algorithms themselves and doing so while considering the environment the algorithm will have to work on. Until now SBO-ANCAS and CATCH were only implemented in MATLAB as part of the initial article and testing [ ][ ]. To work on a satellite OBC well the algorithms need to run efficiently on various systems and computer boards. We started implementing the algorithms in the first part of the project as a feasibility proof for our project and completed the implementation in this part.

### Creating a Testing System

To test the feasibility of running the algorithms on satellites OBC we needed a system fitting for running test and collecting data and results. We needed to run the algorithm on a dedicated system or emulator with a given set of data and parameters as input, to get the output and run time and to save the results and test related parameters in our data set in order to collect enough data on the algorithms expected run time and accuracy in different scenarios. The Testing System needed to be flexible enough to run the algorithms on different machine and environments and manage and collect the data well. We needed the system both for running the feasibility analysis ourself and for leaving it for DR. Elad to use for his research.

### Feasibility Testing and Analysis

The last part of our project is to conduct a feasibility analysis using our system. We needed to run our system with different types of inputs, different types of algorithms and different parameters for each algorithm. TBD

## Users

For our system we have two types of users, the first is DR. Elad and his associates wanting to test the algorithms against an emulator for now and a real satellite’s OBC in the future. The second type of user is us, wanting to run a feasibility test as part of our project, and any possible follow up team that will take part in the ongoing effort of creating and proving the feasibility of the autonomous satellites. (there are already other teams working in their projects toward this goal, that might need to use our system or expend it).

# The solution

## Algorithms Analysis

### Algorithms Complexity

#### ANCAS Time Complexity

In ANCAS **[**[**2**](#Reference2)**]**, for each set of 4 data points we need to create 4 cubic polynomials, one for the relative distance derivative and 3 for the relative distance X, Y, Z. each cubic polynomials required coefficients calculation which consist of 4 equations **[**[**2**](#Reference2)**,Eq 1f-1j]**, meaning the complexity of finding the polynomial coefficients is constant. To map the time points to the interval [0,1] we use a simple calculation for each point **[**[**2**](#Reference2)**]** 4 times, one for each point.

Finding the solution for a 3rd degree equations is quite simple, using a given formula with a constant run time we get between 1 to 3 real result.

For each of the roots we found, we calculate the distance using **Eq.([6](#eq6)),** and check if we found a smaller distance. In the worst case we check 3 times.

Meaning for each set of 4 data points the complexity is (where k is a constant number):



Calculating the complexity for finding the TCA over **n** data points means we check the first 4 points and for each iteration after that we use the last point from the previous iteration as the first points meaning we need 3 new points, so we need to do  iterations.

The complexity of running ANCAS on **n** data points is:



#### SBO-ANCAS Time Complexity

In SBO-ANCAS **[** **]**, we are going over a set of initial points, the outer loop check the first 4 points and for each iteration after that we use the last point from the previous iteration as the first points meaning we need 3 new points, so we need to do  outer iterations.

In the inner loop we run until we reach the desire tolerance in distance and time, thus the number of inner iterations depends on the size of tolerance in distance, the size of tolerance in time, the error of the polynomial approximation, the change in relative distance in time between the 2 objects and the distance between the initial time points. For each inner iteration we use the propagator to sample a single point in time.

Let’s start by looking at the tolerance in time condition for the inner loop, say we have 4 time points, , with the initial distance between 2 time points of , and tolerance in time . To get to the desired tolerance we need the distance between to the other points to be smaller than . which means that at the last iteration we get:

.

To find the worst-case scenario for the number of iterations we need to consider the smallest possible decrement in the total time interval per iteration. In the following example we can see that theoretically there is no limit to how many iterations we get.

We start with a set of 4 points, :

And so on.

And if we look at the interval size in each step:

And we continue until:

We expect the distance between 2 points, , to be bigger than the tolerance and with a small enough we get:

Practically that not the case because there is a limit on how many small numbers we can fit between any set of 2 initial values, depending on the value of , the specific implementation and the precision of the variables. For example, if we use an IEEE 754 double-precision variable, the smallest possible value is about so we will get a large but final number of iterations.

Let’s look at the tolerance in distance, we compare two values of the distance in the same point in time. The first is the value we got from the polynomial approximation and the second is the value we got from the propagator. The different is because the error of the polynomial approximation. The closer the points in time will be, the smaller the change in distance will be and we can expect smaller errors. So, with a small enough time step we will reach the desired tolerance.

#### CATCH Time Complexity

In CATCH**[**[**1**](#Reference1)**]** algorithm we iterate through the number of time points in our external loop, 

Where {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>t</mi><mrow><mi>m</mi><mi>a</mi><mi>x</mi></mrow></msub></mstyle></math>"} is the is end boundry in the time range where we're looking for mininal disdance, and equal to half of the smaller revolution time of the object **[**[**1**](#Reference1)**,part 4]**, The value of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mstyle></math>"} is the order of the polynomial, while we can change the chosen value of N, it was determined that {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>N</mi><mo>=</mo><mn>16</mn></mstyle></math>"} give sufficient results.

Inside the loop we're doing the following steps:

1. Fit the CPP of order {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mstyle></math>"} to {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mover><mi>f</mi><mo>&#x2D9;</mo></mover><mfenced><mi>t</mi></mfenced><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>x</mi></msub><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>y</mi></msub><mo>,</mo><mo>&#xA0;</mo><msub><mi>p</mi><mi>z</mi></msub></mstyle></math>"} over each interval of points:

Assuming the arithmetic operations we use are basic operation done in time complexity of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}, we calculate the Chebyshev polynomials**[[1](#Reference1)]**. We'll iterate through {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mstyle></math>"} points, which is a constant in our case, meaning that the time complexity will also be constant. Each iteration requires us to sample a new time point which will be our input parameter x, calculating the interpolation matrix with size of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mfenced><mrow><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mrow></mfenced><mfenced><mrow><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mo>+</mo><mn>1</mn></mrow></mfenced></mstyle></math>"}which is also constant.

The complexity of this step is: {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mfenced><mrow><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mfenced><mrow><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced><mo>&#xB7;</mo><mi>O</mi><mfenced><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub></mfenced></mrow></mfenced><mo>+</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mrow></mfenced><mo>&#xA0;</mo><mo>=</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}

1. Finding the roots for {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msub><mi>P</mi><mi>f</mi></msub></mstyle></math>"} will consist of calculating the companion matrix with a size of {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><msup><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mn>2</mn></msup></mstyle></math>"} and finding the eigen values, using the complexity of matrix multiplication for this step, the complexity will be {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msup><msub><mi>N</mi><mrow><mi>d</mi><mi>e</mi><mi>g</mi></mrow></msub><mn>3</mn></msup></mfenced><mo>&#xA0;</mo><mo>=</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mn>1</mn></mfenced></mstyle></math>"}, rescaling each eigen value to the actual coefficient value also takes constant time.
2. For each time point we'll calculate in our interval we'll check if we found a new minimal distance, if we did, we'll update the minimum distance and the time of occurrence. This step also has a constant time complexity.

It means that the only inputs that determines our time complexity are the values of how long each interval time, and how long in the future we want to look it,

meaning the complexity equals the number of different time-points we measure, which is:

#### Space complexity

The space complexity of the algorithms is the same. SBO-ANCAS\*, ANCAS and CATCH uses a constant number of internal variables to help with the calculations. Because our task is finding a minimum, we only need one variable to store the current minimum without any dependency for the input size. We also use some internal variables representing the polynomial and other related logics. The only memory that is related to the size of the input is the input itself. The input consists of 2 location vectors, 2 velocity vectors and the time point value for each time point in our data set, so we can see that the size of memory the input uses is linear to the number of points we need to test. We get constant space complexity for the algorithms themselves and linear to the number of points for the input:

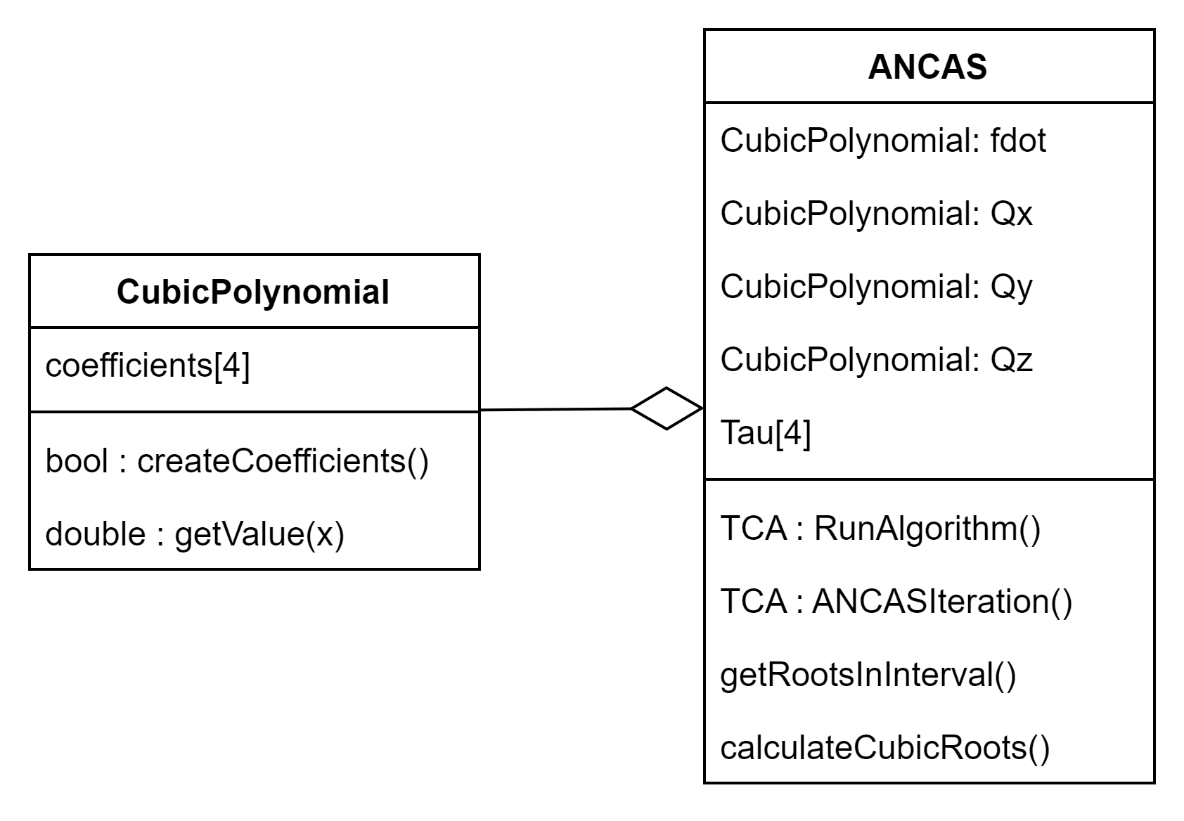
\*For SBO-ANCAS, we assume that the propogator uses a constant amount of memory, but this may not always be the case.

## Algorithms implementation

### ANCAS Implementation

ANCAS implementation is pretty straight forward, there are no inner loop or complicated algorithms in use here, we only need to find the roots of a cubic polynomial, and it can be done using a formula. We kept the implementation as simple and straight forward as possible, only taking out the code for each iteration logic, including fitting the polynomial and finding the roots, into a different function so we can reuse the code for SBO-ANCAS. We created a class representing a cubic polynomial with functions for creating the coefficients and for getting a value at a point x. we created a function for finding the roots using the cubic polynomial formula and created unit tests that check the roots finding for a polynomial with 0 to 3 real roots in range.

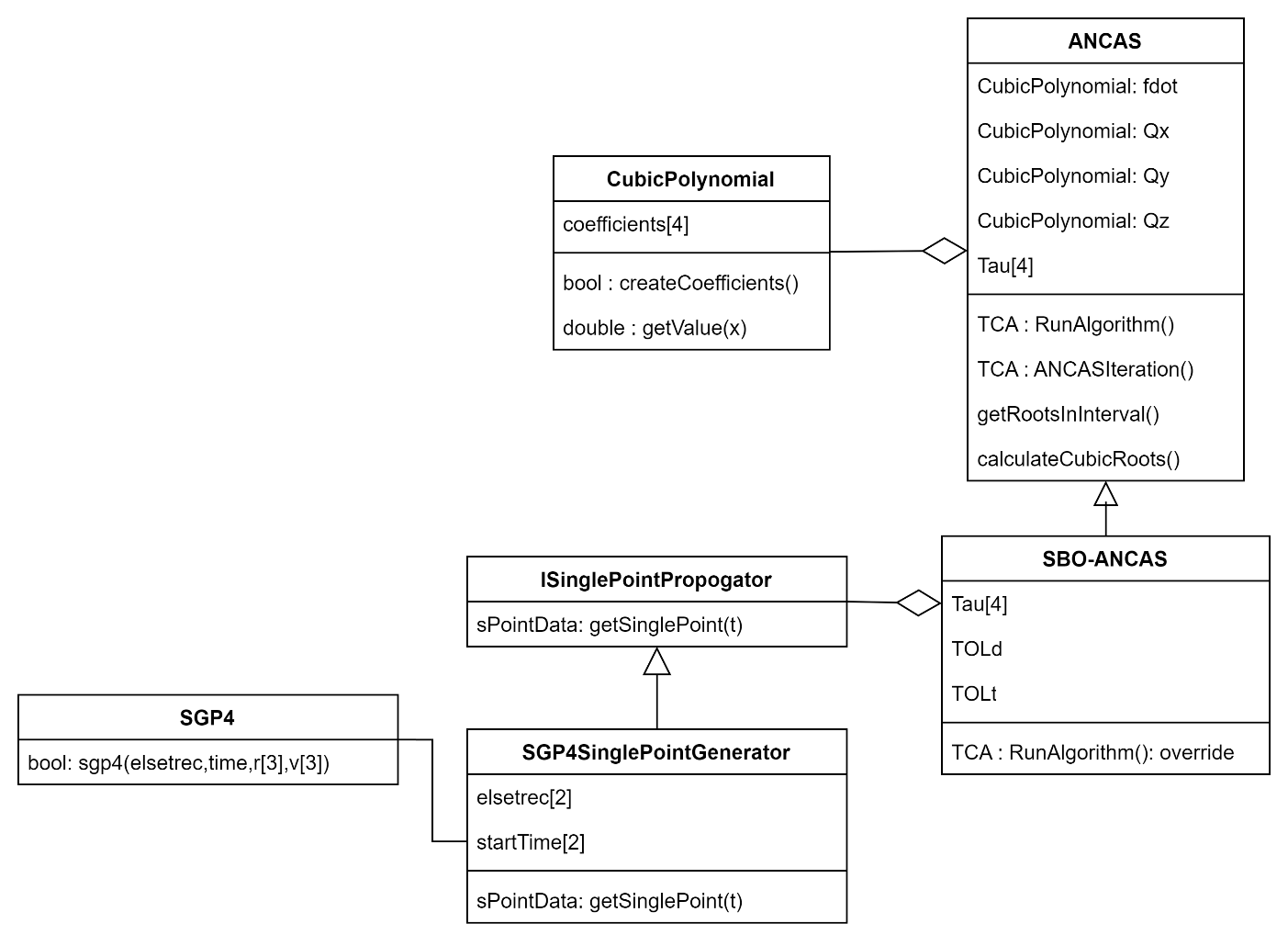
1. Class diagram for ANCAS. Including a function for calculating the roots of a cubic polynomial used by a function for getting the roots in the interval 0,1. The function for ANCAS Iteration return the found minimum and time, used for a single ANCAS iteration.



### SBO-ANCAS Implementation

SBO-ANCAS acts similar to ANCAS in every iteration, initialize the polynomials, finding the roots and so on. To avoid rewriting the same code we inherited ANCAS and only needed to override the RunAlgorithm function. We added an interface for the propagator SBO-ANCAS uses, because we only get a single point in time every time we called it SinglePointPropogator. We implemented the interface using SGP4 and used it for our testing. Additionally SBO-ANCAS needed the tolerances in both time and distance.

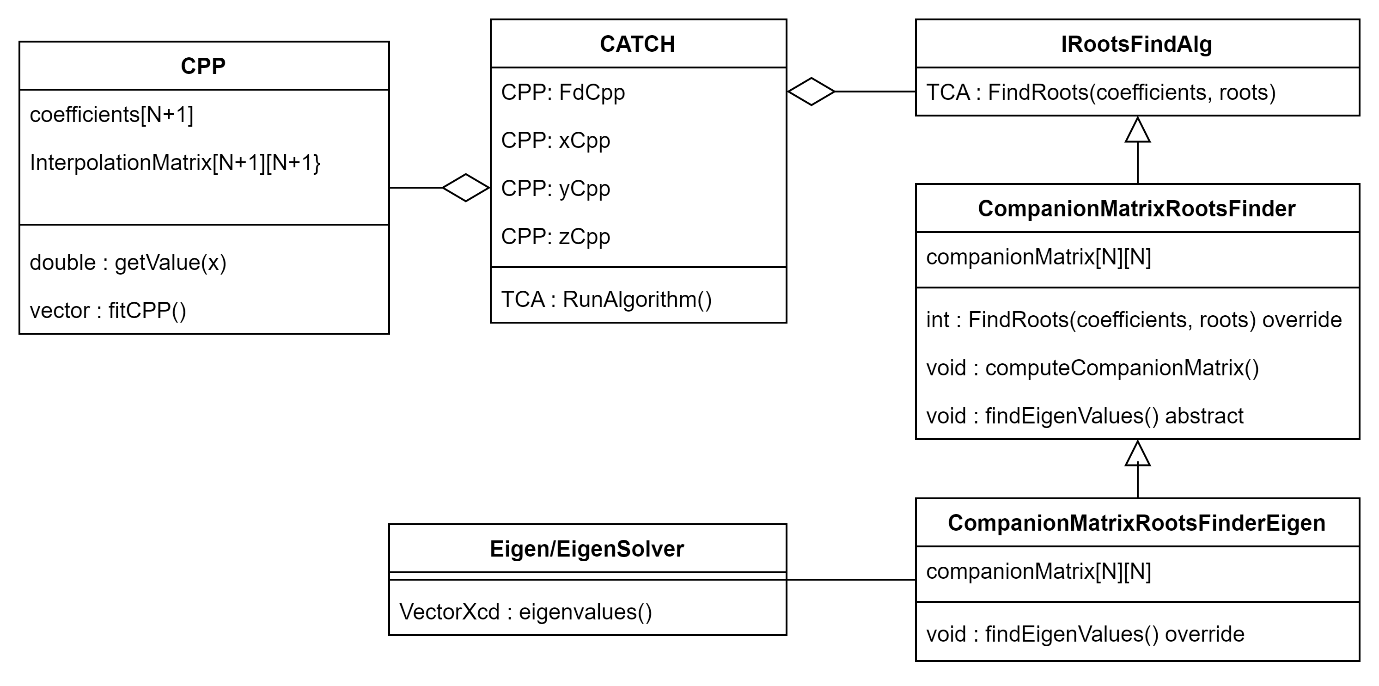
1. Class diagram for SBO-ANCAS. Including the Propagator interface and implementation and the tolerances.



### CATCH Implementation

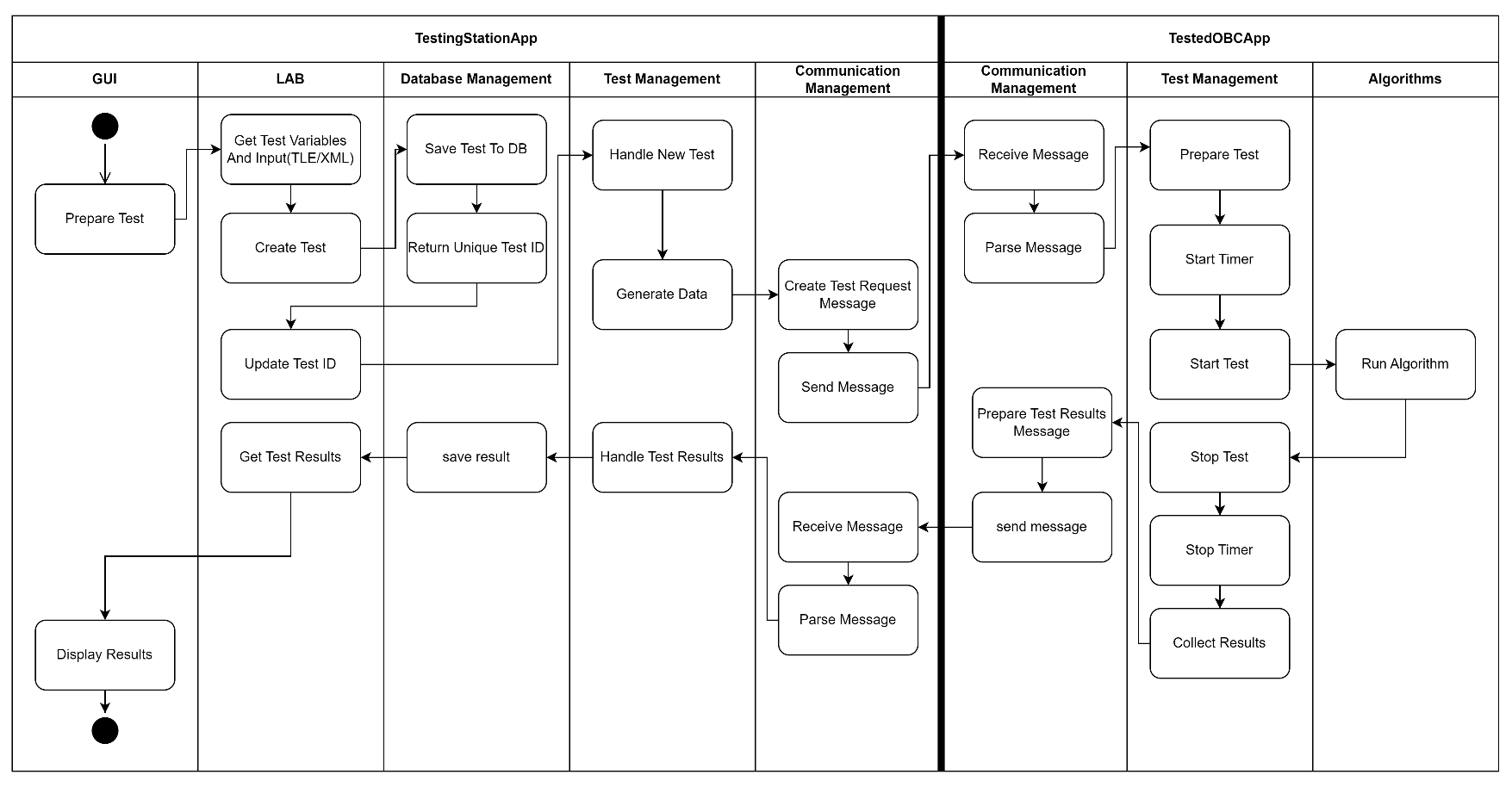
CATCH implementation required implementing the Chebyshev Proxy Polynomials (CPP) class, with function for calculating the polynomial coefficients and to get the value at a point x. we needed the freedom to use different variations of the roots finding to check different libraries so we separated the root finding problem into a different interface. The CATCH class uses 4 CPP, for Fd,x,y,z, additionally it uses the Rootfinder interface to get the polynomial roots in each step. We implemented the CompanionMatrixRootFinder based on the algorithm described in the CATCH’s article [ ] , and we tried two libraries for finding the Eigenvalues of the Companion Matrix. We implemented using Eigen and Armadillo. Unfortunately, the Armadillo library is quite heavy (while using the library the code uses around 400MB) and its too much for the satellite’s OBC so we removed the Armadillo implementation.

1. Class diagram for CATCH, including Rootfinder interface and implementation.



## Testing system

Looking at the full application, the process of creating and running a test starts from the user, the user go to the Test Creation page and create a test, filling all the necessary fields, after that the GUI managers collect the input and call the Lab to create a test, the Lab generate the data, save the test and give it to the test manager who forward it to the Tested OBC App via the communication channels. The Tested OBC App take the input, run the algorithm and send back the results who go all the way back to the tests results page and displayed to the user.

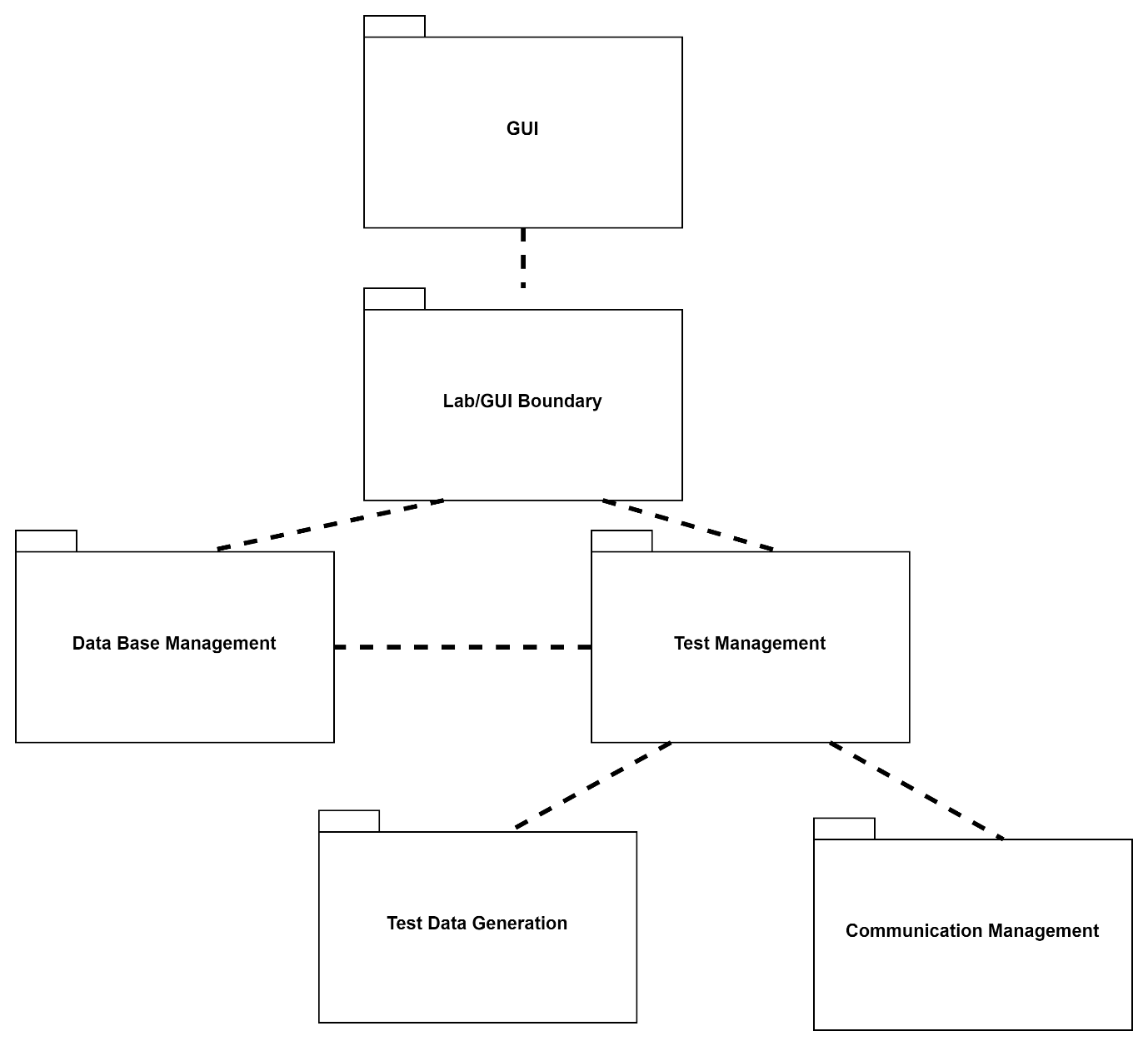
1. Top view activity diagram for running a test, with the full 2Apps system

### Testing Station App

Package diagram and top-down explanation of the system, class diagram for each package with explanations. Activity for running a test

#### App Structure and Architecture

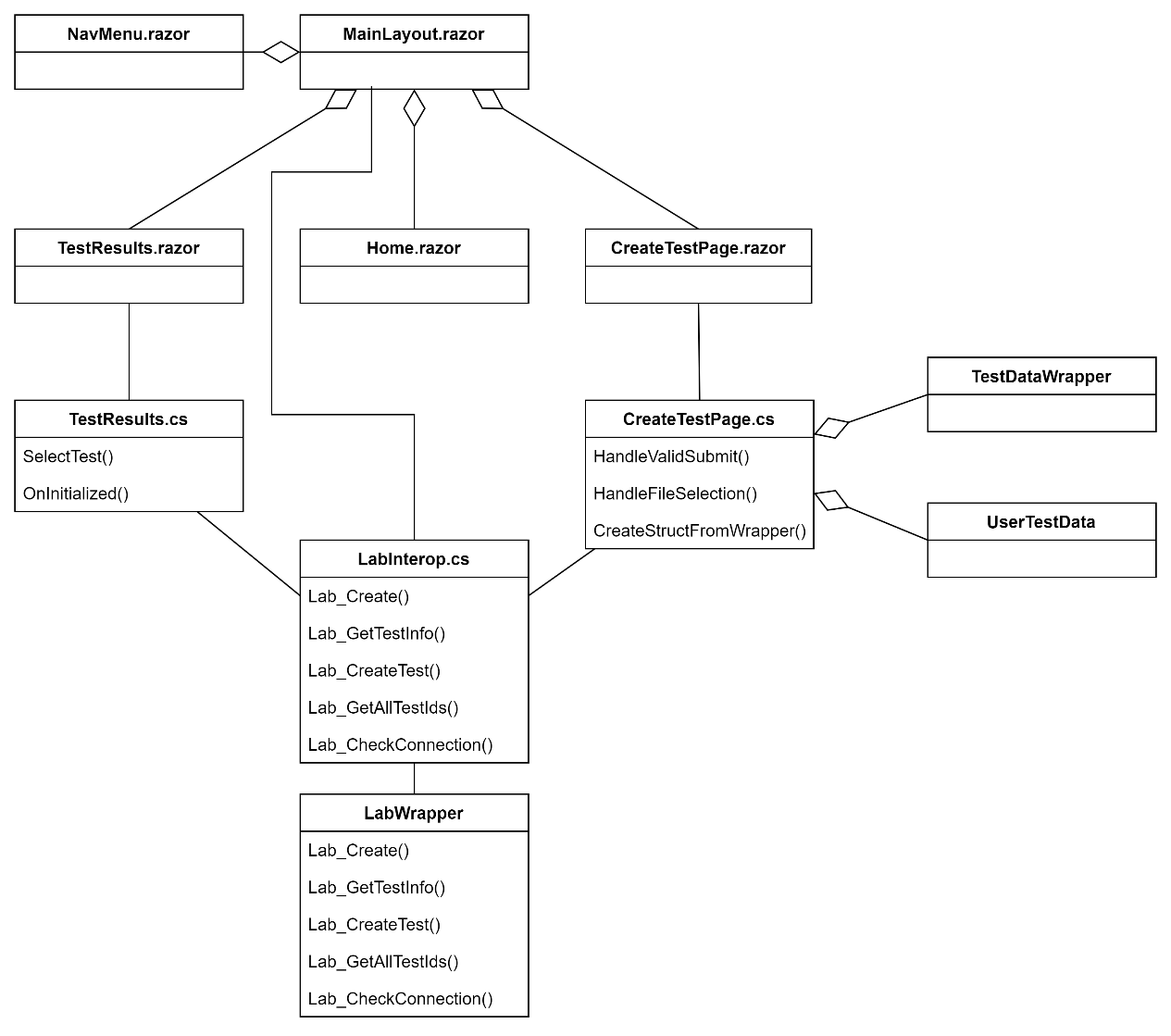
1. The Testing Station App Package Diagram



**GUI Package**

The GUI package include all the application windows (or pages) and their managers. This is where every test starts, the user can go to the test creation page and create a test. After that he can go to the test results page and watch the test results. We have the main layout, holding all the application pages and the top bar(with connection status + links) and the navigation bar. We have 3 pages, the home page with no manager and the test creation and test results pages, each with its own manager. The test creation manager uses 2 data structs, the TestDataWrapper arrive from the GUI and the manager handle the input and place in in the internal application test data structure, the UserTestData struct. Additionally the manager have a function for handling files as input. The test results manager display the full tests list on start and each test information on select. All the interface with the rest of the app is done using the LabInterop static cast, setting the API against the LabWrapper(part of the Lab Package).

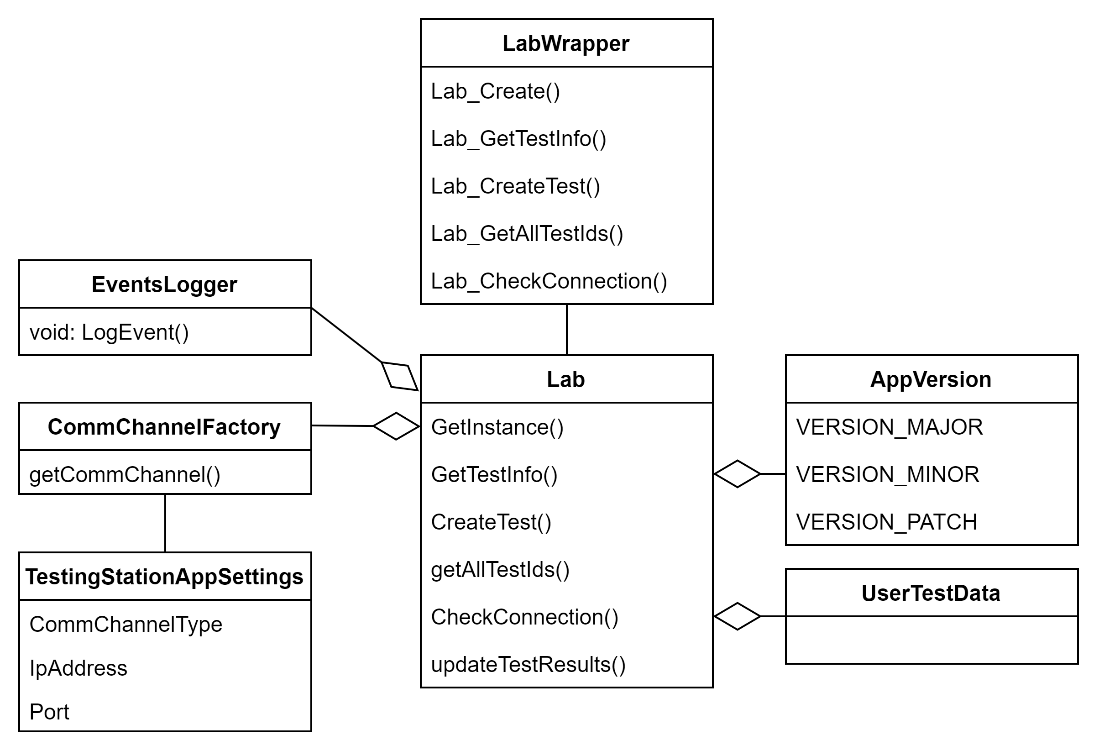
1. GUI Package class diagram.



**Lab/GUI Boundary Package**

The Lab Package is the main package of our application, the Lab Wrapper is the boundary between the GUI and the rest of the application, simply making the Lab class functions available withing the GUI. The Lab class is the main class of the application, managing the rest of the application initialization and use. We use the Comm Channel Factory and the Settings file to initialize the communication, initializing the database and the Logger and so on. Additionally, the Lab holds and manage the application capabilities and processes.

1. Lab Package class diagram.



**Database Management Package**

In this Package we have two classes, the sqllite3 implementation we used and the manager we created, wrapping the sqlite and implementing the creation and management of our tables and data.

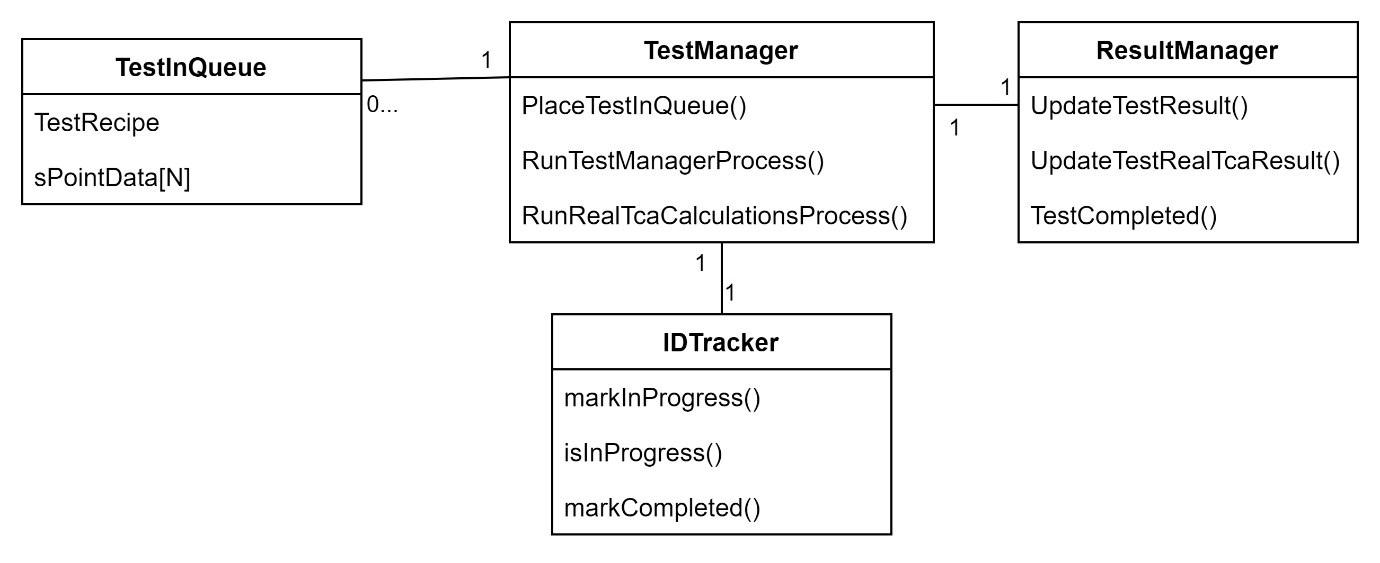
1. Database Management Package class diagram



**Test Management Package**

The package manages the tests and results, there are 2 parts for each test, testing the algorithm and finding the real TCA. The test manager has to asynchronized threads each handling one of the tasks, the Lab call place test in queue when creating a new test and the new test is place inside 2 queues, one for each thread. Each of the threads do its task and update the test manager with the results. Using the shared ID Tracker the thread that completed its task last update the test manager that the test was completed and free any used memory. We use mutex and a “Safe Queue” (synchronized queue) to manage access to any of our shared objects (queues, tracker and shared memory).

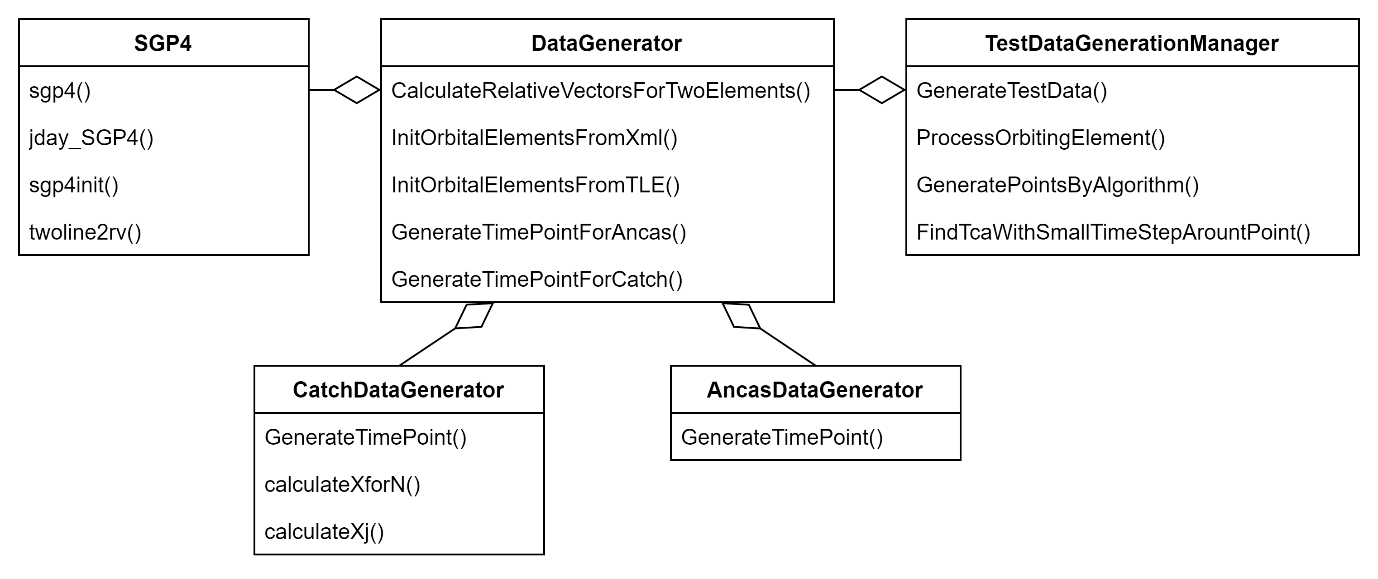
1. Test Management Package class diagram.



**Test Data Generation Package**

The Test Generation package has a central class call the Test Data Generation Manager, using the Data Generator it wrap complicated or sets of action into a single function we use in the rest of our app. The Data Generator do the actual work of initializing the objects, calculating the time points and using SGP4 to generate the data.

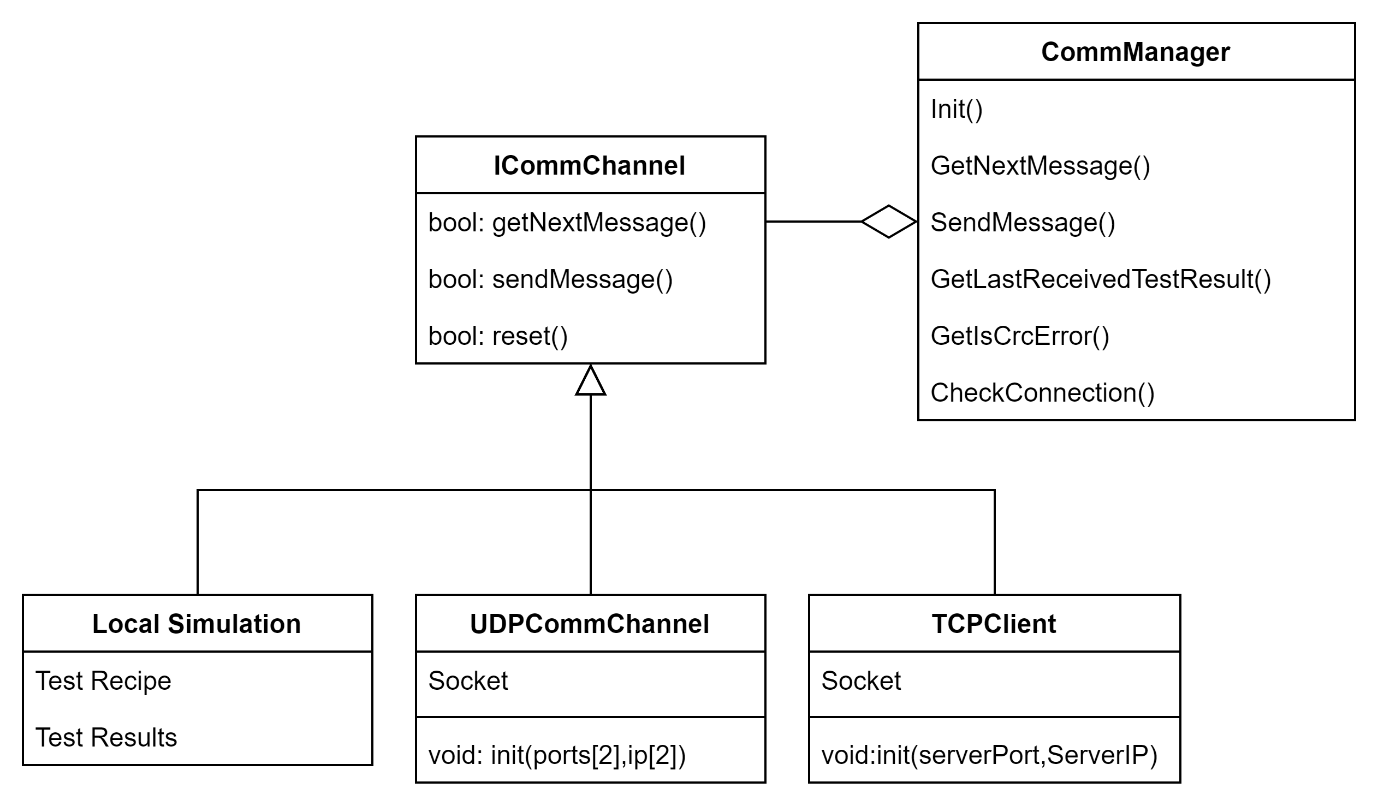
1. Test Data Generation Package class diagram.

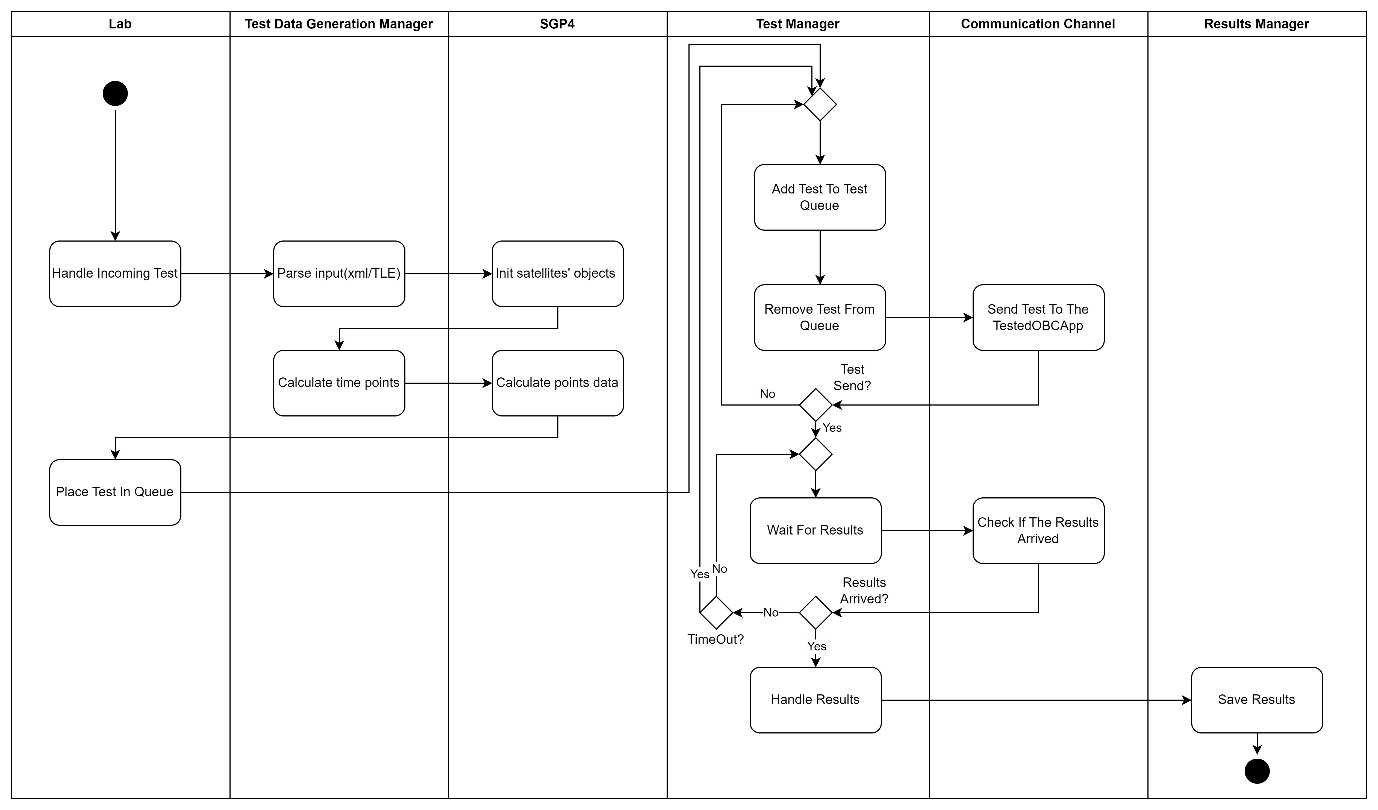


**Communication Management Package**

The package handles anything related to the communication with the Tested OBC App. Using a Comm Channel we received when initialized by the Lab, the manager send outgoing massages and collect and parse incoming messages. We have a few implementations of a Comm Channel including the Local Simulation one.

1. The Communication Management Package class diagram.



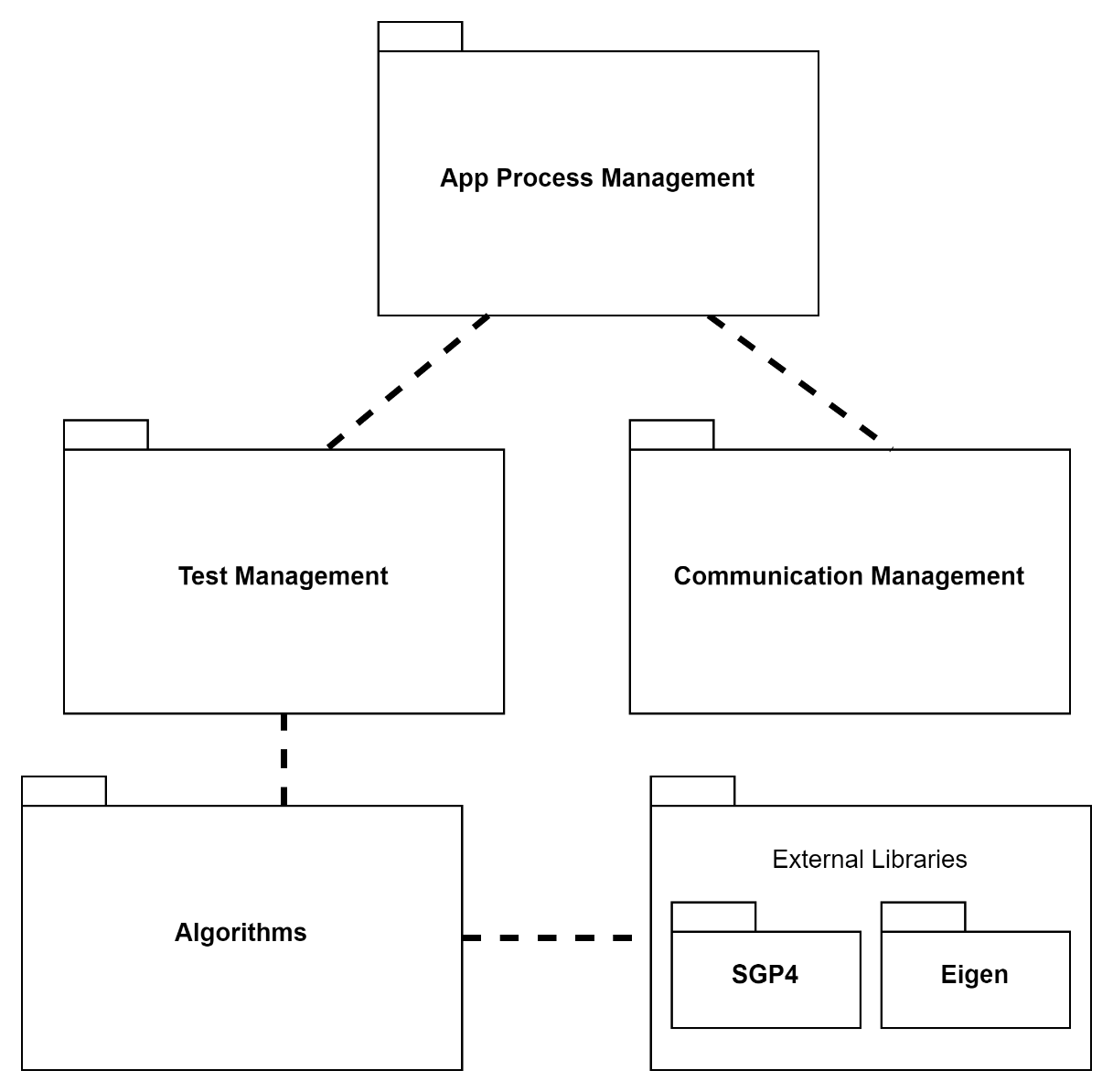


### Tested OBC App

The Tested OBC App works around messages from the Testing Station App, we wait for an incoming message, get the Test Recipe and Test Data from the message, run the algorithm and return the results and run time. When not running a test the Tested OBC App waits for the next message.

#### App Structure and Architecture

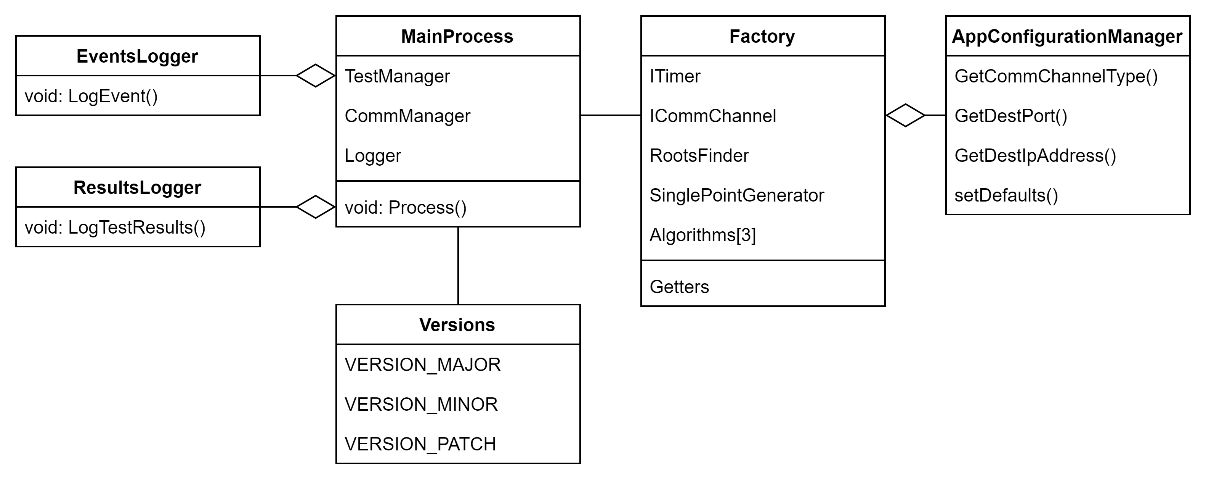
1. The Tested OBC App Package Diagram



**App Process Managements Package**

The App process management package handles the main app process, calling the Communication Management and checking for incoming messages, starting tests using the Test Managements package and handling anything else related to the App creation and process.

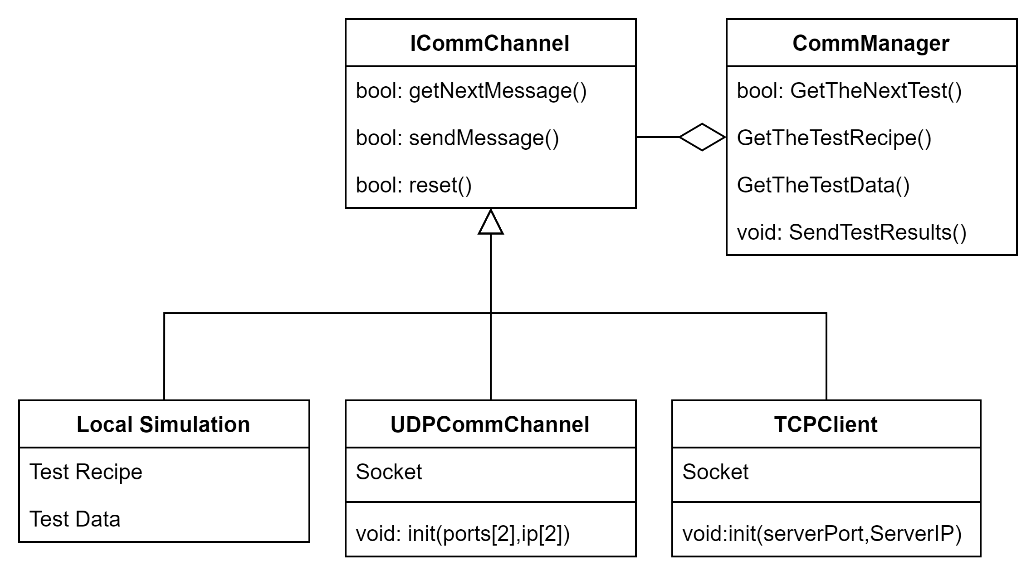
1. The App Process Managements Package class diagram.



**Communication Management Package**

The Communication Management package handles anything related to the communication with the Testing Station. The Factory create the Comm Channel based on the App Configuration and the Comm Manager handle incoming messages, parsing the messages and checking for errors. Additionally, the Comm Manager send the outgoing Results Message based on results set it receive.

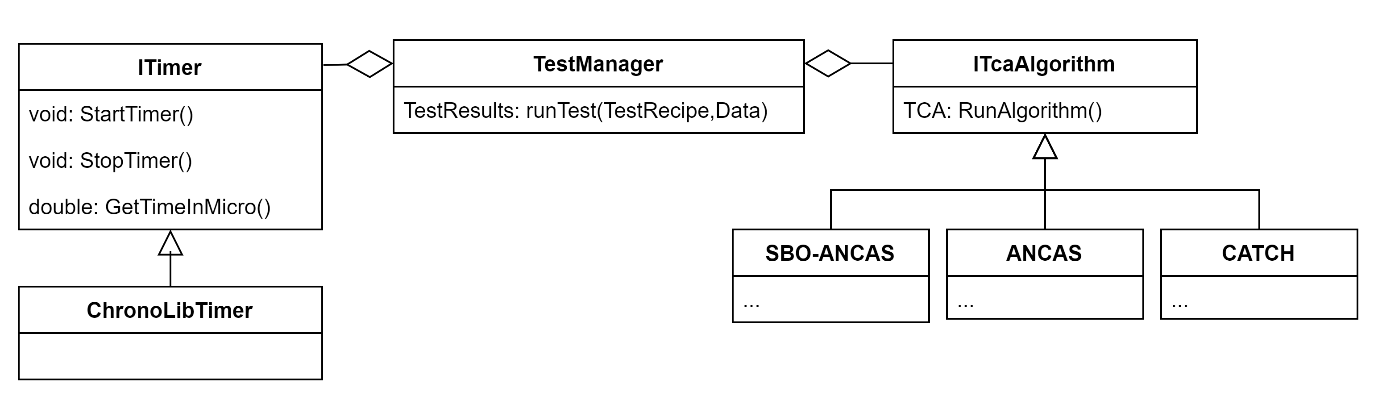
1. The Communication Management Package class diagram.



**Test Management and the Algorithms Packages**

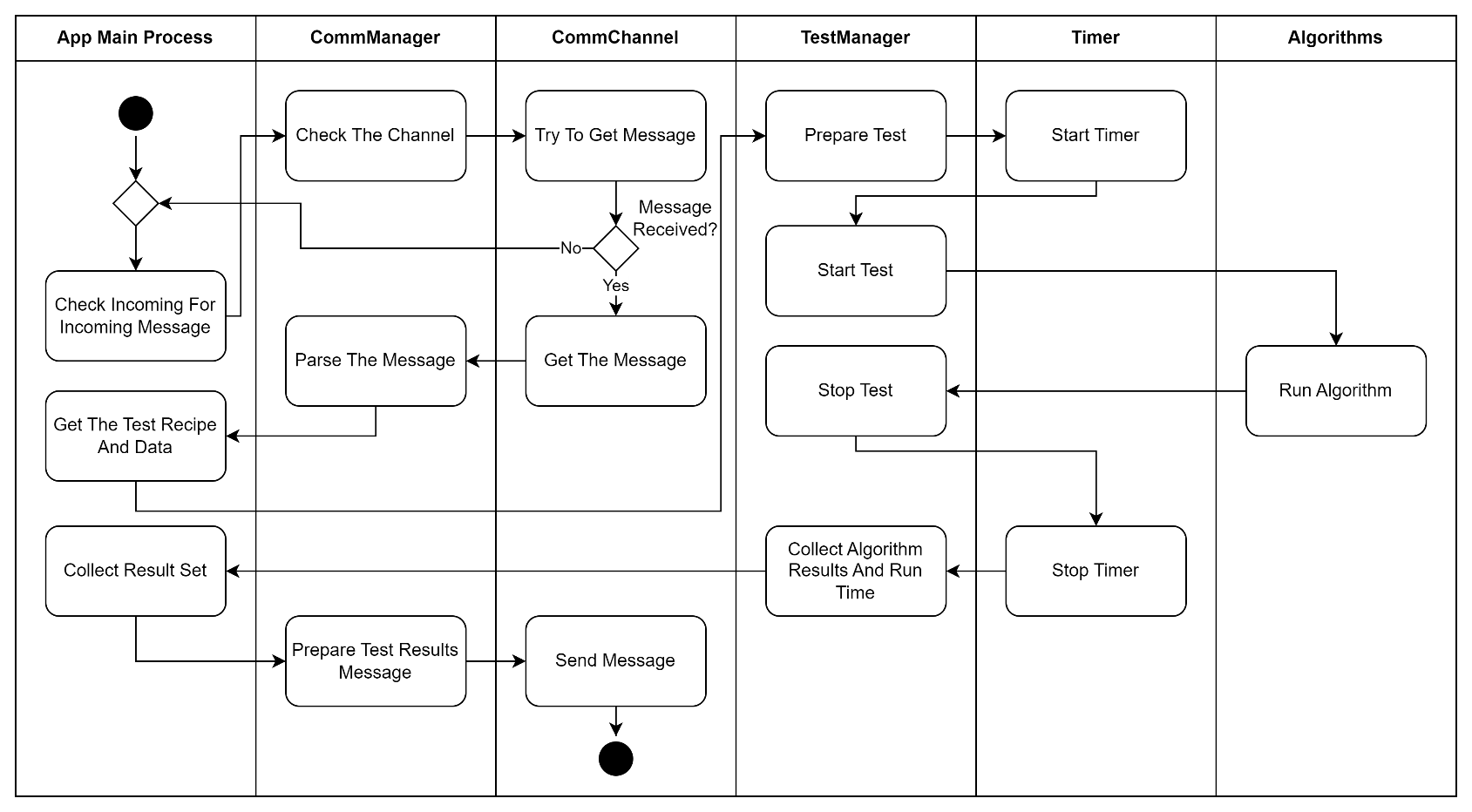
The Test Management package handles the test itself, after receiving a Test Recipe and a set of points as the Test Data, the Test Manager uses the Factory to get the required algorithms object, initialize to the correct degree or with the correct roots finding algorithm based on the Recipe. After receiving the Algorithm, the Test Manager start the timer and run the algorithm. After the algorithm completed the call the Test Manager stop the timer, collect the run time, algorithm output and the test data into the Test Results Set. Additionally, if the test should run over a few additional iterations the Test Manager run the algorithm again and return the Test Results at the end. The Algorithms package contains the Algorithms implementations and variations.

1. The Test Management and the Algorithms Packages class diagram.



#### Running A Test - Activity Diagram

The Tested OBC App is repeatedly checking for incoming messages until a message arrived, then parsing the message and using the Test Recipe preparing and running the test, creating the result set and sending it back to the Testing Station App.

1. Activity diagram of running a test on the Tested OBC App.

### Communication Protocol and Channels

#### The Communication Protocol

We created a simple protocol for the communication between the Testing Station App and the Tested OBC App. The protocol contains 2 messages, one from the Testing Station App to the Tested OBC App and the other one back. The application work in a Master-Slave like manner, the Tested OBC App never start the communication, only waits for a test request message and answering with the test results message after completing the test. The Testing Station App send one test at a time and wait for the reply up to a given Timeout.

**Testing Station to Tested OBC – Test Request Message**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bytes | Type | Field Name | Field Description | Expected Values |
| 0-1 | Unsigned Short | Opcode | Unique identifier for the message, used for sync and opcode, identifying the message start and type. | 0x1234 |
| 2-5 | Unsigned Int | Data Length | The size of the test data, can be different in every message. |  |
| 6-9 | Unsigned Int | CRC | 4 Bytes CRC, to identify errors in the message without relying on the communication type. | Calculated on the message |
| 10-209 | Struct | Test Recipe | Struct containing all the required options for running the test, including the polynomial degree, number of points, tested algorithm and more. | can vary |
| 210-N | Array | Points Data | The algorithms input, array with the data in each point in time, each entry containing the time at the point and 4 3d vectors, the location and velocity of 2 objects in this point in time. | can vary |

1. Test Request Message Description

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bytes | Type | Field Name | Field Description | Expected Values |
| 0-1 | Unsigned Short | Opcode | Unique identifier for the message, used for sync and opcode, identifying the message start and type. | 0x4321 |
| 2-5 | Unsigned Int | Data Length | The size of the test data, a constant size. | 188 |
| 6-9 | Unsigned Int | CRC | 4 Bytes CRC, to identify errors in the message without relying on the communication type. | Calculated on the message |
| 10-197 | Struct | Test Results | Struct containing all the test results data, the found TCA and distance, the run time, average and minimal and more. | can vary |

**Tested OBC to Testing Station – Test Results Message**

1. Test Results Message Description

#### UDP Communication Channel

The first implementation we did was UDP, the easiest to implement and use. But unfortunately, UDP have few major flaws. The first problem is a limit on the message size, in the IP layer we have a total length field in an unsigned short variable, limiting the total size of each IP packet to around 65,500 bytes of data (after subtracting the headers size) so we need to send our message in blocks ourself, and here we get to the second problem, reliability. The protocol doesn’t assure as we get the blocks in the order we sent them or that we will get them at all, meaning that we will have to track the blocks arrival order ourself, send ACK of some kind and resend it if necessary. In each message we can have a lot of data, for example for a test of time period of a week we can easily get 6000 points, each point contains the time value and 4 vectors, location and velocity of two objects, meaning we have 13 double precision variables, each of them is 8 Bytes, the data array will be making the risk of losing parts of the message much higher.

#### TCP Communication Channel

Unlike the UDP protocol, TCP is a much better option for our needs. The protocol handles the full message, sending it fragmented if necessary, collecting and making sure we can the full message in the correct order. The cost is in run time but we only care about the run time when the algorithm is running, in other times it doesn’t really matter. We used the protocol as a client and server duo, the Testing Station being the server, running on a PC and with resources to spare. Additionally, we only need to know the IP address and port of the Testing Station. The Tested OBC connect to the station as a client, and waiting for incoming test request + answering with the results.

### Feasibility Testing Environments

We created the Tested OBC App with the capabilities to work on different system, with the intention of eventually running the application on an actual satellite’s OBC, unfortunately we still didn’t manage to get access to one but we wanted to leave the application with the ability to do so. In the meantime, we tested our app on 3 different systems.

#### Windows

The easiest option is to run our app on the same computer as the Testing Station App, working with TCP over the Local Host. This option is more fitting for testing and debugging the application in real time or even running test that doesn’t require the actual system (for example exploring the relation between the root finding algorithm we use and the size of the error). Another option we implemented is running the application in a Local Simulation mode. In this option we only use the Tested OBC App and the Local Simulation create the Test Request Message and simulate receiving the message via the communication channel. This option is really convenient for test a prepare test case and testing the system logic without relying on an actual communication and synchronization with the second app. Additionally it can be use to run tests on a system in an asynchronized manner.

#### Linux on Raspberry Pi 4

Using a Raspberry Pi 4 we had to run tests on a more limited system, with actual communication and a different operating system. We used it to test the cross-platform communication between the Testing Station App and the Tested OBC App.

#### Emulator using gem5

The last option we tested is running the Tested OBC App on and emulator. Finding an emulator that can emulate the CPU frequencies we needed reliably wasn’t simple until we come across gem5 [ ]. gem5 is a community led project, providing a modular platform for creating and researching computer systems. We can use gem5 to run our application with a simulated architecture, CPU type Memory type and size and more. We created a script for running our app with similar properties to the OBC [ ] we wanted to test. The only problem with gem5 is that its design to run on linux and that it can be slower (for example, running a test that take 2 seconds can take 10 minutes to emulate).

## Research and Development process

### Algorithms Analysis and Implementation

We started our process with the algorithms analysis and implementation, we started with ANCAS and CATCH in the previous part of our project. Using our implementation of the algorithms as a feasibility proof for our project. We continue with implementing SBO-ANCAS and analyzing the complexity and run time of the algorithms.

### The Development Process

For our development process we worked in an Agile like process, we started developing our system with a simple API and classes, adding more features each iteration. We met every week for a short sprint where we divided the tasks and discussed the project. We started with developing the app separately, each app with a local simulation, a simple implementation of a fake communication channel that created the needed messages we needed, used to debug and test ourself in each step.

### Unit Tests and Debug

We created and used the unit tests for testing different logic processes in our code, debugging a specific case if needed and making sure we didn’t make any new bugs after every change. Additionally, we used the local simulation and loggers we created to test more complex, full system cases and scenarios, like running a full test or faulty inputs.

### Testing the System

What we did, development process and so on

Testing, unit tests, testing the system.

## Development tools

### Development environment

We used Visual Studio 22 for our development, supporting both the C++ development and …

### Languages

Earlier on we decided to develop our system with C++. The core of our system, the main purpose is the algorithm and we needed to implemented them efficiently and have a good and usable implementation. After that we also needed to create an app that can work well on a limited system, the Tested OBC App that needed to work well on a relatively weak computer so the decision to create the Tested OBC App with C++ was quite easy. We chose to also develop the Testing Station App with C++, because its easier to manage communication and common structures and it give as faster calculation when needed (creating the input data, calculating the real TCA with a small time-step). To create the GUI, we used …..

### Additional Tools

#### Git

We used Git for source control, creating a repository on GitHub and using additional applications with easier user interface like Sourcetree and TortoiseGit.

#### GTest and GMock

We used GTest and GMock for our unit tests, GTest and GMock are frameworks for unit tests in C++, providing the tools required for creating unit tests, tests cases and mocks easily. Its easy to use and debug and there was a lot of information and tips for it online.

#### CMake

Because we needed to have easy option for cross – platform compiling, and to leave the option to compile easily for whatever OBC needed, we used CMake for the Tested OBC App. We created a CMakeLists file that we can use to compile the application with whatever compiler or configurations we might need. We used it to compile our application for our Raspberry Pi, the gem5 emulator and WSL virtual machine.

## Problems and solutions

Cross platform communication, synchronizing stuff, database management

## Feasibility Analysis and Test Results

### SBO-ANCAS Errors Analysis

We notice that sometimes the error we get from SBO-ANCAS TCA and distance is bigger than the tolerances we defined. To test the errors, we ran two SBO-ANCAS variations and ANCAS over a data set and checked the results.

**The Data Set**

We used a catalog of 9727 object, we got the TLE of all the active satellites from Celestrak [ ] (from 29/04/24 12:05:33UTC) , we ran the algorithms on the first satellite in the list with every other satellite and got 9726 different tests. We used 32 point per minimal revolution and tested over a week with the following tolerances:

**The Algorithms**

We used 2 variations of SBO-ANCAS, the first is the regular algorithm and in the second after finding the set in each iteration we save the first and last points and created 2 additional points to get a set of evenly spaced point in the time interval.

**A Teset Case**

Here we have an example of a single run of the 3 algorithms over the same set of data:

1. Test data for the satellites 1 and 5 in the catalog - CALSPHERE 1 and CALSPHERE 4A

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test ID | Algorithm | TCA[Sec] | Distance[KM] | Real TCa[Sec] | Real Distance[KM] | TCA Error | Distance Error | No Roots | Tolerance Reached | Number of outer iterations |
| 10 | ANCAS | 127.186963 | 551415.4845 | 127.1656473 | 551415.4912 | 0.006735 | 0.021316 | 581 | - | 959 |
| 11 | SBO ANCAS | 127.1656612 | 551415.4845 | 127.1656473 | 551415.4912 | 0.006728 | 1.38E-05 | 887 | 72 | 959 |
| 12 | SBO ANCAS ES | 127.1656473 | 551415.4912 | 127.1656473 | 551415.4912 | 1.37E-05 | 1.10E-10 | 581 | 378 | 959 |

The No Roots column is the number of iterations ended when no roots were found in range for a cubic polynomial, the Tolerance Reached column is the number of iterations ended because we reached both the tolerance in distance and in time. Like you can see in the example, we have an initial number of iteration where no roots can be found at the first inner iteration – 581, the additional iteration SBO-ANCAS failed to find root before reaching the tolerance, meaning we only reached the tolerance in 72 iterations. Unlike SBO-ANCAS in the variation we got to the desired tolerance every iteration we could, except the ones when no roots where found from the start. It seems like the cubic polynomial approximation preform better when the points are evenly spaces, and by running the algorithm we can get extremely uneven distribution of points from time to time and it can lead to slightly worse performance.

**Analyzing The Data:**

We checked how many tests didn’t reach the desired tolerances and got the following data:

1. Total fails(not within the one of the tolerances) and average values.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Fails | Total Tests | Percentage | Avg Time Error on Fail | Avg Distance Error on Fail |
| SBO ANCAS ES | 2067 | 9636 | 0.214508 | 0.001970586 | 3.6774E-06 |
| SBO ANCAS | 2692 | 9712 | 0.277183 | 0.023549618 | 0.008712228 |

\*Note: because we calculated the real TCA and real distance by running SBO ANCAS and then running with a smalltime-step (smaller than the tolerance) 1 second around the TCA we ignored data with error bigger than 1 second.

Even with the SBO ANCAS variation there are still high percentage of tests ended with error bigger than the tolerance, we think it’s a problem with the cubic polynomial, unable to find roots in range when working with small or unevenly spaced values.

**Conclusion**

Even though we got better results with the SBO ANCAS variation, the number of points the algorithm use can be up to 10 times the number of points SBO ANCAS used and the run- time acts the same. So practically SBO ANCAS is still a better option, running significantly faster and getting accurate results most of the time, and still getting only a small error when not.

### Run Time Comparison

Running and analyzing the results!! important

## Results and conclusion

Conclusion. Screen shots of the app and so on.

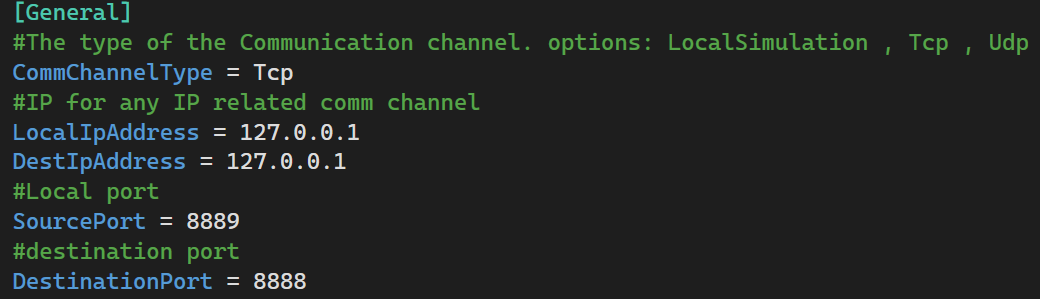
# User guide

## Testing Station App

**Configuring the System**

The application might need specific configuration in order to run properly, because the IP address we used when communicating with the Tested OBC App can be different between system and operation modes. All the application configuration can be done using TestingStationAppSettings.INI file, found where the application is located. In the configuration file you can chose the communication types, UDP (Not recommended), TCP(Recommended) and Local Simulation. The local simulation is an asynchronized mode running without the Tested OBC App while simulating the full application capabilities and can be used to test the environment. For the UDP you need to place both the Testing Station PC IP in the local Ip address field and the Tested OBC PC IP in the destination IP address filed. For TCP you only need to update the local IP and port.

1. Testing Station App configuration. The Source and Destination ports should be the opposite of the Tested OBC App ports.



**Start Window**

...

**Creating a new test**

....

**Watching the tests results**

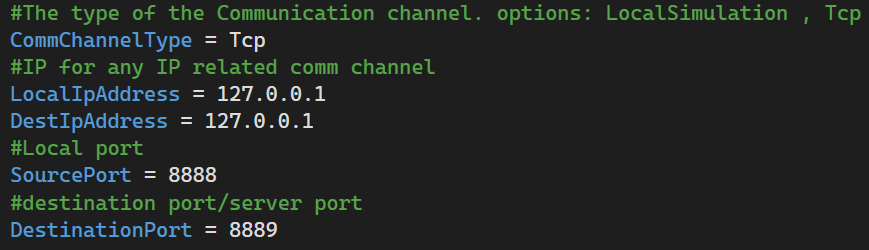
....

## Tested OBC App

To run the Tested OBC App you only need to set the wanted configuration in the INI file and start the application, the application will continually try to connect to the Testing Station, and when connected will wait for a test request message.

The configuration file contains a few important options you will need to consider.

1. Tested OBC App configuration. The Source and Destination ports should be the opposite of the Testing Station App ports.



We can decide between three operational modes, Tcp, Local Simulation and Udp(only on windows).

The Local Simulation doesn’t require the Testing Station and run a simple tests case when activated, can be used for testing the target system.

The Tcp option used for communicating with the Testing Station. For TCP we only care about the destination Ip address and port, both should be updated in the file with the Testing System Ip and port. The Udp option is only supported on windows.

After configuring the INI file it should be place in the same folder as the Tested OBC.

# Maintenance Guide

## Testing Station App

## Tested OBC App

### Installing and Running the Application

#### Running with gem5 Simulation

To run the gem5 simulation you need to clone, build and only then run the required script. The simulation has a few requirements and can only run on Linux. To save us the trouble we created scripts for every step. In the `ReleasedVersions\TestedObcApp` we have a file called `InstallGem5X86`, the script installs any requirement, clone the gem5 repository and build the gem5 simulation. The building process can take a while but only needed to be done one. After the build completed you can run the build with the existing compiled application by running the `RunGem5X86` script or rebuild the Tested OBC App by using the `BuildAppForX86`.

#### Running on Windows

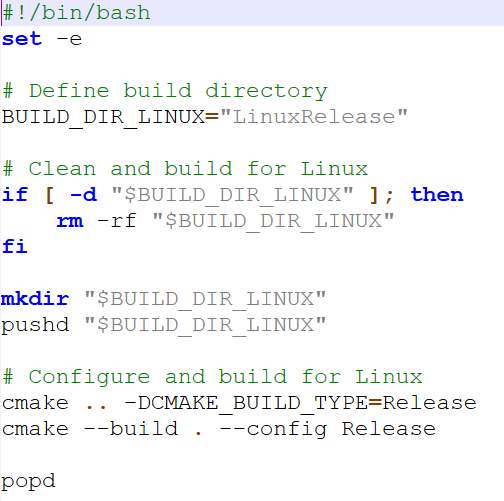
In the ReleasedVersions folder you can find executable of the latest Tested OBC App ready to use. To compile the application for windows you can either compile a released version via Visual Studio or use a script located at ` Code\TestedOBCApp\TestedOBCAppCMake` and building a new windows version.

Image ?,

#### Running on any other systems

For other system you can either use the existing ` build\_Linux` or ` build\_win` scripts or use the `CMakeLists` file to build the project from the command line.

1. The build Linux script, simply building with CMake in release.



### Error Detection and Debugging

#### Log Files

The Tested OBC App have 2 loggers, both should be created in the Logger folder. The first one is an Event Logger, logging system event like starting the system, configuration loaded, receiving messages and starting tests and errors. The second logger is a simple results logger, saving the tests results in a csv file. Both log files are created when the application starts and their names contain the creation time and date, creating a new file for every system run. The event logger can be used to identify problems with the system, like connection or communication errors.

#### Local Simulation

The local simulation is a great debugging option, with an asynchronized run and no dependencies and a single threaded application debugging is as easy as it gets. The only problem that can’t be identified in the local simulation is a specific communication problem. You can run the local simulation mode in Visual Studio by making sure that the configuration INI file (found under the main filter in VS) is set to local simulation. There are additional local simulation related parameters in there, like what algorithms to test and more.

### Implementing Changes

We collected a few possible future needed updates and how to implement them easily.

#### Additional Communication Types

Although we think that the TCP communication should suffice, you might want to run the application on a very specific satellite’s computer with limited access to communication and connection, and might need to use a different communication type (serial communication for example). We designed our application to allow adding such changes easily. All you need to do in order to add a new communication type is to implement the ICommChannel interface, and add the new channel to the applications and the creation to the Factories.

The comm managers on both applications doesn’t assume any protocol or the correctness of the input and can still work and identify errors even with a less reliable communication.

#### Additional Algorithms Variations

Adding a new algorithm can be done by adding the algorithm to the algorithms Enum (for test request and so on) and implementing the creation of the object in the Tested OBC App Factory and Test Manager. Adding a new root finding algorithm for CATCH is the same, updating the relevant Enum and factory, while implementing the root finder interface.

#### Additional Test Creation Options

Adding additional options for the user can be quite easy, for example setting a set of tests at once instead of manually, or running over a set of inputs and creating a test for each one. All we need to do is add an appropriate function to the Lab and Lab Wrapper classes and create the appropriate GUI.

#### Testing Different Algorithms Types

You can use our application as a base and add different tests, for example testing other required algorithms for satellites. To do so might be more complicated, we recommend adding a test type to the test request message header allowing you to parse the data required for your test by adding code instead of changing the full application.

Starting with installation (scripts, requirements) for each system.

Requirement for compiling after changes, used libraries, how to add stuff(catch roots finding algorithms for example)

# REFERENCES:

[1] Denenberg, Elad. "Satellite closest approach calculation through Chebyshev Proxy Polynomials." <https://www.researchgate.net/publication/338609111_Satellite_Closest_Approach_Calculation_Through_Chebyshev_Proxy_Polynomials>

[2] Alfano, S. "Determining Satellite Close Approaches-Part II."

<https://www.researchgate.net/publication/252559100_Determining_satellite_close_approaches_part_II>

[3] NASA on orbital debris: <https://www.nasa.gov/mission_pages/station/news/orbital_debris.html>

[4] SGP4 python library :

<https://pypi.org/project/sgp4/>

[5] NASA Procedural Requirements for Limiting Orbital Debris:

<https://www.iadc-home.org/references/pdfview/id/74>

[6] UN Space Debris Mitigation Guidelines:

<https://www.iadc-home.org/references/pdfview/id/75>

[7] CelesTrak:

<https://celestrak.org/>

[8] Satellite Orbital Conjunction Reports Assessing Threatening Encounters in Space:

<https://celestrak.org/SOCRATES/>

[9] The Kessler Syndrome: Implications to Future Space operations:

<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=227655e022441d1379dfdc395173ed2e776d54ee>

[10] SGP4: Revisiting Spacetrack Report :

<https://celestrak.org/publications/AIAA/2006-6753/AIAA-2006-6753.pdf>

[11]Google test framework

<https://github.com/google/googletest>

[12] EnduroSat’s Onboard Computer Type I:

<https://www.endurosat.com/cubesat-store/cubesat-obc/onboard-computer-obc/>

[13] Virtual COM Port Driver:

<https://www.virtual-serial-port.org/>

[14] Freethink article about space debris:

<https://www.freethink.com/space/space-debris-15000mph>