DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA

Final Report for Técnicas Avançadas de Programação

Instituto Superior de Engenharia do Porto

2022 / 2023

1180761 Jorge Pessoa
1181597 Rui Alves
1191421 Luís Neves

isep Instituto Superior de Engenharia do Porto

Index

Conteúdo

1 Introduction 3
1.1 Contextualization 3
1.2 Problem Description 3
1.3 Report Structure 3
2 Milestone 1 – Fist Come First Served with Emergency 4
2.1 Analysis 4
2.2 Implementation 5
2.3 Testing 6
3 Milestone 2 – Property-based Tests 7
3.1 Analysis 7
3.2 Implementation 7
4 Milestone 3 – Delay Minimization 9
4.1 Analysis 9
4.2 Implementation 9
4.3 Testing 11
5 Conclusion 12
5.1 Accomplished objectives 12
5.2 Possible improvements 12
5.3 Final appreciation 12

1 Introduction

1.1 Contextualization

This report was elaborated for the course Técnicas Avançadas de Programação from the master's degree in software engineering at ISEP. It encompasses both descriptions and solution explanations for the three iterations of the project developed in the discipline's scope. The main objective of the project is to design and develop an application using functional programming techniques.

1.2 Problem Description

The assignment of runways and ideal takeoff or landing times for a group of aircraft getting ready to leave or arrive at an airport is known as the Aircraft Scheduling Problem (ASP). While maintaining sequence-dependent separation criteria and maximizing runway utilization, the goal is to reduce delay costs.

Based on their size and weight, aircraft are classified into certain classes with predetermined target times within specified time ranges. The assignment of runways and timings must consider the aircraft classes and operation type (landing or takeoff)-dependent sequence-dependent separation requirements.

The main objective is to allocate runways and choose the best takeoff or landing times to reduce delay expenses. Penalties for delays are determined by the kind of operation, the class of aircraft, and the duration of the delay. Additionally, some runways could have limitations, like being reserved just for a certain kind of aircraft or operation.

The proposed research effort intends to create methods and algorithms for ASP optimization. The goals of the study include optimizing the scheduling and assignment of the runways, taking diagonal separation restrictions into account, and steadily raising the value of the solution. To improve the effectiveness of aircraft scheduling, the algorithms will progressively manage more complicated constraints and goals.

1.3 Report Structure

Besides the introduction and conclusion, the report is divided into three separate parts, each describing the problem and solution delivery of a milestone. Each milestone section is further divided into the analysis, implementation, and testing sections.

2 Milestone 1 – Fist Come First Served with Emergency

2.1 Analysis

In this initial milestone, the algorithm's objective is to generate a schedule that satisfies the defined constraints for aircraft scheduling without considering the total cost of delay. The focus is on applying constraints as early as possible to eliminate invalid solutions.

The algorithm for Milestone 1 operates on the principle of "first come, first served" for non-emergency aircraft. It assigns aircraft to the earliest available runway based on their arrival or departure times.

The purpose of Milestone 1 is to establish a functioning scheduling algorithm that can produce a valid schedule if one exists. It serves as a foundational step towards achieving the overall objectives of optimizing runway utilization, minimizing delay costs, and meeting separation requirements in subsequent milestones.

Next, in Figure 1, you can refer to the domain model diagram for a visual representation and better understanding of the components, constraints, and relationships involved in the aircraft scheduling problem.

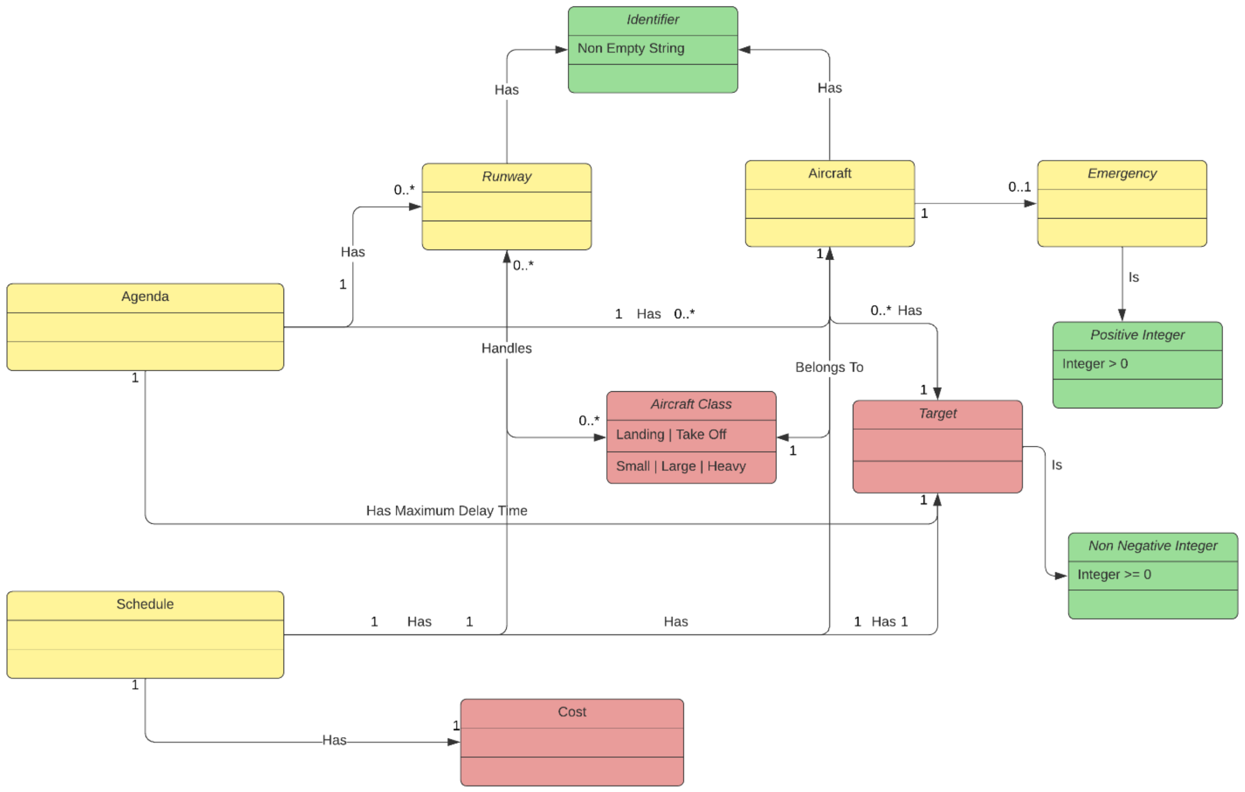


Figure 1 - Domain Model

With a representation of the domain, the algorithm could be designed. This algorithm should produce a schedule which obeys all the defined constraints. The constraints should be applied as soon as possible to prevent domain errors in the algorithm execution.

2.2 Implementation

The first milestone of the airplane scheduling problem's implemented method aims to produce a schedule that complies with the specified restrictions without taking the entire cost of delay into account.

The method uses tail recursion and adopts a functional approach to schedule the aircraft on the runways. It accepts an XML input with the schedule for arriving and departing planes.

The create method takes an XML element as input and tries to parse it into a 'Agenda' object while testing the Agenda data for domain constraint violations. The algorithm moves on to the assignRunway procedure if the parsing is successful.

The *assignRunway* method contains the core logic for scheduling the aircraft. It iterates through the list of aircrafts in the agenda and assigns them to available runways. The algorithm prioritizes the "first come, first served" principle for non-emergency aircrafts, assigning them to the earliest available runway.

The algorithm checks if there are runways available that can accommodate the aircraft's class. If no suitable runways are found, it returns an error indicating that no runways are available for that aircraft class.

When assigning runways, the algorithm also considers the previously assigned runways to avoid assigning multiple aircraft to the same runway when there are vacant ones available. It selects the runway that will cause the least delay to the aircraft based on the separation requirements and current schedule. If no suitable runway is found, it returns an error indicating that no runways are available.

For an emergency aircraft, additional checks are performed to ensure that the emergency landing can be accommodated without violating separation requirements. If a suitable runway is found, the algorithm schedules the emergency landing accordingly.

The algorithm continues the scheduling process until all aircrafts have been assigned to runways or until it encounters an error.

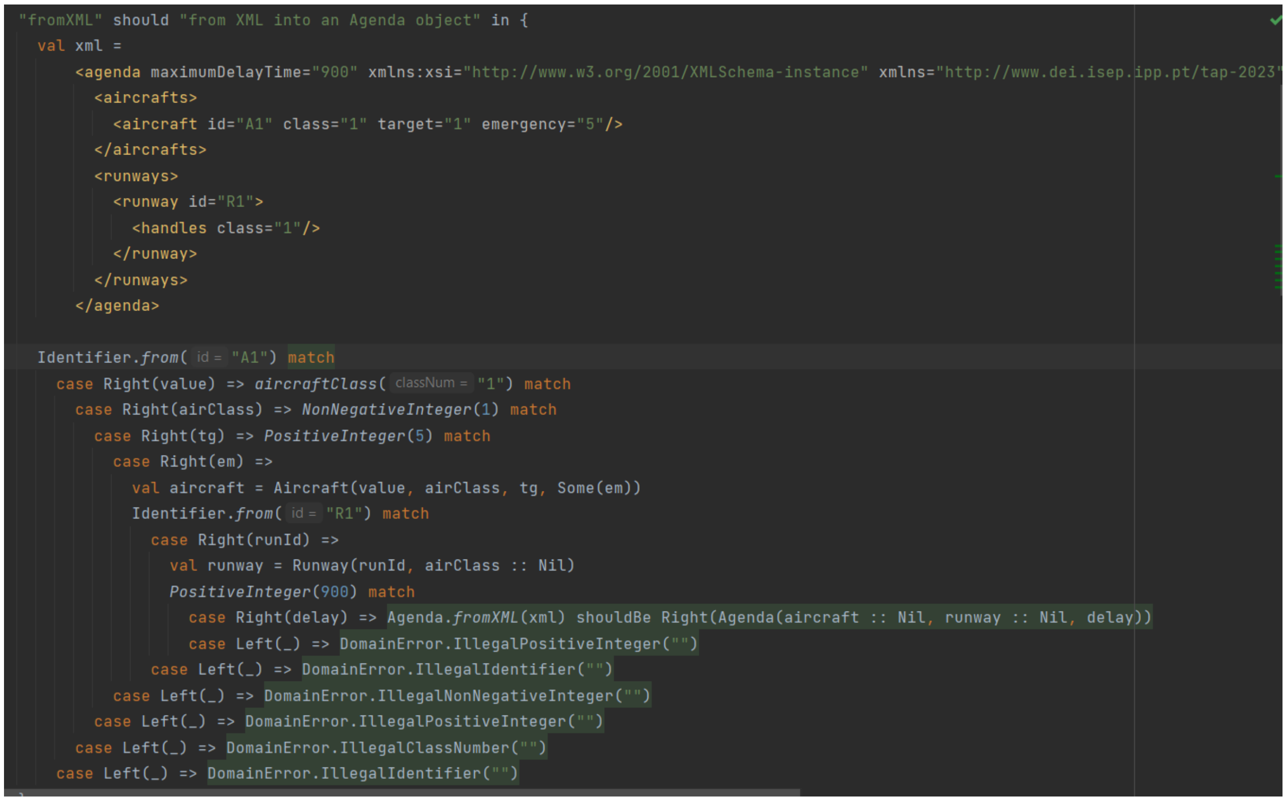
This milestone algorithm lays the foundation for creating a schedule that satisfies the initial constraints. It focuses on assigning runways to aircrafts based on their arrival or departure times, without considering the total cost of delay as a scheduling factor, even though it’s calculated. The subsequent milestones will build upon this algorithm to progressively handle more complex restrictions and improve the overall quality of the solution.

2.3 Testing

For this milestone, the elaborated tests mainly consisted of unit testing the domain objects and the utils functions. The implemented tests are the following:

- *aircraftSeparation* method:
  - Verify that the separation distance between two aircraft is calculated correctly for different combinations of landing and takeoff aircraft of different sizes (small, large, heavy).
- *aircraftClass* method:
  - Validate that the correct AircraftClass is returned for a given string representation of an aircraft class number (1 to 6).
  - Check that an empty string and an invalid class number (for example 7) result in a *DomainError.IllegalClassNumber*.
- *aircraftClassNumber* method:
  - Confirm that the correct string representation of an aircraft class number is returned for different AircraftClass instances.
- *delayPenalty* method:
  - Verify that the penalty for a given operation (landing or takeoff) and delay is calculated correctly.
  - Test scenarios with different delays and aircraft classes to ensure the correct penalty calculation.

Now, to keep the report short, the group will only present the Agenda unit tests, as seen bellow in Figure 2



```
"fromXML" should "from XML into an Agenda object" in {
  val xml =
      <agenda maximumDelayTime="900" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.dei.isep.ipp.pt/tap-2023"
        <aircrafts>
          <aircraft id="A1" class="1" target="1" emergency="5"/>
        </aircrafts>
        <runways>
          <runway id="R1">
            <handles class="1"/>
          </runway>
        </runways>
      </agenda>

  Identifier.from( id = "A1") match
    case Right(value) => aircraftClass( classNum = "1") match
      case Right(airClass) => NonNegativeInteger(1) match
        case Right(tg) => PositiveInteger(5) match
          case Right(em) =>
            val aircraft = Aircraft(value, airClass, tg, Some(em))
            Identifier.from( id = "R1") match
              case Right(runId) =>
                val runway = Runway(runId, airClass :: Nil)
                PositiveInteger(900) match
                  case Right(delay) => Agenda.fromXML(xml) shouldBe Right(Agenda(aircraft :: Nil, runway :: Nil, delay))
                  case Left(_) => DomainError.IllegalPositiveInteger("")
              case Left(_) => DomainError.IllegalIdentifier("")
          case Left(_) => DomainError.IllegalNonNegativeInteger("")
        case Left(_) => DomainError.IllegalPositiveInteger("")
      case Left(_) => DomainError.IllegalClassNumber("")
    case Left(_) => DomainError.IllegalIdentifier("")
```

Figure 2 - Agenda Unit Test

3 Milestone 2 – Property-based Tests

3.1 Analysis

Property-based testing is a powerful technique in software testing that focuses on verifying the properties or characteristics of a system using a wide range of input values. Unlike traditional example-based testing, where specific test cases are defined, property-based testing allows for the exploration of a larger input space by generating random or systematically varied inputs.

3.2 Implementation

Using the ScalaCheck package, we created a set of generators to enable our property-based testing strategy. These generators give us the ability to produce random and organized input data for testing. The generators were divided into three categories: simple, complex, and dependent.

Simple generators provide fundamental and independent values like non-negative integers or random identifiers. To build increasingly complex data structures, such as lists of aircraft classes and runways with individual identifiers, complex generators combine several basic generators. Dependent generators provide data structures that are interdependent, guaranteeing the generation of true-to-life scenarios.

These generators allow us to produce varied and random input data to accommodate a range of edge situations and eventualities. For instance, utilizing the resulting lists of planes and runways, we can build instances of the Agenda class, which represents a scheduling situation. Then, we may test characteristics like runway designators, the adequacy of scheduled runways, and adherence to deadlines.

Additionally, we have generators for testing aspects like runway separation, delay penalty computations, and emergency landings. These generators give us the ability to test a variety of situations and confirm that these system components are correct.

In conclusion, our meticulously crafted generators serve as the cornerstone for successful property-based testing. They enable systematic and automated generation of various input data, in-depth investigation of system behavior, and validation of its properties.

Property-based testing can be used to confirm the behavior and accuracy of the aircraft scheduling system in comparison to a list of predefined properties. The six given tests provide insightful information on the system's operation and adherence to the domain's criteria.

We will now demonstrate the elaborated Property Based Tests and their justification. Keep in mind that the group intends to improve their MS02 grade by redesigning some of the project as suggested by the faculty.

"In a valid schedule, a runway was assigned to each aircraft."

- This characteristic makes ensuring that each aircraft on a valid schedule is given the suitable runway assignment. By putting this attribute to the test, we can confirm that the scheduling system provides a runway to each aircraft, preventing any situations in which an aircraft is not assigned.

"Each aircraft should be scheduled for a runway which can handle it":

- This property ensures that each aircraft is scheduled for a runway that can accommodate its class. By testing this property, we can verify that the scheduling system considers the capabilities and restrictions of each runway when assigning aircraft.

"An aircraft can never be scheduled before its target time":

- This property guarantees that no aircraft is scheduled to operate before its designated "target time". By testing this property, we can ensure that the scheduling system enforces the temporal constraints of aircraft operations, preventing any violations.

"The delay penalties are correctly calculated":

- This property verifies that the delay penalties for aircraft operations are calculated accurately. By testing this property, we can ensure that the system correctly evaluates the delay for each operation and applies the appropriate penalty. This is crucial for maintaining the efficiency and fairness of the scheduling system.

"Ensure the necessary separation between runway operations on the same runway":

- This property ensures that there is sufficient separation between aircraft operations taking place on the same runway. By testing this property, we can verify that the scheduling system correctly considers the time gaps needed between operations to avoid collisions or unsafe conditions. This property is vital for maintaining the safety and smooth operation of the airport.

"In case of emergency, aircraft should land in the feasibility window":

- This property focuses on emergency situations and ensures that aircraft designated as emergencies are scheduled to land within the designated feasibility window. By testing this property, we can verify that the system properly identifies emergency aircraft and assigns them landing slots that align with the established feasibility window. This property is critical for handling emergencies efficiently and prioritizing safety during exceptional circumstances.

4 Milestone 3 – Delay Minimization

4.1 Analysis

This milestone focuses on optimizing the schedule to reduce the cost of delays and factoring emergencies into aircraft scheduling. The major goals are to keep enough space between aircraft operations on the same runway and to make sure that each aircraft is scheduled within its feasibility window, especially for emergencies.

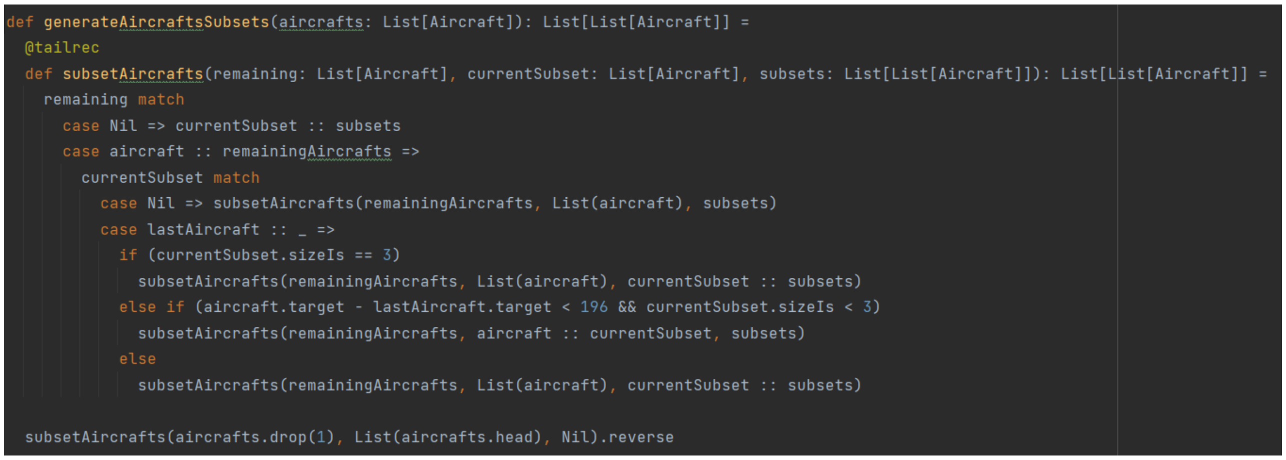
Different strategies can be used to accomplish these goals, such as studying every conceivable timetable or making good use of heuristics. While adhering to separation and feasibility restrictions, these strategies seek to minimize delay costs and increase resource usage.

The scheduling process becomes more dynamic and responsive to urgent situations by taking emergency scenarios into account, giving urgent operations priority. The goal of cost minimization is to reduce associated costs while accounting for the effects of delays on overall operations.

4.2 Implementation

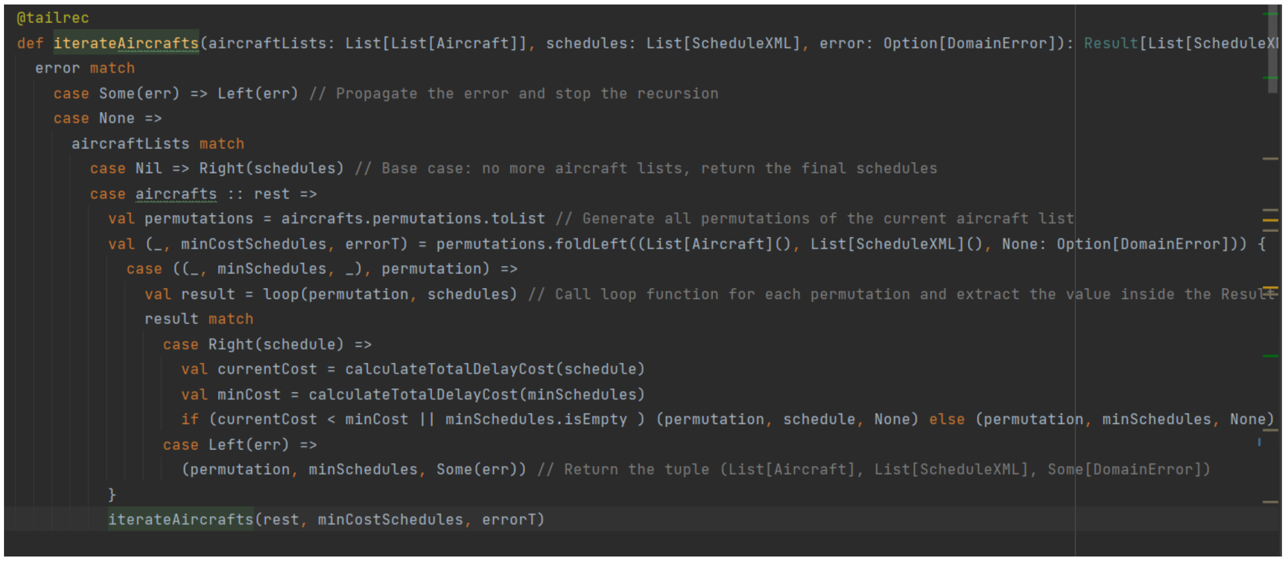
The main difference between the previous milestone and this code is the incorporation of subsets and permutations. The algorithm generates subsets of aircraft and explores different permutations within each subset to optimize the scheduling and minimize delay. This approach improves the efficiency and effectiveness of the scheduling process.

Effectively , the implementation of the subset function is shown below in Figure 3 and the other added method is visible in Figure 4. The combination of these two changes allowed the group to improve the algorithm's cost effectiveness by roughly 20%, as seen in Figure Z.



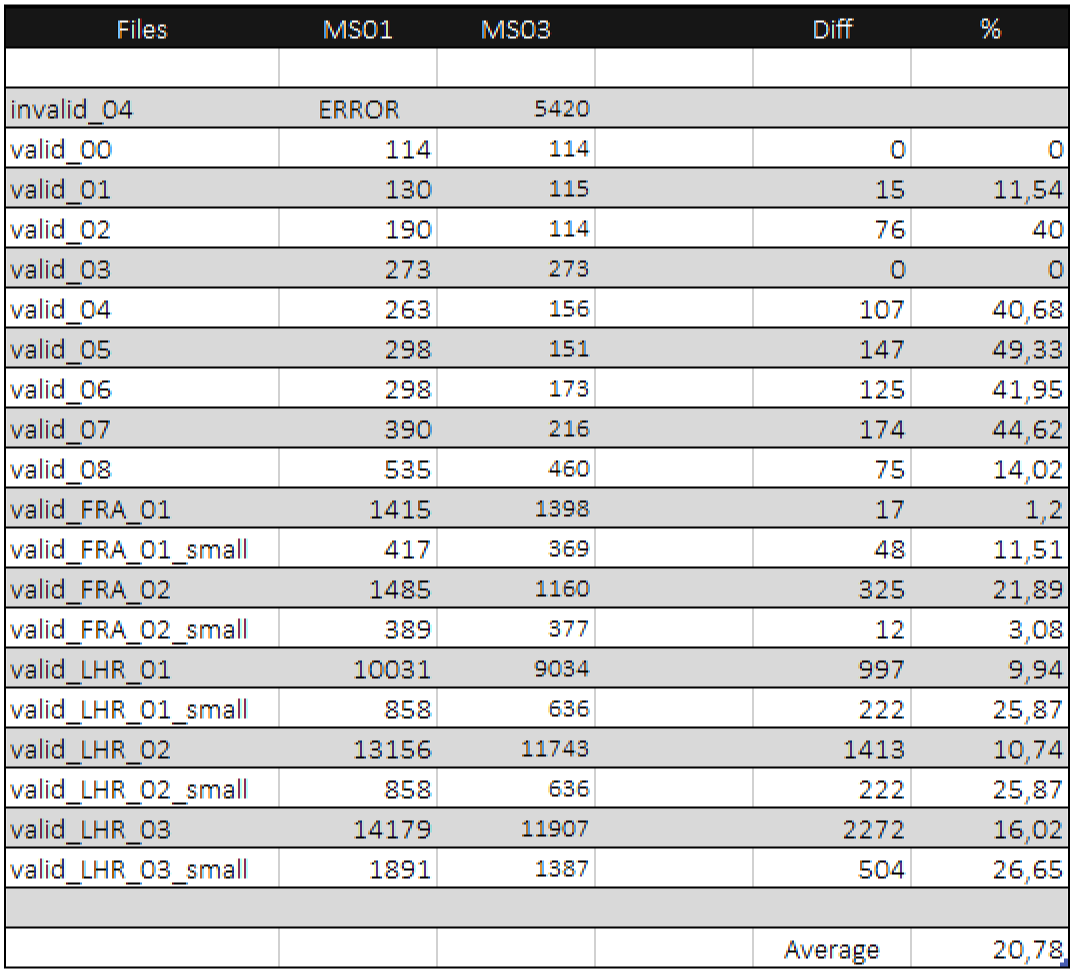
```
def generateAircraftsSubsets(aircrafts: List[Aircraft]): List[List[Aircraft]] =
  @tailrec
  def subsetAircrafts(remaining: List[Aircraft], currentSubset: List[Aircraft], subsets: List[List[Aircraft]]): List[List[Aircraft]] =
    remaining match
      case Nil => currentSubset :: subsets
      case aircraft :: remainingAircrafts =>
        currentSubset match
          case Nil => subsetAircrafts(remainingAircrafts, List(aircraft), subsets)
          case lastAircraft :: _ =>
            if (currentSubset.sizeIs == 3)
              subsetAircrafts(remainingAircrafts, List(aircraft), currentSubset :: subsets)
            else if (aircraft.target - lastAircraft.target < 196 && currentSubset.sizeIs < 3)
              subsetAircrafts(remainingAircrafts, aircraft :: currentSubset, subsets)
            else
              subsetAircrafts(remainingAircrafts, List(aircraft), currentSubset :: subsets)

  subsetAircrafts(aircrafts.drop(1), List(aircrafts.head), Nil).reverse
```

Figure 3 - GenerateAircraftSubsets Method



```
@tailrec
def iterateAircrafts(aircraftLists: List[List[Aircraft]], schedules: List[ScheduleXML], error: Option[DomainError]): Result[List[ScheduleXI
  error match
    case Some(err) => Left(err) // Propagate the error and stop the recursion
    case None =>
      aircraftLists match
        case Nil => Right(schedules) // Base case: no more aircraft lists, return the final schedules
        case aircrafts :: rest =>
          val permutations = aircrafts.permutations.toList // Generate all permutations of the current aircraft list
          val (_, minCostSchedules, errorT) = permutations.foldLeft((List[Aircraft](), List[ScheduleXML](), None: Option[DomainError])) {
            case ((_, minSchedules, _), permutation) =>
              val result = loop(permutation, schedules) // Call loop function for each permutation and extract the value inside the Result
              result match
                case Right(schedule) =>
                  val currentCost = calculateTotalDelayCost(schedule)
                  val minCost = calculateTotalDelayCost(minSchedules)
                  if (currentCost < minCost || minSchedules.isEmpty ) (permutation, schedule, None) else (permutation, minSchedules, None)
                case Left(err) =>
                  (permutation, minSchedules, Some(err)) // Return the tuple (List[Aircraft], List[ScheduleXML], Some[DomainError])
          }
          iterateAircrafts(rest, minCostSchedules, errorT)
```

Figure 5 – IterateAircrafts Method



Files	MS01	MS03		Diff	%
invalid_04	ERROR	5420			
valid_00	114	114		0	0
valid_01	130	115		15	11,54
valid_02	190	114		76	40
valid_03	273	273		0	0
valid_04	263	156		107	40,68
valid_05	298	151		147	49,33
valid_06	298	173		125	41,95
valid_07	390	216		174	44,62
valid_08	535	460		75	14,02
valid_FRA_01	1415	1398		17	1,2
valid_FRA_01_small	417	369		48	11,51
valid_FRA_02	1485	1160		325	21,89
valid_FRA_02_small	389	377		12	3,08
valid_LHR_01	10031	9034		997	9,94
valid_LHR_01_small	858	636		222	25,87
valid_LHR_02	13156	11743		1413	10,74
valid_LHR_02_small	858	636		222	25,87
valid_LHR_03	14179	11907		2272	16,02
valid_LHR_03_small	1891	1387		504	26,65
				Average	20,78

Figure 4 – Comparison Table

4.3 Testing

The performed algorithm tests designed to validate the functionality and correctness of various sub-algorithms implemented in the scheduling system. These tests verify the behavior of the algorithms in different scenarios and ensure that they produce the expected results.

As so, the performed tests were the following:

Generating Aircraft Subsets:

- This test verifies the generation of subsets of aircraft based on their classes.
- The test creates a list of aircraft with different classes and total ground times. It then asserts that the generated subsets contain the aircraft arranged in the correct order based on their classes.

Finding the Best Runway:

- This test ensures the correct selection of the best runway for a given aircraft based on its class and existing schedules.
- The test sets up a list of aircraft and available runways. It then checks that the algorithm correctly identifies the best runway for a specific aircraft, considering its class and the existing schedules on each runway.

Finding the Best Runway (No Runways Available):

- This test handles the scenario when no runways are available for scheduling aircraft.
- The test prepares a list of aircraft but assigns an empty list of runways. It verifies that the algorithm detects the unavailability of runways and returns an appropriate error.

Finding the Best Fit for Emergency:

- This test examines the algorithm for identifying the best runway and existing schedules to accommodate an emergency aircraft.
- The test constructs a list of aircraft and sets up an emergency landing scenario. It then validates that the algorithm correctly determines the best runway and schedules to accommodate the emergency aircraft.

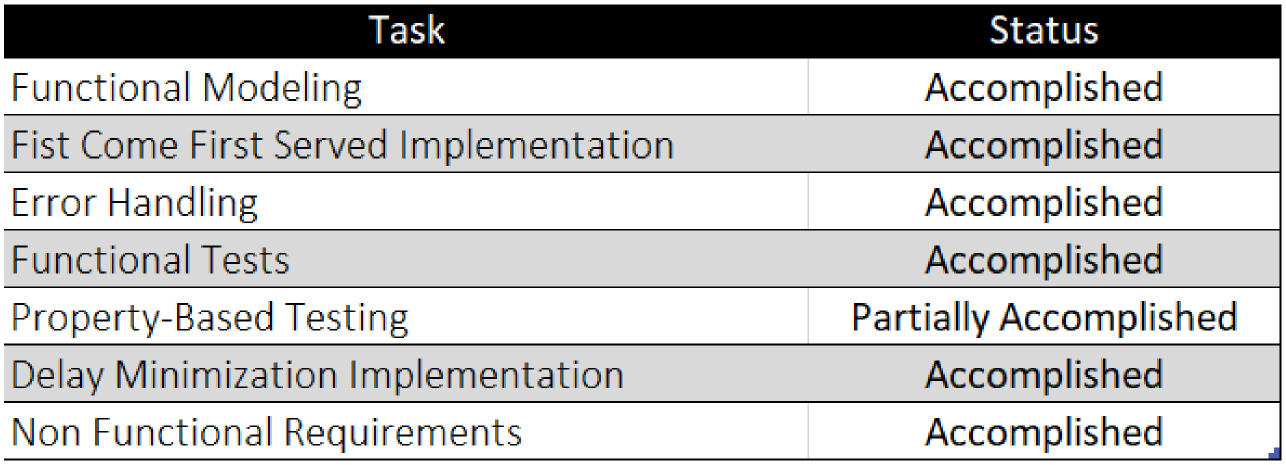
Finding the Best Fit for Emergency (No Runways Available):

- This test covers the case when no runways are available for emergency landings.
- The test creates a list of aircraft but assigns no runways for emergency landings. It ensures that the algorithm recognizes the absence of suitable runways and returns an error accordingly.

These algorithm tests support the scheduling system's ability to handle a variety of scheduling circumstances by ensuring the precision and efficacy of each sub-algorithm.

5 Conclusion

5.1 Accomplished objectives



Task	Status
Functional Modeling	Accomplished
Fist Come First Served Implementation	Accomplished
Error Handling	Accomplished
Functional Tests	Accomplished
Property-Based Testing	Partially Accomplished
Delay Minimization Implementation	Accomplished
Non Functional Requirements	Accomplished

Figure 6 - Accomplished Objectives

5.2 Possible improvements

Several optimization efforts can be undertaken to further improve the solution designed for the aircraft scheduling problem. In fact, there are opportunities for optimizing the algorithm, such as the advanced genetic algorithms or constraint programming, that can improve the overall quality of the aircraft schedule. Also, testing can identify performance bottlenecks and areas for optimization. By analyzing the system's performance under various load conditions and optimizing critical sections of the code, it is possible to improve response times and scalability.

5.3 Final appreciation

In conclusion, we have successfully developed a solution for the aircraft scheduling problem provided, spite having ground for optimization. To cope with the lack of flexibility and efficiency with Fist Come First Served method firstly designed, the development was then focused on delay minimization, applying heuristic solutions that decompose the problems into a set of smaller problems with shorter scopes, preventing brute force operations and enabling efficient allocation of resources.