Collision Avoidance

Mapping Between AADL and Python

Ryan Bomalaski & Venkatesh Sekar
SWE5002
Dr. Siddhartha Bhattacharyya

12/13/2017

Abstract

Automated collision avoidance systems are pivotal to the creation of autonomous aircraft systems.[1] In order to implement a robust yet safe collision avoidance systems, there need to be reviews and checks throughout the design process. Doing these design checks manually can be time and resource intensive, thus it is pertinent to automate as much of this process as possible. There existed previous work in automating AGREE to other languages[2], as well as previous work in collision avoidance systems to build upon[3]. This project took a three step approach to begin the framework of automating the implementation of design language requirements into actual code systems.

The first step was designing a simple collision avoidance system in Python while also mapping the requirements of a collision avoidance system in an architectural framework Architecture Analysis and Design Language (AADL) with AGREE (Assume Guarantee Reasoning Environment). Both of these work products were extensively tested, to ensure correct behavior. The second step was to manually map the features of the Python code to the AGREE design language. The manually created map allowed for the first steps toward automation to begin. The third step was to create a software in Java that could take in an AGREE language file and output a Python software framework.

Introduction

Despite the rate of mid-air aircraft collisions declining in recent years[4], mid-air aircraft collisions still make national headlines and perpetuate the perception that flying is inherently unsafe. Airborne collision avoidance systems (ACAS) are a recent development to both assist human pilots as well as to allow for autonomous systems to fly safely. As of 2009, only one ACAS implementation met the necessary ICAO standards for usage.[5] The rigorous standards and importance of a lower failure rate, mean that creating and testing collision avoidance systems is both time consuming and difficult. Automation has been shown, throughout history, to decrease cost in necessary man power and to decrease the time for creating a product.Likewise, recent studies into design language testing have shown the weakness of simulation only testing. The design language solutions allow for a more thorough testing of all possible cases. [3] In addition to this some automated mapping has already been completed to translate from a design language to a more robust coding implementation. [2]

Using previous collision avoidance solutions, this project will first create a simple collision avoidance algorithm in Python. From there, using the previous maps for AGREE to other languages, a manual mapping will be performed to link the python collision avoidance system to the AGREE design language. Finally, an automated tool will be designed to allow for direct translation from AGREE design language to a python framework. All of this will help to show that, given a design language for testing, an automated framework generator can be created for even the most safety critical systems.

Collision Avoidance Software

Process Model

A software process model is a description of the sequence of activities carried out in a software engineering project, and the relative order of these activities. The main phases of process models are

- **Communication**: Helps the developers to understand the objective.
- **Planning**: Helps the developers work towards a common goal while working on individually different functionality.
- **Modeling**: Helps the developers create a map of the data flow, process flow and eventual product to be produced.
- **Construction**: Here the developers use the work products of the previous steps to create a software.
- **Deployment**: Here the developers integrate each previous increment with the newly created increment.

In this project a hybrid Incremental-Iterative process model was used. The series of releases was referred to as increments, with each increment providing more functionality in an iterative fashion. After the first increment, a basic requirement was implemented. Then, based on feedback, a plan was developed for the next increment. Modifications were made accordingly. This process continued, with increments being delivered until the complete product was delivered.

Communication for this project was coordinated during in class and after class meetings. In order to facilitate collaborative work, a shared github repository was created as well.[6]

Planning was done to ensure that expected work loads were equal and that each domain of the project was covered by the team member most comfortable with doing that work. While each member was expected to take on work outside of their comfort zone, neither member was asked to take on work beyond their abilities.

Modeling for the collision avoidance software was done in two ways. AADL was used to

model the architectural framework of the system. This model was later used for the second and third stages of the project: direct mapping of the code product to the design language and then automated translation of the design language to a workable python framework. The second source of modeling was a series of architectural diagrams to show the overall data flow to be expected in the collision avoidance/aircraft/simulator testing set up.

Construction of the collision avoidance software in python was performed in increments once all initial requirements were gathered. Each increment of the software can be found on the github. Initially, C++ was chosen as the coding language, however Python was later chosen as the coding language due to its ability to allow for quicker prototyping and increments.

Deployment of the collision avoidance system was simulated by creating aircraft and simulator software to show that it was capable of working as expected. Since no automated aircraft were available to use, this psuedo-deployment was the closest to full deployment of the collision avoidance software possible.

Requirements

The process of gathering the software requirements from the client, analyzing these requirements and documenting them is known as requirements engineering. The goal of requirements engineering is to develop and maintain sophisticated and descriptive specification documents.

The requirements for this project were to model a collision avoidance system using altitude as the main adjustment parameter. No feasibility study was performed, as this was to be an in class project and business requirements/cost were not an issue.

Several Use Case scenarios were created to figure out how the system should interact with the pilot or automated system. These use case documents modeled three simple scenarios:

- Scenario 1: There is one airplane with no nearby neighbors.
- Scenario 2: There are two airplanes heading on a collision course.
- Scenario 3: There are two nearby airplanes, however they are not on a collision course.

These cases were documented and the expected outcome of these cases were explored with the client to arrive at an accepted over view of the system.

The final overall "elevator pitch" of the software was as follows: "If there is an imminent collision, the higher aircraft should move up and the lower aircraft should move down. In the event of a tie, the aircraft with the higher unique ID should move up and the lower should move down. If there is no collision imminent, then no action should be taken."

Design

Software design is an information driven method to create coherent and well planned representations of the software to be created.[7] Good architectural design constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together.[7] A focus on good architectural design allowed the project to proceed fluidly from a conceptual phase to realization of working software.

Through the use case documents created in requirements gathering and domain research done prior to the planning of the project, architectural design was represented through data flow models. The classes and structures that would later become components were plotted out and the data flow was mapped for each component [Fig 1.].

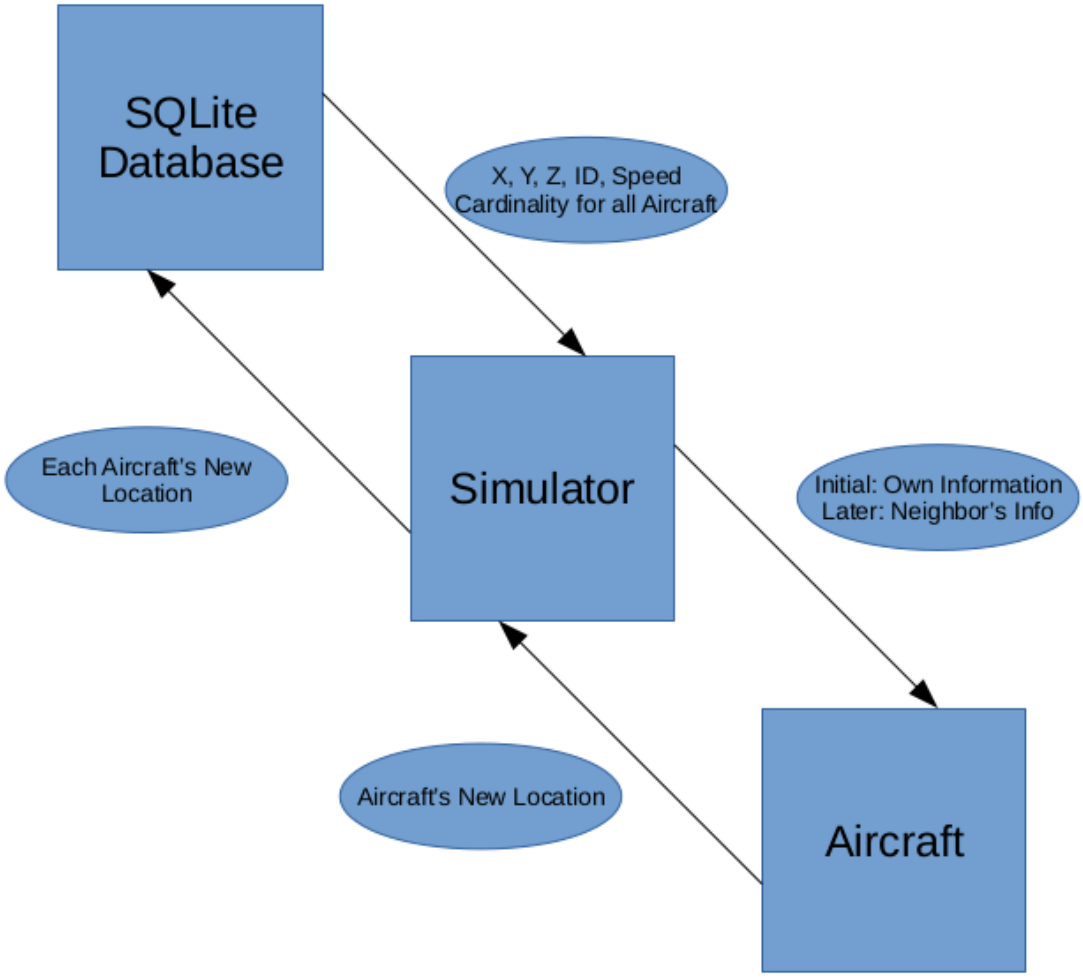

Fig. 1: *Architectural Overview showing Dataflow from Database to Aircraft object*

The dataflow diagram was taken in another level to show how the aircraft object itself handled data and what other objects it passed data to [Fig. 2].

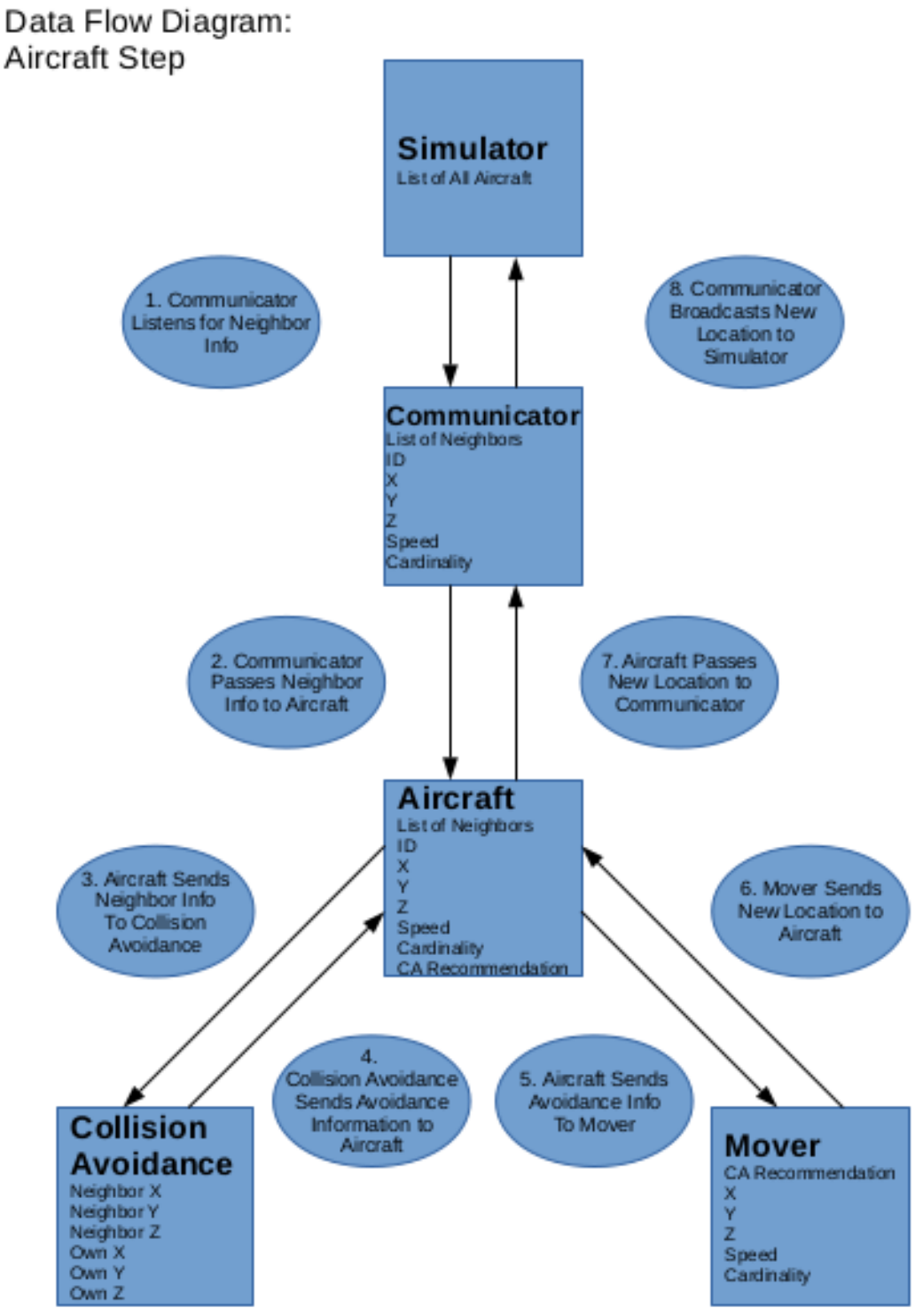

Fig. 2: *Architectural Design showing dataflow for Aircraft class and sub classes*

Components

A component is an identifiable part of a larger program or construction. It provides a particular function or group of related functions. In programming design, a system is divided into components that in turn are made up of modules. For this portion of the project, components were specified in two languages: AGREE and Python.

Python Components

For this project, a component in Python was defined as an instanced version of a class. It would not be correct to call the aircraft class a component, however an aircraft object using the aircraft class would be a component. This distinction allows for an easier conceptual separation of the framework a class provides and the specific attributes of each instance of that class. Further, a software might create multiple instances of an object while only representing the attributes of the class once in the code. For example, the Aircraft class is spelled out only once in the code, however the simulator is able to create multiple instances of that class. As a result, the complexity of the software increases with each created aircraft despite a trivial change in the actual code of the software.

The main overall class from which the components are created is **Aircraft**. The Aircraft class invokes three classes (which in turn are three subcomponents per the aircraft component): **Communicator**, **Mover**, and **Collision Avoidance**. Each component has a set of functions. The functionality of each component is as follows:

- **Aircraft:**
  - *__init__(aircraftInfoArray)*: Creates aircraft instance with initial values.
  - *listen()*: Calls communicator object to listen for nearby neighbors.
  - *broadcast()*: Calls communicator object to broadcast new location.
  - *move()*: Calls mover object to set new aircraft information.
  - *collision_avoidance()*: Calls collision avoidance object to run the collision avoidance check.
- **Communicator:**
  - *__init__(ID)*: Initializes communicator.
  - *listen(ID)*: Populates neighbor dictionary with aircraft ID as key and X,Y,Z, Speed, Card list as the value. Uses own ID to know which of the databases information to ignore.
  - *broadcast(X,Y,Z, Speed, Cardinality)*: Broadcasts the aircrafts current location.
- **Mover:**
  - *__init__(Speed, ID)* Initializes mover.

- *move(AdjustBoolean, X, Y, Z, Cardinality)*: Uses *collision_avoidance()* recommendation in addition to aircrafts current location, speed and cardinality to set a new location.
- *vert_move()*: Internal method used to calculate vertical displacement *if collision_avoidance()* has recommended evasive action.

- **Collision Avoidance:**
  - *update_ca()*: Updates the collision_avoidance object with most recent values to check against. Functionally identical to an __init__() function, however Python convention states that an init function should only be called at object creation. Thus, to not break convention, an updater function was created to allow for repeated calls.
  - *run_ca()*: Uses the neighbor dictionary to check to see if any aircraft are too close.

```
class Collision_avoidance:
    MINDIST = 1000

    def update_ca(self, myInfo, invInfo):
        self.invID = invInfo[0]
        self.invX = invInfo[1]
        self.invY = invInfo[2]
        self.invZ = invInfo[3]
        self.myID = myInfo[0]
        self.myX = myInfo[1]
        self.myY = myInfo[2]
        self.myZ = myInfo[3]


    def run_ca(self):        ## calc distance
        xSqr = (self.myX - self.invX) * (self.myX - self.invX)
        ySqr = (self.myY - self.invY) * (self.myY - self.invY)
        zSqr = (self.myZ - self.invZ) * (self.myZ - self.invZ)
        mySqr = xSqr + ySqr + zSqr
        myDistance = math.sqrt(mySqr)
        if (myDistance < self.MINDIST):
            print(str(self.myID) + ": Intruder Detected.")
            print(str(self.myID) + ": Intruder is " + str(myDistance) + "
    units away.")
        ##figure goUp
            if (self.myY < self.invY):
                print(str(self.myID) + ": Evasive Action Needed - Go Down!")
                goUp = 0
            elif (self.myY > self.invY):
                print(str(self.myID) + ": Evasive Action Needed - Go Up!")
                goUp = 1
            elif (self.myID > self.invID):
                print(str(self.myID) + ": Evasive Action Needed - Go Up!")
                goUp = 1
```

```
34              elif (self.myID < self.invID):
35                  print(str(self.myID) + ": Evasive Action Needed - Go Down!")
36                  goUp = 0
37          else:
38              goUp = -1
39          return goUp
```

Collision_avoidance: Python Code showing collision_avoidance class

AGREE Components

The AGREE is modeled using the framework AADL(Architecture Analysis and Design Language). The AADL provides specification, analysis, automated integration and code generation of real-time performance-critical distributed computer systems. The main component involved in the system is aircraft. Under aircraft there are many sub components namely collision avoidance, communicator, mover, simulator. Each sub component can contain multiple input port, output port, assumptions, guarantee and internal logic. The logic of the process is handled in the process implementation section. If the program is consistent then the no counterexample is provided. On the contrary if the system is inconsistent then the AADL provides counterexample. With the help of counterexample we can figure out the error in the system.

Testing

In order to ensure that quality software was being delivered and that the software being delivered met the expected requirements, testing was done with multiple conceivable scenarios. Further, to show that the code worked as expected, a simulator needed to be developed. As this simulator was to be an internal testing tool, it did not undergo a thorough process and was developed quite quickly.

Simulator

The main focus of the simulator was to allow for simultaneous flight of two aircraft and to facilitate the *broadcast()* and *listen()* functionality of those aircraft. It accomplished this by storing *broadcast()* data in a SQLite database, and then retrieving *listen()* data from that database. The functionality of the simulator is as follows:

Simulator:

- *__init__(numSteps, Res)*: Creates simulator instance, sets steps to run and resolution (steps per second).
- *create_airplanes()*: Creates a list of airplane objects using SQLite database.
- *run_sim()*: Runs the simulator for its specified number of steps and resolution.

```
    def run_sim(self):
        simStep = 0
        if self.steps ==0:
            while True:
                simStep +=1
                print("This is step: " +str(simStep))
                for a in self.airplanes:
                    a.listen()
                    a.collision_avoidance()
                    a.move()
                    a.broadcast()
                commit_stage()
        else:
            while simStep < self.steps:
                simStep+=1
                print("This is step: " +str(simStep))
                for a in self.airplanes:
                    a.listen()
                    a.collision_avoidance()
                    a.move()
                    a.broadcast()
                commit_stage()
```

run_sim():example of Simulator code, this code shows the stepwise order that each aircraft takes

Test Cases

Using the use case scenarios as a guide, scenario tests were created to ensure desired functionality. ST-001, or **S**cenario **T**est **001**, was designed to simulate the first use case. ST-002 and ST-003 were created to simulate the second and third use cases respectively. Scenario Test 004 was created to show that diagonal movement worked correctly. Scenario Test 005 was created to show that the collision avoidance worked for collisions that were not head-on. Each test case was written into a bash script for automated testing, with each test cases specific aircraft values hard coded into corresponding SQL files.

```
#!/bin/bash
cd ..
cd src/python
sqlite3 airwaves.db ".read db_update_5.sql"
python3 -i main.py
```

test_005.sh: Bash script to run ST-005

```
DROP TABLE airwaves;
CREATE TABLE airwaves (
airplane_id INTEGER PRIMARY KEY,
x INTEGER,
y INTEGER,
z INTEGER,
```

```
speed INTEGER,
cardinality INTEGER);
INSERT INTO "airwaves" VALUES(1,0,10000,0,200,0);
INSERT INTO "airwaves" VALUES(2,1200,10000,1200,200,270);

DROP TABLE stage;
CREATE TABLE stage (
airplane_id INTEGER PRIMARY KEY,
x INTEGER,
y INTEGER,
z INTEGER);
INSERT INTO stage VALUES (1,0,0,0);
INSERT INTO stage VALUES (2,0,0,0);
```

db_update_5.sql: SQL script to initialize ST-005

Test Results

Each scenario test was run on a pass/fail basis. If a scenario was found to fail, the software was examined to find the problem area. A fix would then be implemented to implement correct behavior and the test was run again. The final results were as follows:

	PASS/FAIL
ST-001	Pass
ST-002	Pass
ST-003	Pass
ST-004	Pass
ST-005	Pass

Table 1:Results for final executions of each scenario test

This shows that, at least as far as the scenario tests were concerned, the python collision avoidance software worked as specified by the customer.

Manual Mapping from AGREE to Python

The AGREE consists of input and output ports along with the data types representing the nature of the system. The python consists of input variables and their data types. The AGREE and python can be mapped by modifying the input and output ports as python variable and the logic written in AGREE can also be mapped to some extent in python. The AGREE provides basic representation of the logic that can be implemented in detail. In python, we can declare a string variable where as in AGREE we cant declare a string variable. Similarly in python we can construct nested if..else statement where as in AGREE we can't construct nested statements. The assumptions and guarantee provides the range of data in which the program will work. This part of the code cannot be mapped to python since in python we cant specify the range of operations. The logic of the program is written

in the process implementation section. This is also translated into python. Below table provides the mapping between AADL and python.

	Python	AGREE
	Class	Process
Datatypes:	Boolean, Int	Boolean, Integer
Methods:	update_ca(), run_ca()	Sub clauses: AGREE nodes
Decision Statements:	if, else	if, else
Declaration Statements:	combined with assignments i.e. num = 5	EQ Statements
Input Statements:	self.invY = info[2]	In port
Output Statement:	print(), db_write()	EQ Statements

Table 2:Results for manual mapping of Python to AGREE

Automated Translation Software

Overview

This will discuss the desire for automation and how to go about doing that. The automated mapping does the job of changing the AADL input output ports to python variables. This is done with the help of Java code. The Java code gets the input AADL file name from the user. This file translates the AADL input ports and output ports to python variables with the data types. The assumptions and guarantee is saved in a separate file with the same process name. The AADL logic is also translated into python. The output file is a python file saved in the folder and the assumption guarantee file will be saved in a text format. The translated code contains python variables, when the code is executed as such, it will throw few errors. By removing few AADL commands and applying few python syntax the code can be compiled and executed successfully. The AADL can translate basic decision statement to python statements.

Design and Implementation

The translator is designed to get input from the user, which will be a file name of aadl format. The program locates the file in the folder and starts the translation process. The translator translates each and every components given in aadl file to respective python format. The code is implemented such that it is open for extension but closed for modification.

Testing

There are two test cases as part of automated testing.

1. If there are no assumptions and guarantee statements in aadl file then an aadl file is passed as input to the system. The locates the file in the project folder and translates the aadl file to python file and saves it in the project folder.

2. If there are assumptions and guarantee statements in aadl file then an aadl file is passed as input to the system. The translator locates the file in the project folder and translates the aadl file to python file. The assumptions and guarantee statements in the aadl file are moved to a separate file with the name as process name and saved in the project folder.

Conclusion

This project demonstrated a successful implementation of a basic Collision Avoidance system in Python, a successful manual mapping from AGREE to Python and a successful automated mapping to translate from AGREE to a Python framework with some logic translation as well. Further work would bring up to speed more logic translation, and possibly allow for complete translation from AGREE to Python with only platform specific implementation code updates needed by the end user.

In general, the project was an overwhelming success. The over process produced three pieces of interesting software: Collision Avoidance in Python, an aircraft simulator in Python and a AADL to Python translator in Java. Further, it showed that following and adapting software engineering principles leads to a useful end point and keeps the development of software on track. Only slight changes in iteration frequency and the project time line would have lead to better results.

References

1 *Next-Generation Airborne Collision Avoidance System*, Kochenderfer M., Holland J., & Chryssanthacopoulos J. *Lincoln Laboratory Journal* Vol. 19, No. 1, 2012, pp.17-33

2 *From Design Contracts to Component Requirements Verification* Liu J., Backes J., Cofer D., & Gacek A. *Rockwell Collins Advanced Technology Center*

3 *High-Level Modeling and Analysis of the Traffic Alert and Collision Avoidance System (TCAS)* Livadas C., Lygeros J., & Lynch N. *Proceedings of the IEEE* Vol. 88, No. 7, July 2000, pp.926-948

4 Bureau of Aircraft Accidents Archives, Crashes Rate Per Year *http://www.baaa-acro.com/general-statistics/crashs-rate-per-year/*

5 *Introduction to TCAS II Version 7.1* https://www.faa.gov/documentLibrary/media/Advisory_Circular/T

6 Project Github Repository *5002_collision_avoidance* maintained by Bomalaski R. & Sekar V. *https://github.com/ryanbomo/5002_collision_avoidance*

7 *Software Engineering: A Practitioner's Approach* Pressman R. & Maxim B., Eighth Edition, McGraw Hill 2015

8 *http://www.aadl.info/aadl/currentsite/downloads/Plug-in/20Guide/202005-06-16/201030.pdf*

9 *https://www.sei.cmu.edu/architecture/research/model-based-engineering/aadl.cfm*

10 *http://www.aadl.info/aadl/currentsite/*

11 *https://github.com/osate/examples/tree/master/esweek2016-tutorial*