



**STATE UNIVERSITY OF CEARA  
SCIENCE AND TECHNOLOGY CENTER  
COMPUTER SCIENCE POSTGRADUATE PROGRAM  
ACADEMIC MASTER IN COMPUTER SCIENCE**

**LUCAS VIEIRA ALVES**

**DRES-ML: A DOMAIN-SPECIFIC LANGUAGE FOR MODELLING EXCEPTIONAL  
SCENARIOS AND SELF-ADAPTIVE BEHAVIOURS FOR DRONE-BASED  
APPLICATIONS**

**FORTALEZA – CEARÁ**

**2021**

LUCAS VIEIRA ALVES

DRES-ML: A DOMAIN-SPECIFIC LANGUAGE FOR MODELLING EXCEPTIONAL  
SCENARIOS AND SELF-ADAPTIVE BEHAVIOURS FOR DRONE-BASED  
APPLICATIONS

Dissertation presented to the Academic Master in Computer Science Course of the Computer Science Postgraduate Program of the Science and Technology Center of the State University of Ceara, as a partial requirement to obtain the title of Master in Computer Science. Concentration Area: Computer Science  
Supervisor: Prof. Paulo Henrique Mendes Maia, PhD

FORTALEZA – CEARÁ

2021

Dados Internacionais de Catalogação na Publicação  
Universidade Estadual do Ceará  
Sistema de Bibliotecas

Alves, Lucas Vieira.

DRES-ML: a Domain-specific Language for  
Modelling Exceptional Scenarios and Self-adaptive  
Behaviours for Drone-based Applications [recurso  
eletrônico] / Lucas Vieira Alves. - 2021.  
97 f. : il.

Trabalho de conclusão de curso (GRADUAÇÃO) -  
Universidade Estadual do Ceará, Centro de Ciências  
e Tecnologia, Curso de Ciência da Computação,  
Fortaleza, 2021.

Orientação: Prof. Dr. I. Título.

LUCAS VIEIRA ALVES

DRES-ML: A DOMAIN-SPECIFIC LANGUAGE FOR MODELLING EXCEPTIONAL  
SCENARIOS AND SELF-ADAPTIVE BEHAVIOURS FOR DRONE-BASED  
APPLICATIONS

Dissertation presented to the Academic  
Master in Computer Science Course of the  
Computer Science Postgraduate Program  
of the Science and Technology Center of  
the State University of Ceara, as a partial  
requirement to obtain the title of Master in  
Computer Science. Concentration Area:  
Computer Science

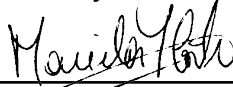
Approved in: 25/02/2021

EXAMINATION BOARD



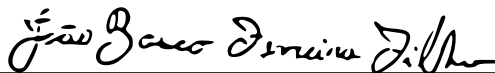
---

Prof. Paulo Henrique Mendes Maia, PhD (Supervisor)  
State University of Ceará – UECE



---

Prof. Dr. Mariela Inês Cortês  
State University of Ceará – UECE



---

Prof. Dr. João Bosco Ferreira Filho  
Federal University of Ceará – UFC

To my family, for their ability to believe in me and invest in me. Mother, your care and dedication gave you, in some moments, the hope to continue. Father, your presence meant security and certainty that I am not alone on this journey.

## **ACKNOWLEDGEMENTS**

First of all to God who allowed all this to happen, throughout my life, and not only in these years as a university student, but who at all times is the greatest teacher anyone can know.

To my parents, for their love, encouragement and unconditional support.

Thank you my brothers and nephews, who in the moments of my absence dedicated to higher education, always understood that the future is made from the constant dedication in the present!

To this university, its teaching staff, management and administration that provided the window that I now see a superior horizon, surrounded by the strong confidence in merit and ethics present here.

I thank all the teachers for providing me with not only rational knowledge, but also the manifestation of the character and affectivity of education in the professional training process, for so much that they dedicated themselves to me, not only because they taught me, but because they made me learn . the word master, will never do justice to the dedicated teachers to whom without naming they will have my eternal thanks.

“ It’s really good to go to the fight with determination, embrace life with passion, lose with class and win boldly, because the triumph belongs to those who dare ... Life is too much to be insignificant ”

Charles Chaplin

## RESUMO

Os drones estão ganhando atenção devido à possibilidade de suportar diversos tipos de aplicações, como busca e resgate, vigilância e entrega de mercadorias. Como podem operar em diferentes ambientes, é possível encontrar incertezas e situações excepcionais, não previstas inicialmente, durante o uso de aplicativos baseados em drones. Nesse domínio, estratégias auto-adaptativas têm sido usadas com sucesso para garantir resiliência e execução contínua de tais aplicativos, apesar das mudanças no ambiente. Embora algumas abordagens tenham proposto o uso de notações de cenário, como Message Sequence Charts, ou linguagens de especificação de comportamento formal, como LTS, para modelar as situações excepcionais, elas são muito genéricas ou exigem um bom conhecimento dos métodos formais do usuário, que pode dificultar sua adoção. Além disso, eles também requerem um conhecimento profundo dos detalhes técnicos para implementar os mecanismos auto-adaptativos. Para mitigar esses problemas, este trabalho propõe uma linguagem de domínio específico, denominada DRES-ML, que permite modelar situações excepcionais e comportamentos auto-adaptativos para aplicações baseadas em drones. Ele se baseia na estrutura Dado-Quando-Então, usado na técnica de desenvolvimento orientado por comportamento (BDD) e nos principais conceitos de Programação orientada a aspectos. Também fornecemos um mecanismo de transformação de modelo em texto que traduz automaticamente os cenários excepcionais modelados em uma plataforma específica para drones, a fim de verificar os comportamentos adaptativos. A abordagem é avaliada por meio de prova de conceito que verifica a aplicabilidade em diferentes cenários excepcionais.

**Palavras-chave:** Drones cenários excepcionais. Linguagem de modelagem. Sistemas auto-adaptativos. Programação orientada a aspectos



## ABSTRACT

Drones are gaining attention due to its possibility to support wide different types of applications, such search-and-rescue, surveillance and goods delivery. Since they can operate in different environments, it is possible to encounter uncertainties and exceptional situations, not initially predicted, during the use of drone-based applications. In this realm, self-adaptive strategies have been successfully used to guarantee resilience and continuous execution of such applications despite environment changes. Although some approaches have proposed the use of scenario notations, such as Message Sequence Charts, or formal behaviour specification languages, like LTS, to model exceptional situations, they are either very generic or demands a good knowledge on formal methods from the user, which may hinder their adoption. Moreover, they also require a deep understanding in technical details in order to implement the self-adaptive mechanisms. To mitigate those problems, this work proposes a domain-specific language, called DRES-ML, which allows modelling exceptional situations and self-adaptive behaviours for drone-based applications. It relies on the Given-When-Then template used in the Behaviour-driven development (BDD) technique and the main Aspect-oriented Programming concepts. We also provide a model-to-text transformation engine that automatically translates the modelled exceptional scenarios to a drone-specific platform in order to verify the adaptive behaviours. The approach is evaluated through a proof of concept that verifies its applicability in different exceptional scenarios.

**Keywords:** Drones exceptional scenarios. Modeling language. Self-adaptive systems. Aspect-oriented programming

## LIST OF FIGURES

Figure 1 – The MAPE-K reference model . . . . .	21
Figure 2 – Aspect-oriented programming concepts . . . . .	24
Figure 3 – Modelling Process overview . . . . .	31
Figure 4 – Scenarios for the drone delivery system . . . . .	33
Figure 5 – Example of an modelled exceptional scenario . . . . .	35
Figure 6 – Domain resources . . . . .	36
Figure 7 – Metamodel of the DRES-ML . . . . .	40
Figure 8 – Metamodel of Exceptional Scenario . . . . .	41
Figure 9 – Metamodel of When clause . . . . .	43
Figure 10 – Metamodel of Then clause . . . . .	44
Figure 11 – Template for exceptional scenario in the DRES-ML . . . . .	47
Figure 12 – Autocomplete menu in the DRES Modeling Environment . . . . .	48
Figure 13 – Move Aside exceptional scenario using DRES-ML . . . . .	49
Figure 14 – <i>ModelToText</i> process . . . . .	50
Figure 15 – Screenshot of the <i>Dragonfly</i> simulation tool . . . . .	53
Figure 16 – Simulation of the monitored environment. . . . .	56
Figure 17 – Scenarios of example application . . . . .	56
Figure 18 – Keep flying exceptional scenario modeling with DRES-ML . . . . .	57
Figure 19 – Switch to Manual exceptional scenario modeling with DRES-ML . . . . .	59
Figure 20 – SafeRTH exceptional scenario modeling with DRES-ML . . . . .	60
Figure 21 – Necessary adaptation movements . . . . .	61
Figure 22 – MonitorEnvironment exceptional scenario modeling with DRES-ML . . . . .	62
Figure 23 – EmergencyCamera exceptional scenario modeling with DRES-ML . . . . .	63
Figure 24 – Correlation between DRES-ML and AOP structures. . . . .	64
Figure 25 – AST and TextGen scripts for Move Aside exceptional scenario . . . . .	65
Figure 26 – Generated move aside wrapper. . . . .	66
Figure 27 – Generated Keep flying wrapper . . . . .	67
Figure 28 – Generated SafeRTH Wrapper . . . . .	68
Figure 29 – Generated SwitchToManual Wrapper . . . . .	69
Figure 30 – Generated MonitorEnvironment Wrapper . . . . .	69
Figure 31 – Generated EmergencyCamera Wrapper . . . . .	70

## **LIST OF ABBREVIATIONS AND ACRONYMS**

AOP	Aspect-Oriented Development
API	Application Programming Interface
BDD	Behavior-Driven Development
DSL	Domain-Specific Language
GPL	General Purpose Language
GPS	Global Positioning System
MDD	Model-Driven Development
SaS	Self-Adaptive System
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language

## CONTENTS

1	INTRODUCTION .....	14
2	OBJECTIVES .....	17
2.1	General Objectives .....	17
2.2	Specific Objectives .....	17
3	OVERVIEW .....	18
4	BACKGROUND .....	19
4.1	Unmanned Aerial Vehicle.....	19
4.2	Self-Adaptive System .....	20
4.3	Domain-Specific Languages.....	21
4.4	Aspect-oriented Programming .....	23
5	RELATED WORK.....	26
5.1	Self-adaptive Approaches For Drones.....	26
5.2	Domain-specific Language Approaches For Self-adaptive System	27
5.3	Modelling Language Approaches For Drones .....	29
5.4	Summary .....	30
6	THE DRES MODELLING LANGUAGE .....	31
6.1	DRES-ML Overview .....	32
6.2	Domain Analysis.....	35
6.3	Abstract Syntax .....	39
6.4	DRES-ML Modeling Environment.....	46
6.5	Model To Text - Code Generation Process .....	49
7	EVALUATION .....	52
7.1	The Dragonfly Tool .....	52
7.1.1	Interface.....	52
7.1.2	Execution of flight simulation .....	54
7.1.3	Tool extension flow .....	54
7.2	Proof Of Concept.....	55
7.2.1	Motivating Example .....	55
7.2.1.1	<b>Exceptional Scenarios Specification</b> .....	57
7.2.1.1.1	<i>KeepFlying</i> .....	57
7.2.1.1.2	<i>SwitchToManual</i> .....	58

7.2.1.1.3	<i>SafeRTH</i> .....	59
7.2.1.1.4	<i>MonitorEnvironment</i> .....	60
7.2.1.1.5	<i>EmergencyCamera</i> .....	62
7.2.1.2	Wrapper Generator.....	63
<b>8</b>	<b>CONCLUSION AND FUTURE WORKS</b> .....	<b>71</b>
<b>8.1</b>	<b>Achievements</b> .....	<b>72</b>
<b>8.2</b>	<b>Limitations</b> .....	<b>72</b>
<b>8.3</b>	<b>Future Work</b> .....	<b>72</b>
	<b>BIBLIOGRAPHY</b> .....	<b>74</b>
	<b>APPENDIX</b> .....	<b>80</b>
	<b>APPENDIX A – ABSTRACT SYNTAX RESOURCES</b> .....	<b>81</b>

## 1 INTRODUCTION

Unmanned aerial vehicles (UAVs), most popularly known as drones, have been used for military purposes for several years (CONCEPT... , 2015). In military applications, drones are considered an essential element of the battlefield, since they capture different kinds of information on a large scale in terms of time and space (e.g. surveillance and intelligence recognition) (WANG et al., 2019).

Recent technological advances and miniaturization have enable the use of drones in several civil applications, such as protection of agricultural products (PERERA et al., 2019), search and rescue (SILVAGNI et al., 2017) (RAHMES et al., 2018), delivery and monitoring of goods (ROBERGE et al., 2018), weather and natural disasters monitoring (CECIL, 2018) (ERDELJ; NATALIZIO, 2016) construction monitoring (HAM et al., 2016) and traffic surveillance (NIU et al., 2018). Currently, drones are also being used to help containing the global pandemic of the COVID-19 in different ways. For instance, Police in Spain and Chine<sup>1</sup> are using drones equipped with speakers to tell the people in the streets to only go outside when it is strictly necessary and to remain at home, while French authorities<sup>2</sup> are using drones to find out (and tax) people that are leaving home to go to unnecessary places, such as beaches.

Although drones can be directly controlled via a remote control by a pilot, more advanced drones have pre-planned behaviours that reduce the need of a human intervention, thus increasing the level of automation. An autonomous drone selects actions from a fixed palette following a static decision process based on its immediate state, environment, and goals (MOD, 2011). Take-off, landing and flying between way points are examples of drone's automatic operations (CULLEN et al., 2017).

However, there are lots of uncertainties that cannot not initially be predicted at design time that generate exceptional situations during the use of drone-based applications (MAIA et al., 2019b). To tackle that, there are two possibilities: on the one hand, there is the creation of custom-made drones for specific situations (e.g., a tailored drone has been built for delivering an organ to a hospital in the USA in 2019<sup>3</sup>). This is

<sup>1</sup> <https://www.forbes.com/sites/zakdoffman/2020/03/16/coronavirus-spy-drones-hit-europe-police-surveillance-enforces-new-covid-19-lockdowns/#238c14e47471>  
<https://globalnews.ca/news/6535353/china-coronavirus-drones-quarantine/>

<sup>2</sup> <https://www.businessinsider.com/coronavirus-drones-france-covid-19-epidemic-pandemic-outbreak-virus-containment-2020-3>

<sup>3</sup> <https://www.nytimes.com/2019/04/30/health/drone-delivers-kidney.html>

very costly, time consuming, and usually non-reusable for other situations, thus making this option unfeasible or very restrictive. On the other hand, drones could expand their degree of autonomy through self-adaptive capabilities to deal with dynamic or unknown environments, contextual changes and system failures that may appear during the flight. In this realm, self-adaptive systems have the ability to adjust its behaviours or structures at runtime without (or with minimised) human intervention through adaptations triggered by the changes in the environment or by the software requirements, capabilities or goals (OREIZY et al., 1999).

Some work proposes self-adaptation approaches for drone-based applications (GOMES et al., 2017; YU et al., 2019). However, those work focus on proposing solutions at the level of non-functional requirements and architecture. Those solutions aim at adapting the system configuration, changing the component structure and operational parameters, thus ensuring that the operation in progress is not interrupted or the system's non-functional aspects are improved, such as data-intensive computing, and distributed execution and scalability (BRABERMAN et al., 2015). Therefore, specifying both exceptional scenarios and adaptive behaviors for drone-based applications is still a challenge.

In this direction, Maia et al. (2019) have proposed a *cautious adaptation* approach that supports changes in the behaviour of *defiant components*<sup>4</sup> in order to satisfy global requirements in exceptional situations. The approach relies on using Message Sequence Charts (MSC) (HAREL; THIAGARAJAN, 2003) to specify both normal and exceptional scenarios and wrappers, implemented using Aspect-oriented Programming (AOP) (KICZALES; HILSDALE, 2001), to apply the adaptation behaviour.

Although the authors used a drone-based example in the paper, the use of a generic scenario modelling notation, such as the MSC, may limit the description of more specific scenarios for a particular domain, since some important information may not be properly represented. When modelling exceptional scenarios for drones, some peculiarities have to be taken into account, like: (i) resources (e.g. sensors, actuators and internal components); (ii) environmental conditions (e.g. wind power and direction); and (iii) policy regulations (e.g. the drone has to be in the visual line-of-sight (VLOS) of

<sup>4</sup> A defiant component is a participating components a system-of-systems that has been designed to satisfy predefined requirements, and not necessarily intended to change its behaviour in order to support global requirements of the system-of-systems

the pilots). An exception scenario that involves some of those elements may be very difficult to model using only MSCs.

Moreover, in (MAIA et al., 2019a), exception scenarios were modelled with the purpose of formally identifying defiant components in a system-of-systems (SoS) context (via scenario transformation into a labelled transition system and model checking techniques). However, exceptional scenarios for drones may arise in applications that not necessarily contain defiant components or are part of a SoS. Nonetheless, the cautious adaptation approach requires that the adaptive behaviour should be specified directly in a wrapper implemented using AOP. Thus, the user should have a good knowledge not only in the MSC modelling notation, but also in the technical solution using AOP.

To tackle those problems, this work proposes the DRES-ML (**D**rone **E**xceptional **S**cenario **M**odelling **L**anguage), a domain-specific language (DSL) to model exceptional scenarios and self-adaptive behaviours for drone-based applications that provides a high level abstraction of the main available drone resources and environment variables. This benefits the modeller (domain expert) by reducing the complexity of specifying both the exceptional scenarios and the corresponding behavioural adaptation strategies and by enabling the modelling of more possible situations. The DSL syntax is built upon concepts from both the Behavior-driven Development (BDD) (NORTH et al., 2006), which allows the specification of exceptional scenarios by using the worldwide well-known *Given-When-Then* structure, and the Aspect-oriented programming (AOP) (KICZALES; HILSDALE, 2001), which allows the represent the drone main features as *joinpoints* and adaptation strategies as Before/After/Around in advice clauses (NORTH et al., 2006). Furthermore, it was provided a model-to-text (M2T) transformation engine that take as input a DRES specification file and generates a wrapper to the Dragonfly simulator (MAIA et al., 2019b), thus allowing the user to verify whether the self-adaptation behaviour executes correctly. Similarly to (Maia et al., 2019) , DRES-ML also relies on AOP. However, while in the former the user must know how to program using an aspect-oriented language, in this work (s)he only needs to understands some AOP concepts, specially, join point and advices, for specifying the self-adaptive behavior specification.



## 2 OBJECTIVES

### 2.1 GENERAL OBJECTIVES

The main objective of this work is to develop a domain-specific language to support software engineers in modeling exceptional scenarios and self-adaptive behaviours in the domain of drones.

### 2.2 SPECIFIC OBJECTIVES

To achieve the general objective, it is necessary:

- To identify the main elements related to the drone domain;
- To design the language abstract and concrete syntax;
- To design a modeling tool for this domain-specific language;
- To implement an engine that can transform DRES-ML scripts into a drone simulator tool;
- To conduct a concept proof to validate the approach.

### **3 OVERVIEW**

The remainder of the dissertation is divided as follows: Chapter 2 presents the background that supports the work. Chapter 3 discusses the main related work. Chapter 4 details the proposed DSL and modeling environment, and transformation approach. Chapter 5 describes the evaluation process and main results. Finally, chapter 6 draws the main conclusions and future work.

## 4 BACKGROUND

This chapter presents main the concepts that are important for understanding the work, such as Unmanned Aerial Vehicle (UAV), Self-Adaptive System (SaS), Domain-Specific Language (DSL), Aspect-Oriented Development (AOP) and Behavior-Driven Development (BDD).

### 4.1 UNMANNED AERIAL VEHICLE

Unmanned aerial vehicles (UAV), a.k.a. drones, have been used in many new applications in improving people's way of life with the on-going miniaturization of sensors and processors and pervasive wireless networking. There are many drone technology uses, ranging from on-demand delivery services to monitoring of traffic and wildlife, infrastructure inspection, search and rescue, agriculture, and cinematography (GHARIBI et al., 2016).

However, due to the heterogeneity and dynamism of the environments in which a UAV acts, there has been a need for autonomous management capacity. Thus, if a fault occurs, the UAV has conditions to take corrective actions automatically. To do this, it must be able to communicate with its controller and to return payload data such as images from cameras, and its primary state information - geographic position, speed and altitude. It also transmits information as its own conditions that covers aspects such as the battery percentage, current temperature, and sensor and actuator conditions.

Thus, due to the ability to communicate with a ground-based controller, and to perform monitoring and actions in the physical environment allows framing a drone as a component that belongs to the cyber-physical system (CPS), which is a system composed by heterogeneous components that interact with a physical environment and with other computational or physical components to achieve a goal (ROMANOVSKY; ISHIKAWA, 2016).

Different mission requirements need the creation of different UAVs, then there are a lot of categories of UAVs in regards to their mission capabilities such as HTOL (horizontal take-off landing), VTOL (vertical take-off landing), hybrid model (tilt-wing), etc. The first one has the propulsion systems at the rear or at front of the fuselage and usually has flying wing. The second one often uses a vertical propulsion system at the

front of its fuselage and have cross wings. This type of drone does not need the runway to take off because it can take off and land vertically. In addition, these drones have good performance compared to HTOL, however it has limitations with cruise speed in long distance missions. At, last but not least, that drone is a combination of the capabilities of the other aforementioned types, therefore it has the ability to perform both long-distance missions and vertical take-off landing (HASSANALIAN; ABDELKEFI, 2017).

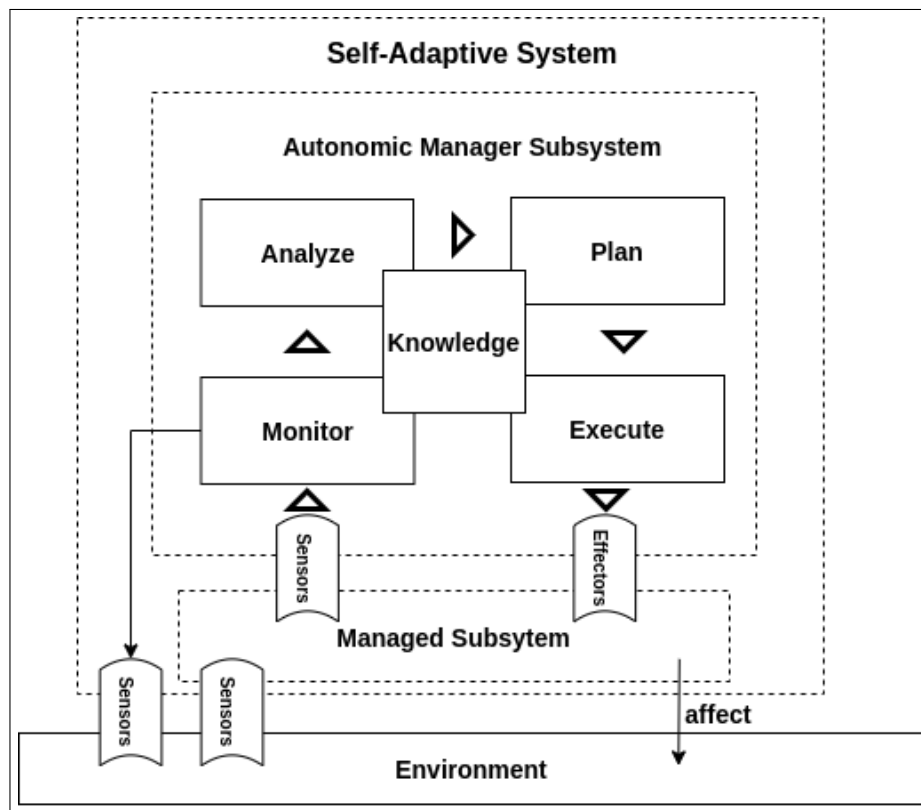
## 4.2 SELF-ADAPTIVE SYSTEM

Recently, there has been a lot of effort invested to allow systems to self-manage themselves in order to decrease total the cost for controlling a complex software systems (Ganek; Corbi, 2003). Then, a self-adaptive system (SaS) works autonomously, with minimal or no human interactions. A SaS provide some properties (SALEHIE; TAHVILDARI, 2009), such as self-configuration, self-healing, self-optimization and self-protection. Thus, that given the ability at run-time to detect, build and apply adaptations in reaction to changing conditions in order to achieve some objective (AHMAD, 2010).

These systems are often based on Feedback Control Loops (FCL) or Closed Control Loop mechanisms for self-adaptation (Ganek; Corbi, 2003). The control loop structure is composed of three phases: monitoring, decision-making, and reconfiguration. Monitoring inspects the state of the target system and its environment; Decision-making uses information from last phase to decide what actions should be taken to get a desired state; and reconfiguration executes the decided changes in the application. FCL is supported by the sensor (probes) and effector (actuators) interfaces. The first one is a software or hardware component responsible for collecting information about the drone internal state, state of the target component or environment state. Effectors are also software or hardware components that operate on the target system to enable self-adaptation.

The MAPE-K (Monitor, Analyze, Plan, Execute, over a Knowledge base) control loop (KEPHART; CHESS, 2003) is the most common FCL architecture to devise self-adaptive software systems (CALINESCU et al., 2017). Typically, the system is divided into two subsystems (Figure 1):(i) the autonomic manager subsystem (adaptation engine) and (ii) the managed subsystem (adaptable system). The former represents

Figure 1 – The MAPE-K reference model



Source – Prepared by the author

the controller and implements the management mechanisms for self-adaptation, while the latter, represents the target system that is accessed through the sensor and effector interface.

In addition, the autonomic manager subsystem consists of four components organized around a common Knowledge component representing the management flow data. The Monitor component provides resources to collect, organize and filter data through sensors. The Analyser component analyses the collected data from the previous component and is able to relate to the data stored in the knowledge component, thus allowing an autonomic manager. The Planning component provides the mechanisms to perform the required adaptations. Finally, the Executor component applies the plan built from the previous component through effectors.

#### 4.3 DOMAIN-SPECIFIC LANGUAGES

The Model-Driven Development (MDD) approach is very related to the notion of domain-specific languages (DSL)(GROHER; VOELTER, 2009). Then in MDD, devel-

opers work with high-level abstraction and can generate target code using model-to-text transformations.

A Domain-specific language (DSL) is a language with high-level abstraction for a particular domain that permits users to create models based on their domain knowledge and removing complexity from deployment. DSLs can improve productivity, understating, and maintenance of codes (BROY et al., 2012).

There are two main types of DSLs: internal (often called an embedded DSL) and external. The former can be defined as a coding of APIs inside an existing general-purpose language (GPL) that are more expressive for a specific domain, thus allowing an understanding from an expert in the domain to developers bringing the idea of a ubiquitous language (GHOSH, 2010). One well-known internal DSL is Rails, which is implemented on top of the Ruby programming language. The latter has its own syntax and semantics, thus bringing a set of techniques for lexical analysis, parsing techniques, interpretation, compilation, and code generation (GHOSH, 2010). Popular examples of external DSLs are the HTML language, designed to represent the layout of web pages, and the SQL language for querying and updating relational databases.

Besides, execution engine and target platform are two essential concepts for a DSL. The target platform is the operating system or environment in which the program needs to be executed. The execution engine can be changed and bridges the gap between the language and the target platform. It can be either an interpreter or compiler. An interpreter is a program that runs on the target platform and whose function is to load a program and act on it. A compiler transforms a program into an artifact (often a source code for a GPL) that can run directly on the target platform.

According to Voelter et al. (2013), every language, be it domain-specific or not, is composed of the following properties:

- Abstract Syntax: it is the data structure that holds the semantic information expressed by the program. It is typically a tree or a graph and does not contain any details about the notation, such as keywords, symbols, or blanks;
- Concrete Syntax: defines the notation with which users can express programs. It can be textual, graphical, tabular, or a combination of these;
- Static Semantics: it is the set of restrictions or typing rules that must conform to the programs (which must also be structurally correct); and

- **Execution Semantics:** refers to the program's behavior once it is executed. It is performed by the execution engine.

The development process can be facilitated through the use of a language development system (Language Developing System) or a toolkit. The tools can range from a consistency checker and interpreter to an integrated development environment (IDE), which consists of an editor with a syntax marker, formatter, consistency checker, analysis tools, interpreter or application compiler / generator and debuggers code if the DSL is executable (the other benefits also apply if the DSL is non-executable).

Despite a large number of systems for language development, the term *workbench language* has been used by several researchers. Language workbenches are software engineering tools that facilitate the implementation of new languages and associated tools (IDEs, debuggers, consoles, etc.) (ERDWEG et al., 2015).

This term was popularized by Martin Fowler (FOWLER, 2005). According to Fowler, a DSL that uses a language workbench should have three aspects:

- A *Schema*, which is an abstract syntax of the language;
- An *Editor*, which represents a graphical or textual representation of the abstract syntax and concede the user to manipulate the abstract syntax tree through *projections*;
- A *Generator*, which translate the abstract syntax into a low-level abstract executable representation.

Jetbrains Meta Programming System (MPS) is a well-known language workbench example. MPS is an open-source framework that provides the tools, from JetBrains, to design both domain-specific and general-purpose languages (CAMPAGNE, 2014). MPS does not need grammar or parser. Instead, editions in to the program change the abstract syntax tree directly, projected as text. Consequently, MPS supports mixed notations (textual, symbolic, tabular, and graphic) and many compositional characteristics.

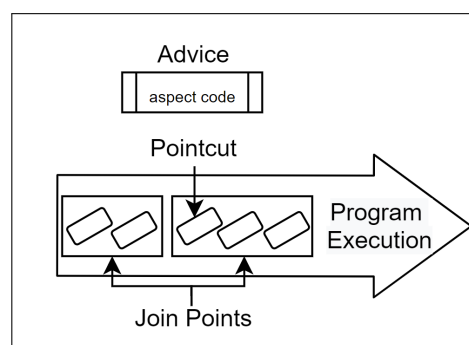
#### 4.4 ASPECT-ORIENTED PROGRAMMING

Crosscutting concerns are concerns that cut across other concerns, thus generating a system that is difficult to maintain and evolve (MOREIRA, 2005). Thereby the Aspect-oriented programming (AOP) aims to encapsulate crosscutting concerns in

separate modules, called *Aspects*. There are many languages to implement aspect modules: AspectJ (KICZALES et al., 2001b), AspectC (KICZALES; COADY, 2001), AspectC++ (SPINCZYK et al., 2002), among others. Those languages provide mechanisms for weaving aspects and base code as cohesive structure.

Aspect codes can interact with base code through the *join points*. These points could be a method called, an exception thrown, or even an attribute changed. *Pointcut* defines at what joinpoints the aspects should be associated. Often a pointcut is specified using regular expressions that represent a method signature. In addition, it is possible to use a conditional to define a conditional pointcut expression that will be evaluated at runtime for each candidate join point. An *Advice* is the implementation of an aspect that includes API invocations to the base system representing the set of action to execute at a joinpoint specified by a pointcut (KICZALES et al., 2001a). In addition, there are different types of advice: "around," "before" and "after" advice. The first one surrounds a join point, such as a method invocation. That is the most powerful kind of advice. The around advice can perform custom behavior before and after the method invocation. The second one executes before a join point but cannot prevent execution flow from proceeding to the join point. The last one is the advice to be executed after a join point completes normally.

Figure 2 – Aspect-oriented programming concepts



Source – Prepared by the author

Figure 2 visually represents the interaction of aspect-oriented programming concepts. As can be seen, there are candidate join points (larger rectangles) in the program execution of the application where an aspect can be plugged in. Smaller rectangles represent pointcut that optionally expose some of the values in the execution context of that join point. And finally, associated with the pointcut, there is the advice



implemented with the aspect code.

## 5 RELATED WORK

This section discusses the main related work in the areas associate with the contributions of this dissertation. Initially, studies that propose self-adaptive approaches in the domain of drones are presented. Then, proposals of domain-specific languages for self-adaptive systems are shown and, subsequently, DSLs for drones.

### 5.1 SELF-ADAPTIVE APPROACHES FOR DRONES

Maia et al. (2019) proposed the cautious adaptation approach for defiant components in a system-of-systems (SoS) (MAIER, 1998) environment. A component is defiant if it cannot be adapted to meet both individual (its own) and global requirements. The approach specifies normal and exceptional conditions through scenarios modeled as Message Sequence Charts (MSC) and implements the adaptation behavior in exceptional situations using wrappers based on the aspect-oriented paradigm. That work is motivated by exceptional situations that occur in a drone-based application, more specifically on organ delivery between hospitals. However, as the wrappers are implemented with join points, it is necessary that the developers have deep knowledge on an AOP language and the signature of some internal application methods.

Yu et al. (2019) proposed LiveBox, a self-adaptive distributed architecture to enable drones forensic-readiness and regulation compliance requirements to support the investigation of an eventual incident in a drone application. In order to limit the transfer rate, it provides self-adaptation activities through the MAPE-K feedback loop for dynamically reducing flight data accuracy without sacrificing run time verification accuracy. Although the LiveBox is directly related to the drone domain and uses self-adaptation techniques, it is specialized in forensic-readiness requirements. On the other hand, this work has another purpose of adapting the behavior of drones at runtime.

Due to limitations imposed by the environment, battery capacity, and movement space, it is not always possible to find an optimal path that satisfies all objectives and deal with privacy restrictions. To fill that gap, Luo et al. (2020) proposed a self-adaptive online path planner and reconfiguration of privacy-sensitive sensors (example, camera) to satisfy motion security, task completion, and privacy requirements. The privacy conditions and the necessary adaptations can be compared to the exceptional

scenarios of this dissertation's approach.

Gomes et al. (2017) proposed an unmanned aerial systems of systems architecture that performs self-control to respond to situations that impact the flight capabilities. That architecture includes components that perform maintenance and diagnostics tasks that coordinate and monitor the drone activities, thus making it possible to achieve an interconnected system that can address several issues and solutions related to flight autonomously. The work of Gomes et al. (2017) brings great architectural contributions that assisted in the definition of the DSL proposed in this work. However, this approach goes beyond the architectural level and proposes mechanisms that make self-adaptation of behavior itself.

Zhang et al. (2020) used self-organized swarm drones to monitor ships to detect pirate attacks. The drones are represented by agents that use pheromone path marks to plan optimizing monitoring coverage. In order to solve the path optimization, a heuristic depth-first branch and bound search (H-DFBnBS) algorithm is intended. This approach focuses on adaptations in various drone devices such as sensors and actuators, not only makes adaptations related to drone movement.

## 5.2 DOMAIN-SPECIFIC LANGUAGE APPROACHES FOR SELF-ADAPTIVE SYSTEM

Baresi et al. (2008) outlined an architecture, configured by special-purpose languages and aspect-oriented techniques, for the formation of component-based of decentralized self-adaptive systems. The architecture has supervised (sensors and actuators that execute the business logic) and supervisor (that monitors and decides for adaptation strategies) components. Each component implements a control loop that comprises event *collection* (collects and builds a finite sequences of events), *analysis* (checks whether these sequences satisfy a set of properties), and *reaction* (if there are property violations, this step indices new behaviors in components). The control loop is performed by Aspect-oriented techniques. Supervisors exploit the specific language to recognize patterns of events received from the elements in their clusters and trigger adaptation in the supervised elements. The language is a mix of XML technologies, such as XPath for recovery data, along with typical boolean, relational, and arithmetic operators. Similarly to the work of Baresi et al. (2008), this approach also uses the

concept of aspect to make local and global adaptations. However, the DRES-ML focus on the adaptation of drones.

Jahan et al. (2018) proposed adaptations to local objectives for the multi-agents to achieve the general objectives of the systems. In addition, to model the specifications of local and global missions, it uses the Partial-Order, Causal-Link (POCL), a formal representation that uses temporal and spatial constraints. It defined flaws as adaptation triggers so that each agent can self-integrate into another agents' plan to achieve the global mission goal. In the proposal of this dissertation, the defined language triggers and specifies the necessary adaptation.

Vogel et al. (2012) presented a modeling language for runtime megamodels that describes adaptation logic and an interpreter that executes them. In addition, those megamodels can specify multiple feedback loops specialized in different concerns. In this approach, a runtime model was not used to represent adaptive behavior.

Shetty et al. (2004) presented a domain-specific graphical language to define adaptive behaviors in fault scenarios for a large-scale system whose users are physicists. In addition, that approach has the ability to synthesize low-level programming implementations from the defined language. On the other hand, the DRES language specifies exceptional situations through a textual language.

Arcaini et al. (2019) described a pattern-oriented framework that uses the MSL (MAPE Specification Language) for designing self-adaptive systems. MSL models can be automatically translated in OpenHAB control rules that provide an implementation of architectural solutions in the context of home automation. The DSL proposed in this dissertation handles behavioral adaptations in exceptional scenarios. Furthermore, it also has mechanisms to translate that language into other output languages.

Chhetri et al. (2018) presented a Java-embedded DSL (ADSL) that enables the specification of components, and their relationship and constraints in the context of cyber resilience. The specifications achieve a distributed self-management in accordance with a model@runtime. The authors use a language to specify architectural components.

Alvares et al. (2015) defined a DSL named Ctrl-F to described architectural adaptations and constraints in the context of software components. In addition, it provides a translation to a reactive language (Finite State Automata - FSA) allowing

program behavior verification, thus ensuring that policies are not violated.

Kounev et al. (2018) proposed an approach that employs Descartes Modeling Language (DML) for modeling the self-adaptive performance and resource management of service infrastructure in heterogeneous environments. That language captures the properties of the system that are relevant for performance analysis and can be used to guide adaptations in the system. While DML focuses on specifying architectural adaptations for Quality-of-Service (QoS), the DRES-ML addresses behavior adaptation of drones.

Křikava (2013) provides a model to specify feedback control loop-based architectures through an internal domain-specific language in Scala (ODERSKY et al., 2004) programming language for EMF (Eclipse Modeling Framework)(STEINBERG et al., 2008). In addition, it is presented a set of mechanisms for model-transformation and verification and an associated tool support for modeling.

The studies cited above (CHHETRI et al., 2018; ALVARES et al., 2015; KOUNEV et al., 2018; KŘIKAV, 2013) deal with domain specific languages for self-adaptive system. However, they do not bring a specific language focused on adaptive behaviors for exceptional situations.

### 5.3 MODELLING LANGUAGE APPROACHES FOR DRONES

Hoppe et al. (2019) proposed a framework, named DronOs, that enables users to prototype personalized routines and behaviors of drones through three interaction types: Unity Scripting, Vive Scripting, and Vive Realtime. The first modality utilizes the Unity user interface for the detailed definition of routines through a drag-and-drop mechanism. The second one is a programming-by-demonstration approach that, with the use of a controller that takes the position of the user's hand, configures the drone's flight path. Lastly, the user controls the drone directly by pointing with a hand controller to the final target. Those modalities facilitate the use of drones by both experts and non-experts and the programming interfaces are enabled only for making the planning and modification of flight routines. Moreover, the proposed approach requires the use of specific hardware components to do track and enable the drone to carry out commands. Differently, this approach requires no hard components.

Da Silva et al. (2018) conducted a Systematic Literature Review investigating

how the Unified Modeling Language (UML) has been used to create Domain-Specific Modeling Languages (DSMLs) that provide support for SaSs modeling. As a result, 15 primary studies published between 2005 and 2017 were selected and reveals that the UML has been tailored through the profile-based mechanism to provide adequate support to analysis and design of the context-awareness and self-adaptiveness properties.

Bozhinoski et al. (2015) presented FLYAQ, a tool that enables end-users with no technical skills to define graphical description of a mission for a drone team through a DSL named Monitoring Mission Language (MML), which can be translated to an intermediate language named QBL. That language generates a detailed flight plan, which includes preventing collisions, respecting no-fly zones and unexpected behaviors. DRES-ML provides more mechanisms to control and monitors a drone and its environment, thus enabling more possible behaviors.

Costiou et al. (2016) designed a context-oriented of drone language, called Lub, that addresses the problem of behavioral adaptation temporarily at runtime when unexpected events come from their environment. In addition, it includes a parser and a compiler to generate code for classes and methods defined with Lub. However, not all features described have been implemented yet. In contrast, this work provides both a DSL and an editor tool that allow user to model exceptional scenarios and their respective adaptation behaviors.

## 5.4 SUMMARY

In this chapter, the works related to this research were presented. To identify such works, searches were performed in the main research sources for approaches that Self-adaptive approaches for drones, Domain-specific language approaches for Self-adaptive system, and Modeling language approaches for drones. At the end of each related work, comparisons are presented with the approach proposed in this dissertation. Next chapter gives an overview of the proposed approach.

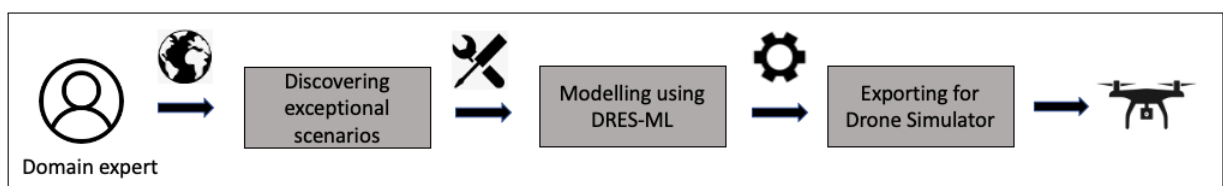
## 6 THE DRES MODELLING LANGUAGE

This chapter presents the proposed domain-specific language for specifying exceptional scenarios and self-adaptive behaviours for drone-based applications. The main user of the DSL is the expert in the drone domain, i.e., the person who understands how a drone works, including its main features (take off, manoeuvre, landing, among others), sensors (e.g, GPS, gyroscope), and resources (e.g. camera), as well as the main environment conditions that may impact the drone's fly (wind, rain, ...), and how the drone should behave in those scenarios, in a given application.

Figure 3 provides an overview of the modelling process using DRES-ML. Initially, the domain expert identifies the possible exceptional scenarios regarding the drone-based application and the adaptive behaviour strategies for each one. This step is out of the context of this work and may be carried out using a specific methodology (like the one shown in the cautious adaptation approach (MAIA et al., 2019a)) or in an informal way.

After that, the domain expert will specify the exceptional scenarios and the self-adaptive behaviours using the high-level abstractions provided by the DRES-ML, which is described in details in Section 4.1. Finally, the user can export the modelled scenarios for the Dragonfly drone simulator (MAIA et al., 2019b), where (s)he can validate the self-adaptive behaviour. The Dragonfly has been chosen since it allows the implementation of the self-adaptive behaviour of drones through the use of aspect-based wrappers and provides the necessary drone resources and environment variables used in the scenarios. However, it is possible to implement transformation engines for other specific drone platforms if they provide the scenario elements used in the DSL and make available artefacts that support the adaptive behaviours.

Figure 3 – Modelling Process overview



Source – Prepared by the author

## 6.1 DRES-ML OVERVIEW

The DRES-ML provides high level abstractions to specify both exceptional scenarios and self-adaptive behaviours for drone-based applications, thus reducing the effort from the domain expert on modelling such situations. To do that, the DSL uses BDD and AOP concepts to represent scenarios and the introduced behaviours, respectively, using a notation closer to the user natural language.

To describe the exceptional scenarios in this DSL, the *Given-When-Then* structure was used. The first clause (*Given*) represents the state of the drone context, i.e., a set of information that the drone can sense or request from either its internal resources/components (e.g battery, GPS, or camera) or external sources (e.g. weather condition from a web service).

The *When* clause represents the occurrence of a specific drone event or action, such as take off, landing, manoeuvre, among others, that should execute in a normal scenario if the expressions in the *Given* clause are satisfied. Finally, the *Then* clause is used for describing the self-adaptive behaviour that will be applied when the exceptional scenario occurs.

To specify the self-adaptive behaviour, the DRES-ML uses the concepts of advices from AOP: *before*, *after*, and *around*. This means that the new behaviour will be executed before, after or overlap the current drone expected behaviour, respectively, when the drone action that is in the *When* clause is triggered.

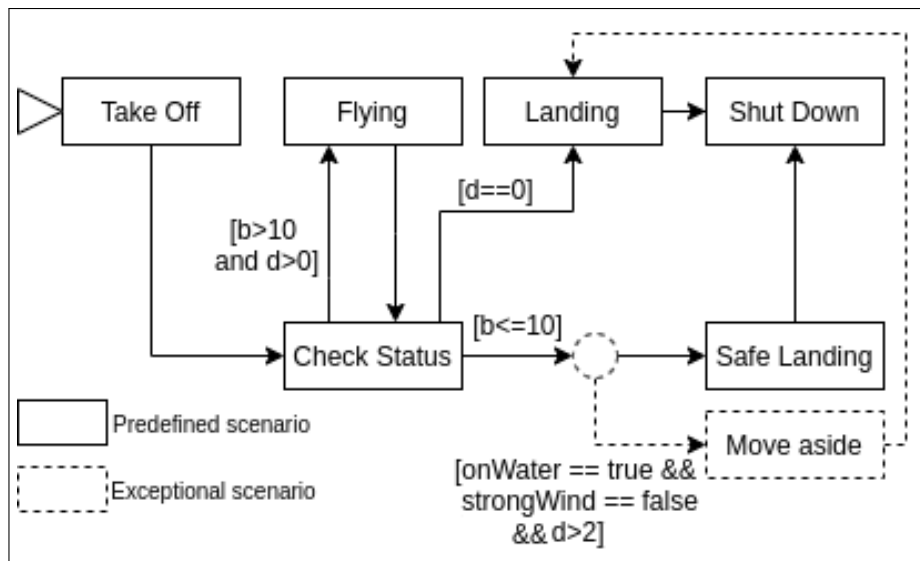
The structure of the DRES-ML supports self-adaption by following the MAPE (Monitor, Analysis, Planning and Execution) (KEPHART; CHESS, 2003) control loop. The *Given* and *When* clauses provide both the elements that need to be monitored, such as the drone internal state, sensor and actuator status, among others (monitoring phase) and the expressions that are analyzed (analysis phase) and have to be satisfied to trigger an adaptation, such as checking environment conditions and detecting malfunction of sensors or actuators.

The planning phase corresponds to the new behaviours specified by domain specialist in the *Then* clause (set of commands that the drone should perform). Finally, the execution phase regards how the adaptive behaviour is implemented, based on recommendations from the previous phase, either in a simulator or a specific drone platform, using aspect weaving, dependency injection, or other code strategy.



To give an example on how modelling exceptional scenarios using the DRES-ML, it is used an excerpt of the same scenario shown in (MAIA et al., 2019a), in which the drone was used to deliver an organ from one hospital to another (as in the Maryland case). Figure 4 shows the scenarios specification of the drone delivery system using the Message Sequence Chart notation.

Figure 4 – Scenarios for the drone delivery system



Source – Prepared by the author

Full boxes represent the drone's standard specification. It was assumed that the drone can monitor its battery ( $b$ ), the target distance ( $d$ ), wind conditions and verify the type of geographical region it is flying over (water or land), either using internal sensors or accessing external information provided by web services, for instance. It is also assumed that the drone has a predefined function that calls a safe landing procedure when its battery reaches a critical limit, in this case 10%.

However, in a critical mission, like delivering a valuable payload (for instance, blood bags or an organ), the drone has to satisfy two main global requirements:

- R1: the drone must take the payload organ from the sender hospital to the receiver hospital.
- R2: in the case when the payload cannot be delivered, the drone should not lose it.

Given that, and following the process depicted in Figure 3, it was identified that an exceptional scenario may happen when the drone's battery reaches 10%, i.e., it

should perform a safe landing, but it is flying over water (e.g, a river or the sea) and it would not manage to arrive at the destination due to the distance from the target hospital and the weak wind conditions. This would make the drone to break R2 since, by landing on water, the payload would be lost. In Figure 4, the exceptional scenario, called *Move Aside* and represented by a dashed box, should take place when the transition guard conditions are true instead of the normal behaviour Safe Landing. After executing the exceptional scenario, i.e., when the drone is manoeuvred to fly over a land region, the expected behaviour may happen.

This description of the exceptional scenario corresponds to the first step of the phase 1 of the proposed modelling approach (exceptional scenario identification). Now the exceptional scenario using the DRES-ML (step 2) can be specified.

Figure 13 depicts the exceptional scenario modelled using DRES-ML. Firstly, it is necessary to define a name for the scenario, which is *Move aside*. After, it is defined the state of the drone context for the exceptional scenario to be valid in the clause *Given*. More specifically, the exceptional scenario happens when the drone: (i) is flying over a water region (the drone's distance from a water region); (ii) its battery reaches the critical limit of 10%; (iii) the wind is not strong; (iv) and the drone is at least 2000 m away from the target.

Note that, at this moment, the domain expert is interested in specifying the conditions that apply for the exceptional scenario regardless how the information will be acquired (e.g, how to know that the drone is above water). This will be translated to a drone platform or simulator by a technology expert in the M2T phase (Section 4.3).

As previously mentioned, due to the local requirement of the drone, the safe land command will be invoked when the battery level drops to 10%, which is one of the conditions of the *Given* clause. This triggers the *When* clause of the scenario specification of Figure 13.

Finally, following the specification of the *Then* clause, the self-adaptive behaviour must occur before the execution of the safe landing command, since the advice *Before* has been used. In this case, the drone verifies whether it is still over water and, if so, it moves aside. This is performed until it gets over the land. After the adaptive behaviour has been finished, then the safe landing command is executed.

Figure 5 – Example of an modelled exceptional scenario

```

1. Exceptional Scenario MoveAside
2. Given: ( ( DRONE.distance from WATER region == 0 m ) and
3.         ( ( Wind.speed < 5 m/s ) and ( UAV.distance from DESTINATION region ≥ 2000 m ) ) )
4. When: SafeLanding STARTS
5. Then: execute Before goLandRegion
6.   goLandRegion :
7.     while ( UAV.distance from LAND region ≠ 0 m )
8.       UAV.direction to LAND region

```

Source – Prepared by the author

## 6.2 DOMAIN ANALYSIS

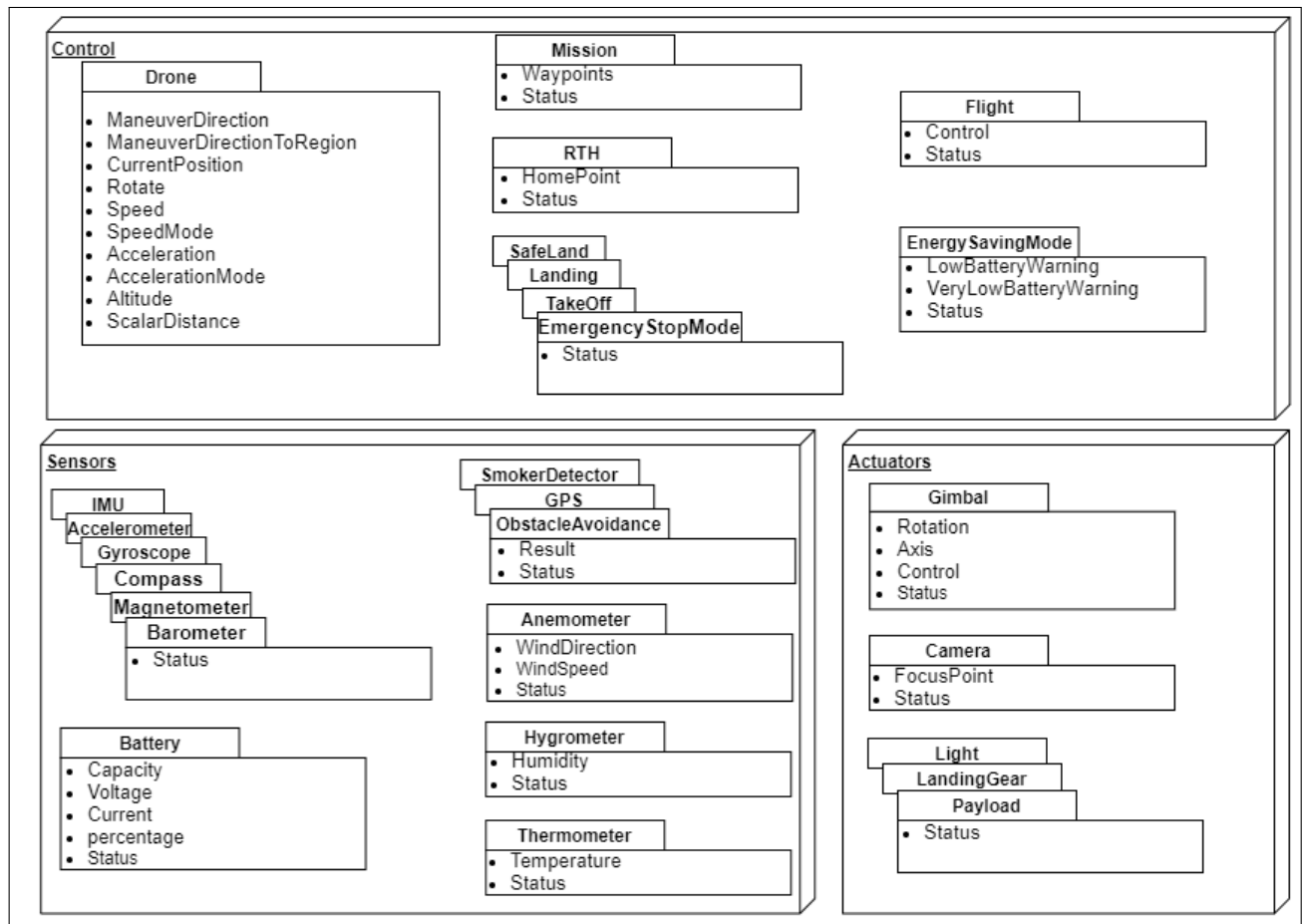
To increase the expressiveness of the language, it is necessary to encapsulate the knowledge about the target domain. Therefore, the domain analysis phase should be part of the process of developing a domain-specific language (VOELTER et al., 2013). To identify and understand the concepts and properties of the domain addressed in this work, it was gathered and analyzed the main productions found in drone context, such as simulation tools, specific application programming interfaces (APIs), and functioning of hardware components. Thus relevant knowledge was extracted and defined in abstract concepts that base the specific language. These concepts are called *resources* and they are presented in Figure 6.

The *resources* are a group of abstractions of domain elements that together assume the role of telemetry, flight control, sensor, and actuator. It was composed of three principal components namely the *Control*, *Sensors* and *Actuators*.

The *Control* component combines high-level resources from the remote and built-in smart control and internal state information to realize navigation operations. The Control component was divided in sub-components responsible for managing specific activities: Drone, Mission, RTH, SafeLand, Landing, TakeOff, EmergencyStopMode, Fight and EmergencySanvingMode. The *Drone* sub-component deals with operations directly associated with the drone movement, which may be in terms of direction maneuver, current position, distance from a position, rotation, speed, acceleration and altitude.

- Maneuver is specified by cardinal directions (such as, north, east, south and west) or direction to a specific region (for example, origin and destination);
- Current position indicates the current geographical position of the drone collected by the GPS (latitude and longitude);

Figure 6 – Domain resources



Source – Prepared by the author

- Distance between the current position of the drone and a given position (drone to regions, obstacles, pre-established points or GPS coordinates);
- Rotation is represented in degrees in principal axes (vertical axis (yaw), transverse axis (pitch) and longitudinal axis (roll));
- Acceleration and speed mode can be automatic or manual, and acceleration and speed level indicates the rate of change applied to the respective physical measure;
- Altitude, also collected by GPS, meaning the height of the drone in relation to the sea level.

Resources that have only status have been grouped, as seen in Figure 6. *Takeoff* and *Landing* resources represent operations for drone take off and landing, respectively. The *SafeLand* resource deals with an operation to force landing in situations that may be considered dangerous for the drone (for instance, low level battery, hard-

ware/software errors or GPS failure). Finally, *EmergencyStopMode* resource manages operation to disarm the motors if the drone has a critical error during the flight. The other resources are associated mainly with automatic high-level operations that can have configurable attributes and status, which represents the current situation of the drone operation.

The *Mission* resource models an automate flight. It exposes waypoints, a set of coordinates of interest (three-dimensional position) that the drone will fly to, and status, which indicates the situation of the mission. The *RTH* resource represents the execution of the return to home operation. It is a very useful safety feature that helps bringing the drone back to a safe, accessible landing location (Home Point). Therefore, this resource has an attribute to represent the home point and another one that determines the situation of this operation (status).

The *EnergySavingMode* resource sets battery level thresholds, which should trigger warnings to the pilot or to pre-defined operations (for instance, safe landing and RTH). Flight wrappers switch management to manual or automatic flight mode. The former is possible via remote controller to manipulate limited features of the flight, while the latter is performed through high-level flight automation via the *Mission* resource. Both resources have the status attribute representing the current status of each operation.

The *Sensor* component is responsible for telemetry and monitoring activities supporting the component *Control*. Some common sensors in drones are accelerometers, gyroscope, magnetic compass and barometer. The *accelerometer* measures the acceleration force that the drone is subjected to in all three axis X, Y and Z. Besides, it is also used to estimate linear acceleration in horizontal and vertical direction. This data can be used to calculate speed and direction. The *Gyroscope* detects angular velocity in three axis, thus calculates angle in pitch, roll and yaw. It controls the speed dynamically to provide stability to the drone and to also ensure that the drone rotates at the exact expected angle. The *Compass* provides information of magnetic field and then detects geographical direction. *Magnetic* material can create variables in the sensor reading, thus calibration operations can be used to deal with possible interference and avoid accidents. The *Barometer* converts atmospheric pressure into altitude, thereby it helps to achieve the desired altitude. The *IMU* represents the inertial measurement unit sensor, which measures linear and angular velocity and attitude using data from other

sensors, such as accelerometer, gyroscope and sometimes magnetometers. That unit works when the GPS is unavailable, such as inside buildings or during electronic interference. The resources that represent those sensors were grouped and represented, as shown in Figure 6, and they contain a status attribute to provide the current status of the respective sensor.

The *ObstacleAvoidance* resource deals with the sensor that detects objects near the drone. The *Global Positioning System* (GPS) resource is used to determine the ground position of the drone. Both resources have result and status attributes. The first one represents the detection or not of an object and the other one represents strength of the GPS signal, respectively. The latter (status attribute) represent internal state of each sensor.

*Weather* sensing is an important key in decision making on a flight plan since it provides information about wind conditions. The *Anemometer* resource handles the sensor that measures wind speed and direction, and provides information about the anemometer state. The *Thermometer* resource wrappers the sensor that collects external temperature and also gives thermometer status.

*Hygrometer* and *SmokeDetector* are resources that handle sensors for more specific applications. The former deals with the sensor used to measure relative humidity (humidity attribute), which is the amount of vapor in the air compared to the maximum amount possible. The latter manages the sensor that detects smoke (result attribute), indicating fire in the vicinity. Both features have a status attribute to display the status of each sensor.

Drones have limited battery capabilities, therefore it is important to monitor and to ensure that they consume low power. The *Battery* resource provides information about the capacity, voltage, electrical current, actual percentage and status of battery battery.

The *Actuator* component wrappers devices that convert energy into mechanical movement in order to iterate the drone with the environment. Camera and gimbal devices allow effective and flexible aerial imaging missions for a wide range of applications. The former captures photos and videos, while the latter supports camera stabilization allowing the camera to remain horizontal regardless of the motion performed. The *Camera* resource exposes the cameras action and status. It has a

focus point attribute that allows the camera to fix the focus in a geographic position and automatically remain it during the flight. In addition, this resource has a status attribute that gives the current status of the device. The *Gimbal* resource treats gimbal motion control through rotation (rotation angle), axis (axis that the rotation is being applied) and control (whether operations are automatic or manual). This resource has a status attribute.

The *Light* resource provides accesses information about on-board lights. Usually, lights are necessary to minimize the chance of collisions, mainly during a night flight. The *Landing gear* device prevents the drone from touching the ground when landing or taking off, as well as the gimbal/camera. Furthermore it absorbs the landing shock on any sudden landings. The *Payload* resource expresses the device that provides package/payload delivery.

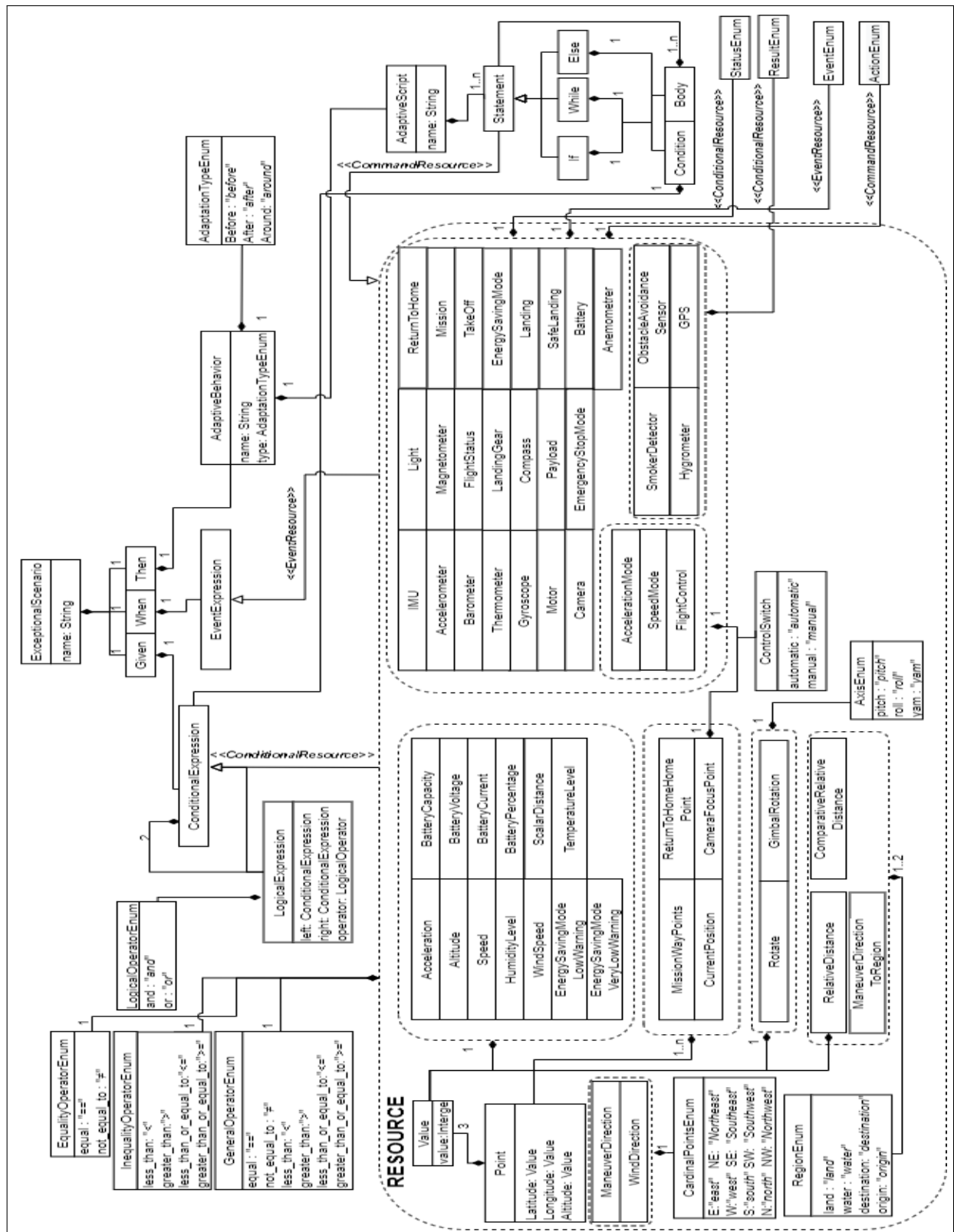
### 6.3 ABSTRACT SYNTAX

The abstract syntax is the specification of the language grammar and, thus, defines all valid sentences of that language (BRAMBILLA et al., 2017). Usually, the abstract syntax is represented by the metamodels, which are normally based on object-oriented models, such as classes, attributes and associations. Figure 7 provides a visualization of the high-level specification of the elements of the DRES-ML syntax using UML class diagrams. These elements are based on the resources listed in the domain analysis presented in last section and represent a set of abstractions to express a model for drone exceptional scenarios.

To reduce the size of the metamodel illustrated in Figure 7, boxes group elements containing common relationships. For example, inside the dashed resources box, there is another dashed box that contains elements from the acceleration resource to the temperature level resource (see Figure 7), this box represents a common relationship with integer value, i.e., to model these expressions it is necessary to use an integer value. In addition, also inheritance elements are omitted, such as enumerators of status, results, events, and actions (they are detailed later). Thus, detail of the elements of the abstract syntax can be found in Table 1 in appendix A and in an public spreadsheet<sup>1</sup>.

<sup>1</sup> <https://cutt.ly/fjw07ws>

Figure 7 – Metamodel of the DRES-ML



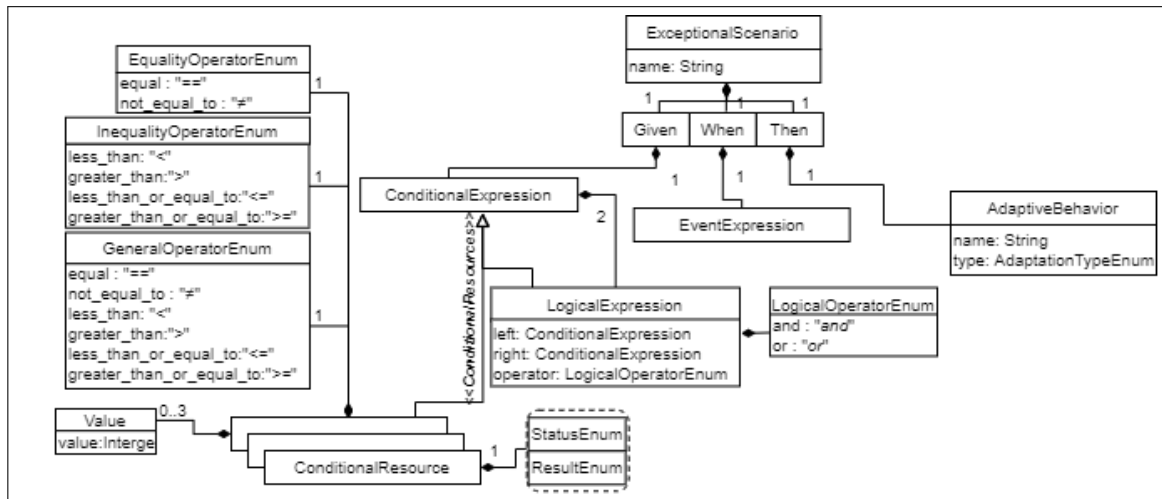
Source – Prepared by the author



For the exceptional scenario (*ExceptionalScenario*) specification using DRES-ML, it is required to define a Given-When-Then structure provided by the BDD technique. In addition, it is required to define a name for the modeled exceptional scenario. This attribute is used as an unique identifier to differentiate exceptional scenarios. Figure 8 shows a snippet of the DRES-ML metamodel focusing on exceptional scenarios.

The *Given* concept expresses the context states including the drone resources and the environments through conditional expressions (*ConditionalExpression* concept). The essential characteristic of a conditional expression is that its result produces a binary value (true or false). In addition, it has concepts that assist in the construction of conditional expressions: *ConditionalResource* and *LogicalExpression* concepts.

Figure 8 – Metamodel of Exceptional Scenario



Source – Prepared by the author

The *ConditionalResource* concept takes as a basis the domain analysis, summarized in the previous session, to create conditional expression representing specific drone resources and environmental conditions. Those concepts represent information that can be obtained from monitored components, i.e, drone on-board equipments (e.g. sensors) or any external sources from which the drone may obtain information (e.g. control tower or web services), information on the status of operations and actuators. These conditional resources enable to formulate conditional expressions using the obtained information. They allow comparative operators “==” and “≠”; “>”, “<”,

"<=" ">=", specified by *EqualityEnum* and *InequalityEnum* enumeration<sup>2</sup>, respectively, and *GeneralEnum* enumeration incorporating all operators, as shown in Figure 8. Finally, conditional expressions contains compared values, which are enumeration, such as *StatusEnum* and *ResultEnum*, and integer values. The *StatusEnum* manifests the internal state or status monitored by the resource concepts, while the *ResultEnum* represents a monitored outcome produced by resources. The conditional expressions for resources are disposed in the dashed box, labeled Resource, in Figure 7.

In order to exemplify the grammar of a conditional expression of a resource, the list 4.1-4.3 presents the BNF of anemometer resource (sensor used for measuring wind speed and direction). This example focused on the expression that deals with the wind speed, which can generate a condition for the Move Aside exceptional scenario presented in Session 6.1. In addition, it is worth mentioning that in order to avoid ambiguity in the application of operators, brackets are used in expressions (line 4.1).

$$\langle \text{WindSpeedCondExp} \rangle \models ( \text{Wind.speed} \langle \text{GeneralEnum} \rangle \langle \text{Value} \rangle \text{m/s} ) \quad (6.1)$$

$$\langle \text{GeneralEnum} \rangle \models == \mid \neq \mid > \mid < \mid \leq \mid \geq \quad (6.2)$$

$$\langle \text{Value} \rangle \models \{ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \} \quad (6.3)$$

In the BNF below, it was exemplified conditional expression grammar that represents the anemometer resource with the attribute status. The pre-defined values of status are indicated by the derivation of the *StatusEnum*, more specifically, the *AnemometerStatusEnum*. It represents the status for the described resource enabling, such as, ACTIVATE, DEACTIVATE, and ERROR as possible values. "**(Anemometer.status == ERROR)**" is an example of conditional expression produced through that syntax.

$$\langle \text{AnemometerStatusCondExp} \rangle \models ( \text{Anemometer.status} \langle \text{EqualityEnum} \rangle \langle \text{AnemometerStatusEnum} \rangle ) \quad (6.4)$$

$$\langle \text{EqualityEnum} \rangle \models == \mid \neq \quad (6.5)$$

$$\langle \text{AnemometerStatusEnum} \rangle \models \text{ACTIVATED} \mid \text{DEACTIVATED} \mid \text{ERROR} \quad (6.6)$$

The *LogicalExpression* concept has three attributes (shown in Figure 8): *left* - it is the same type as the parent concept, and represents the clause on the left side of

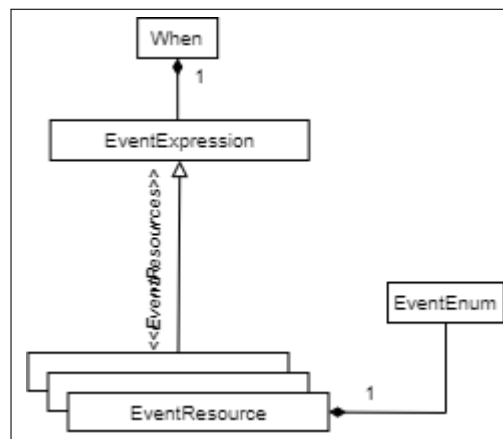
<sup>2</sup> Enumeration concepts are sequence of pre-established keywords for each specific concept.

an operator in an conditional expression; *operator* - it is an element type that represents logical operators in an conditional expression, such as “*and*” and “*or*” (specified by the *LogicalOperator* enumeration); and *right* - it is also same type as the parent element, and represents the clause on the right side of the operator in an conditional expression. It is worth mentioning that concept enables the use of combined logical and resource expressions, giving the possibility of more complex expressions.

The *When* concept defines an event that is being invoked, more specifically, an action/command of a resource that is being executed. These events are executed through commands performed by the remote controller, pre-defined operations (for example, missions) or any other flight control command source. These events are called to control/managing actuators and sensors present in the drone.

The *EventResource* concepts specifies events to a type of resource. It is complemented with a set of predefined values that can specify the type of event through the *EventEnum* element, as seen in Figure 9. These values are mapped in the domain analysis phase which presents a range of event that a drone resource can execute.

Figure 9 – Metamodel of When clause



Source – Prepared by the author

*EventResource* was based on the Joinpoint concept from the AOP paradigm. Whenever the drone executes the event specified in the *When* clause, the adaptive behavior modeled in the *Then* element is performed, which means that events serve as a trigger for adaptation. A wildcard value (“\*”) may be required to represent the execution of any action for an event resource.

The list below presents the grammar to specify events related to the safe

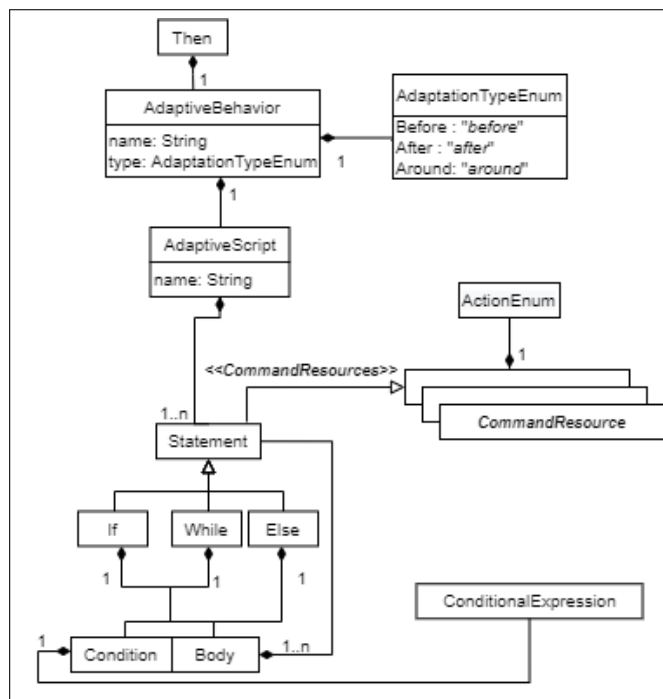
landing resource. The mapped *ResourceEvent* to the respective resource are *ACTIVATES*, *DEACTIVATES*, *CALIBRATES* or *""*. That example permits to generate the event shown in the *When* clause of the exceptional scenario move aside.

$\langle \text{SafeLandingEvent} \rangle \models \langle \text{SafeLandingEventEnum} \rangle \text{ SafeLanding}$  (6.7)

$\langle \text{SafeLandingEventEnum} \rangle \models \text{STARTS} \mid \text{PAUSES} \mid \text{CANCELS} \mid \text{RESUMES} \mid *$  (6.8)

The *Then* concept contains information on the adaptation steps required for the exceptional scenario. It contains the *AdaptiveBehavior* concept that specifies the type of adaptation that will be performed, and makes an association with the *AdaptiveScript* concept that represents adaptation itself (shown in Figure 10). The idea behind *Adaptive Script* is to create a "forced" sequence of command flow (sub-routine) to represent the adaptive behavior. Through an identification (attribute name), it is possible to realize the relationship between these two concepts. This attribute is defined for *AdaptiveScript*, thus making it to be a unique identifier (signature). Then, when the *AdapitativeBehavior* concept defines in its structure the identifier of an already defined *AdaptiveScript*, it means that the adaptive behavior is modeled in the respective adaptive script.

Figure 10 – Metamodel of Then clause



Source – Prepared by the author

Another important structure for *AdaptiveBehavior* is the *AdaptationTypeEnum* concept. It is pre-established that adaptations are based on AOP advice concepts: *before*, *after* and *around*. The *before* and *after* types have a similar behavior because they do not replace the specified behavior in the *When* concept. The *before* implies that the adaptive behavior will be invoked at runtime, before the predefined behavior. This means that adaptation will be performed before the event call defined in the *When* concept. The *after* type, in turn, invokes adaptation at runtime, after being called in the event defined in the *When* concept. The *around* type surrounds the command call defined in the *When* clause. In other words, the predefined behavior invoked by the event specified in the *When* clause is replaced at runtime by the behavior modeled in the *Then* concept.

The *AdaptiveScript* concept includes a sequence of *Statement* concept, which is derived from other concepts, for instance, the *CommandResource* that represents commands coming from resources. These commands are entities that, when executed, can produce an effect on the environment and the internal state of the drone. Thus, the result of a command can be a different state from the last state modified by the effect. Therefore, commands have the purpose of changing the state of the resources. The *ActionEnum* is a high-level element created in order to represent the possible commands for the resources. The resources that allow commands also were included in the dashed box Resource in Figure 7.

Inside the *AdaptiveScript*, it is possible to define a command or sequence of commands through *CommandResource*. That sequence specifies a command execution flow in which a next command is only executed when the current command is ending. In order to determine an alternative or conditional flows, other types of statements were derived, such as, *If*, *If-Else* and *While*.

The *If* concept allows the specification of alternative flows. The satisfaction of a condition (*Condition* concept) to allow the execution of a specific flow (specified in the *Body* concept). The condition is formed by the *ConditionalExpression* concept resulting in a value of true or false. The *If-Else* concept is an extension of the *If* concept. If the result of the condition is the value true, then is executed the flow of actions in the *Body* concept, otherwise the else-statement body is executed with the alternate flow. The *While* concept allows the repetition of the a given flow execution (*Body* concept)

until the satisfaction of the specified condition (*Condition* concept).

The list below shows an example of a resource for commands. It is shown the maneuver to a direction command resource. This resource is modeled with values predefined cardinal directions to allow maneuvering actions for a specified direction. This command resource composes the adaptive behavior present in the exceptional scenario move aside.

$$\langle \text{DroneManeuverDirectionToRegionCommand} \rangle \models \mathbf{Drone.direction} \langle \text{DirectionEnum} \rangle \quad (6.9)$$

$$\langle \text{DirectionEnum} \rangle \models \text{NORTH} \mid \text{EAST} \mid \dots \mid \text{SOUTH\_WEST} \quad (6.10)$$

## 6.4 DRES-ML MODELING ENVIRONMENT

In this session, the technology and resources used to implement the modeling environment for the DRES-ML are explained. In addition, an example of an exceptional scenario modeled using the environment is presented.

The implementation of DRES-ML was based on the JetBrains Meta Programming System (MPS)<sup>3</sup>. The MPS is an open-source project that provides the tools to design domain-specific and general-purpose languages (CAMPAGNE, 2014). It is a well known example of language workbenches, which are software engineering tools that ease the implementation of new languages and associated tools (IDEs, debuggers, consoles, etc.) (ERDWEG et al., 2015).

The Editor concept is an important aspect used in a DSL based on a language workbench according to Martin Fowler (FOWLER, 2005). It represents a graphical or textual representation of the abstract syntax and allow the user to manipulate the Abstract Syntax Tree (AST) through *projections* (VOELTER et al., 2013). Software engineers can modify the AST directly when editing a program<sup>4</sup>, thus that process does not involve parsers.

For specifying an exceptional scenario using the DRES-ML modeling environment, it is necessary to create an exceptional scenario element<sup>5</sup>. As aforementioned, an exceptional scenario is outlined based on the Given-When-Then structure, therefore

<sup>3</sup> <https://www.jetbrains.com/mps/>

<sup>4</sup> A program is a set of instructions for the computer using a programming language.

<sup>5</sup> An element is a node of an AST that represents a construction in the source code of the program.

a template to model that structure is pre-established when creating that kind of element, as shown in Figure 11. The *<value>* placeholder underlined in red represents entry points for software engineers filling the exceptional scenario. These visual indicators are constraints implemented in the modeling environment to prevent the engineer filling in improperly the current node. The value in the placeholder presents an error message containing necessary information to explain the problem. In Figure 11, error messages are presented indicating the absence of the value of an attribute or a child node.

Figure 11 – Template for exceptional scenario in the DRES-ML

```

Exceptional Scenario <no exceptional scenario name>
Given: <no given>
When: <no when>
Then: execute <no type of adaptation> <no reference to adaptation script>
      <no adaptation script name> :
      << ... >>

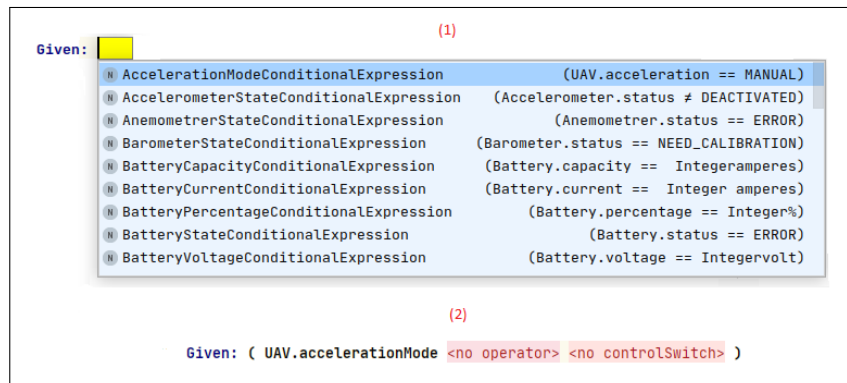
```

Source – Prepared by the author

As shown in Figure 12, it is possible to view enabled model elements to fill the respective fields through the code-completion pop-up menu (Ctrl+Space). This is a useful feature because it prevents mistakes and helps the user to correctly select the possible elements based on their type and the grammar rules, therefore ensuring the correctness of the modeling good based on the abstract syntax. In addition, along with the autocomplete menu, there are examples of each model element aiming it facilitating the understanding of the usefulness of a specific model element. After the engineer selecting the desired element to compose the exceptional scenarios, the respective node is shown as a template to be specified the required values through placeholders as shown by 12.

A pattern of style and colors was designed for the elements of the nodes (see Figure 13). Elements shown in bold style are immutable values, while elements presented in regular style are variable values that the software engineer can input. In the template the exceptional scenario reserves expression is shown in red, while the Given-When-Then terms are presented in blue. The attributes colored in orange represent the specification of the type of adaptation (before, after and around). Attributes in green color and italic style indicate the reference scope between elements (reference between the *Then* clause and the adaptation script). Finally, statement elements (if,

Figure 12 – Autocomplete menu in the DRES Modeling Environment



Source – Prepared by the author

while, if-else and commands) are indicated in blue. Figure 13 shows the implementation of the Move Aside exceptional scenario presented in Session 6.1 using the DRES-ML and its modeling environment.

To specify the conditions of the environment and the drone's internal state for this exceptional scenario, *Relative Distance* and *Wind Speed* conditional expressions were associated using conjunctions (logical operator "and") inside the *Given* clause. The former compares if the drone's distance from a region/area with water is 0 meters and if the distance from the destination region's drone is greater than or equal to 2000 meters. The textual values that are in upper cases specify regions for conditional expressions. These comparisons are carried out through equality operators ("=="). The last compares if the wind speed is less than 5 meters per second using the inequality operator ("<"). As expected, the standardization specified in the modeling environment resource elements and their attributes is in bold as for *Distance* and *Speed* attributes of *Drone* and *Wind* resources, respectively.

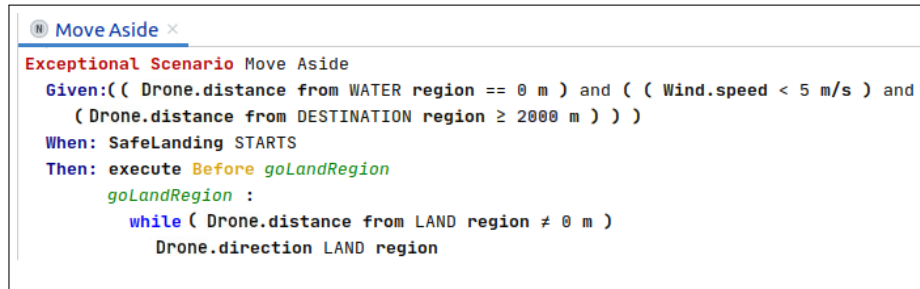
The *When* clause specifies the operation of initialization of the safe landing as the trigger event. Operation, such as "starts", "pauses", "cancels," and "resumes", are presented in upper case.

The *Then* clause determines "before" values for the type of adaptation and "goLandRegion" as a reference for the adaptation script. The next is to establish adaptive behavior through a script identified as "goLandRegion". To execute the behavior defined in the Move Aside exceptional scenario, a *While* statement element uses a guard represented by a conditional expression that compares whether the drone's distance to the land region is different from 0 meters, that is, the drone is not over a land



region. Besides, the while element's body contains another statement that represents a command specifying that the drone flies to a land region.

Figure 13 – Move Aside exceptional scenario using DRES-ML



```

Exceptional Scenario Move Aside
Given:(( Drone.distance from WATER region == 0 m ) and ( ( Wind.speed < 5 m/s ) and
( Drone.distance from DESTINATION region >= 2000 m ) ) )
When: SafeLanding STARTS
Then: execute Before goLandRegion
      goLandRegion :
        while ( Drone.distance from LAND region != 0 m )
          Drone.direction LAND region
  
```

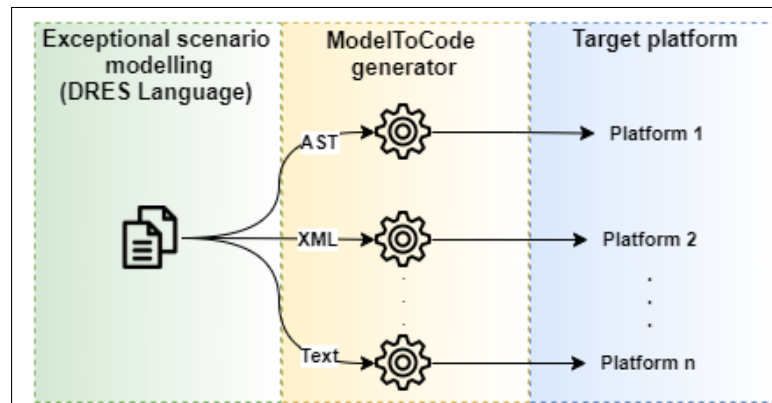
Source – Prepared by the author

## 6.5 MODEL TO TEXT - CODE GENERATION PROCESS

The DRES-ML has been designed to be attached with different *ModelToText* (M2T) generators, which means that it can generate code or scripts representing the specified exception scenarios to distinct *target platforms*. This section provides an overview about M2T generators.

A *generator* is a translator that maps elements of the model to elements of another model, thus allowing the conversion between them (VOELTER et al., 2013). This translation can be achieved by either a model-to-model transformation, generating a new model as output, or a model-to-text transformation, generating text or source code as output. A *target platform* is where the generated code has to run on at the end of the process, then it needs to be "platform understandable". It is common to assume that there are several target platforms, therefore each target platform has to have a specific generator.

Figure 14 depicts the three phases of the code generation generic process. The first phase addresses the specification at the exceptional scenario using the DRES modeling environment. The second phase, it is required the use of the created generator that allows translation of program encoded in the DRES language into constructions encoded in the output language. There are several appropriate languages for defining transformations that can be used for defining mapping in terms of a rule-based

Figure 14 – *ModelToText* process

Source – Prepared by the author

templates, such as Acceleo<sup>6</sup>, TextGen<sup>7</sup> and Xtext<sup>8</sup>. They have placeholders that are applied on the elements in the produced model code. For example, the TextGen is a mechanism proposed by MPS JetBrains that executes transformation processes (CAMPAGNE, 2014), enabling the creation of generators. It contains templates to print out text, to transform elements from AST (the artifacts accessed directly by the modeling environment) of the program written with DRES-ML into text values and gives the output with a layout.

Model interpretation is another transformation process that the output code does not generate directly from the model (BRAMBILLA et al., 2017), such as ANTLR<sup>9</sup>. Then, a generic generator is implemented that performs a model parser on-the-fly with an interpretation approach (the same way that interpreters do to interpret programming languages). This process analyzes string of symbols or data structures, conforming to the rules of a defined grammar. The parser process is made possible by exporting artifacts in either XML (Extensible Markup Language) or even text by the DRES-ML environment.

The benefits of this approach is that it does not depend on transformation tools avoiding the learning required to use them. For example, TextGen and Acceleo are restrict to JetBrains and Eclipse IDEs, respectively, and requires knowledge of its language to implement the generators. Moreover, it is worth mentioning that, to create a generator, an expert is required. Who should know the syntax elements of the DRES-ML

<sup>6</sup> <https://www.eclipse.org/acceleo/>

<sup>7</sup> <https://www.jetbrains.com/help/mps/textgen.html>

<sup>8</sup> <https://www.eclipse.org/Xtext/>

<sup>9</sup> <https://www.antlr.org/>

and the language and API used in the target platform to be able to make an appropriate transformation process.

The last phase consists of generating the code artifact that should be added to be executed in the target platform. It is common to assume that there are several target platforms, therefore the execution generator can be switched to a new target platform. However, the model-to-code process may not be smart enough to generate a complete output code directly executed in a specific target platform. Nevertheless, the software engineer can still benefit from a generation approach by creating a partial implementation of the system, thus completing the code manually to obtain the full functionality on the platform. Another point to be highlighted is that the target platform must be able to perform adaptive behaviors at runtime. To do that, it can use mechanisms such as dependency injection (FOWLER, 2004), plugin-based approaches (WERMELINGER; YU, 2008), design patterns (GAMMA, 1995) or Aspect-oriented programming (AOP) (KICZALES; HILSDALE, 2001).

## 7 EVALUATION

The main goal of this work is to create a language and an environment for modelling exceptional scenarios available for different drone situations. This chapter presents an evaluation with a focus on analyzing the applicability of DRES-ML to a variety of possible exceptional scenarios in a given application and their execution on a target drone simulator platform.

### 7.1 THE DRAGONFLY TOOL

*Dragonfly* simulator is an open source and extensible Java tool available at GitHub <sup>1</sup>. It supports creating environments for simulating the behaviour of a set of drones in drone-based applications. In this section, it is described a overview the of *Dragonfly*.

#### 7.1.1 Interface

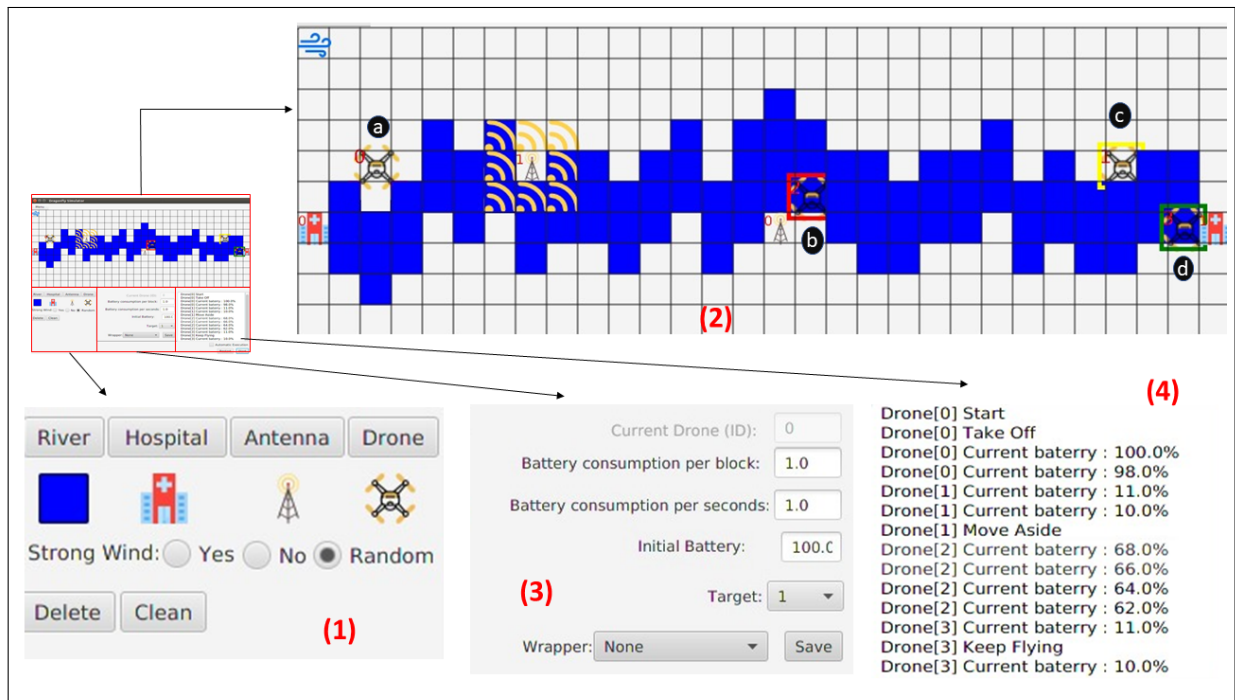
Figure 15 shows the interface of the *Dragonfly* tool. As shown in the figure, the interface is divided into four main panels, namely: *graphical elements panel* (1), *drone flight environment panel* (2), *drone properties panel* (3), and *trace log panel* (4). In Figure 15 each panel is expanded for better visualisation.

The graphical element panel (1) provides a set of graphical elements that can be used to represent different environments of various applications. The current version provides elements to represent rivers, hospitals, communication antennas, and drones. Other elements can be easily added to the tool and is part of a future extension of *Dragonfly*.

The user can insert the graphical elements in the drone flight environment panel (2), which is used to create the environment in which the tool will simulate drones in an application. This panel consists of a grid layout in which each cell can contain one or more graphical elements.

The example in Figure 15 illustrates an environment with two hospitals: one on the far left and the other one on the far right, representing origin an destination of a flight. The pathway of this scenario consists of a river. The environment also has two

<sup>1</sup> <https://github.com/DragonflyDrone/Dragonfly>

Figure 15 – Screenshot of the *Dragonfly* simulation tool

Source – Prepared by the author

communication antennas to simulate the emission of signals that may interfere with the journey of the drones. There are four drones flying in the environment: one drone (a) with its original specifications (i.e., without having any wrapper), and three drones ((b), (c), and (d)) weaved with wrappers representing different adaptive behaviours in case of exceptional situations during their respective journeys.

The drone property panel (3) allows setting of initial values of some of the resources of the participating drones. It also associates each participating drone with one or more wrappers representing behaviour adaptation functionalities, in case of exceptional situations. Examples of initial resource values are initial battery level and battery consumption rate.

The trace log panel (4) shows the current status and activities of each participating drone (identified by a number), during runtime simulation. An example of a current status is concerned with a drone's battery level, while examples of activities are concerned with take off, fly, move aside, and land.

### 7.1.2 Execution of flight simulation

When using the simulator, the first step consists of constructing the environment of an application. In this case, the user inserts graphical elements by selecting an element from the graphical element panel (Figure 15 (1)) and choosing a specific position in the grid of the drone flight environment (Figure 15 (2)), where the element should be placed.

The user needs to configure the following properties for each drone inserted in the environment: battery consumption rate per block and per second, initial battery level, and target element (i.e., the place to where the drone will fly). In addition, he/she can associate a drone with an available wrapper. Afterwards, the user chooses the mode that the drone should fly. In the case of automatic pilot mode, when the simulation starts, the drone will execute the shortest path to reach the target destination. In the case of user pilot mode, the user will manoeuvre the drone by using the keyboard. The commands to manoeuvre a drone are available at the *Dragonfly's* GitHub repository.

The final step consists of starting the simulation by clicking the “Start” button, which triggers the execution of each inserted drone simultaneously. The currently implementation of the tool supports up to 400 drones flying at the same time, with and without wrappers. If the user has chosen to pilot the drone manually, the available commands to be executed are: turn on/off the drone, take off, move (up, right, down, and left), and land.

### 7.1.3 Tool extension flow

The current version of *Dragonfly* is extensible in terms of new graphical elements to represent other environment settings and in terms of new wrappers to represent new exceptional situations. For the creation of new graphical elements, it is necessary to create one class in each layer of the architecture for each new element. It is also possible to associate an image with the new element in its correspondent view class. For creating new wrappers, it is possible to implement the behaviour in aspect-oriented programming to represent the new exceptional situations and associate drones with the new wrappers.

## 7.2 PROOF OF CONCEPT

### 7.2.1 Motivating Example

Due to the large ecological degradation resulted by forest fires that occurred in 2020 (XU et al., 2020), an application example was elaborated using drones for forest fire detection and monitoring. This application provides a continuous remote recorder, geographic position and smoke detection through the drone and its sensors, thus enabling to detect fire areas.

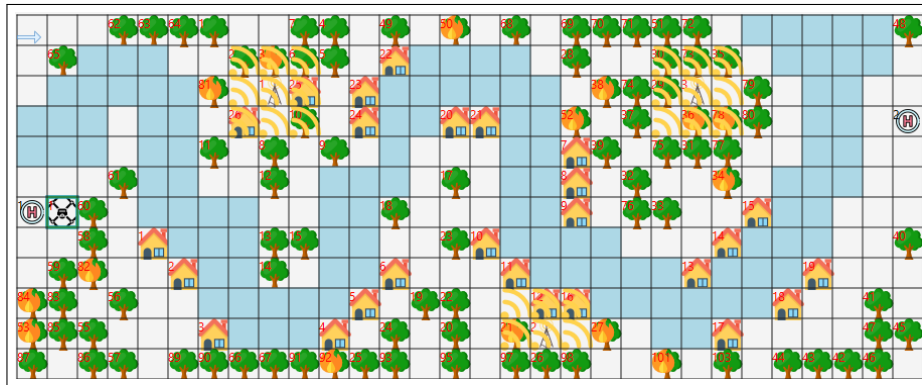
This supervision is intended to collect data about the evolution of fire deviation and to assist teams in the control of fires, thereby helping to protect the environment and riverside communities. Given that, the objectives of the application is (i) to monitor fire outbreak and (ii) to minimize drone losses. The exceptional scenarios were elaborated based on these goals. The drone will realize monitoring its resources, such as camera, smoke detector and GPS.

The environment of the proposed drone application was created using the Dragonfly tool and is shown in Figure 16. It illustrates a forest crossed by a river, represented by the blue squares, and regions of land. In addition, there are houses representing riverside communities and antennas that perform transmission and reception radio and TV signals. The drone will start at a home point that is located at the west side of the forest (represented by a circle with a letter H inside), it will fly over the forest following the flight plan and monitoring the environment, and it will land at the destination point (also indicated by a circle with a letter H) in the east side of the forest.

The drone flight can be carried out either automatically or manually. In the first one, the pilot creates the flight plan and loads it on the drone before starting the mission, while in the other one, the pilot can control the drone manually using a remote control. If necessary, the pilot can change the drone's flight mode.

The drones of this application behave following the scenarios of Figure 17, which is an extension of the example in described in Session 6.1, that uses the MSC-based exceptional scenario representation proposed by Maia et al. (2019), combined with new monitored variables and new scenarios. The origin distance, represented by *od*, and the *status* of operations and sensors of a drone are new variables that are obtained through the states of both sensors and drone. *KeepFlying*, *SafeRTH*, *RTH*,

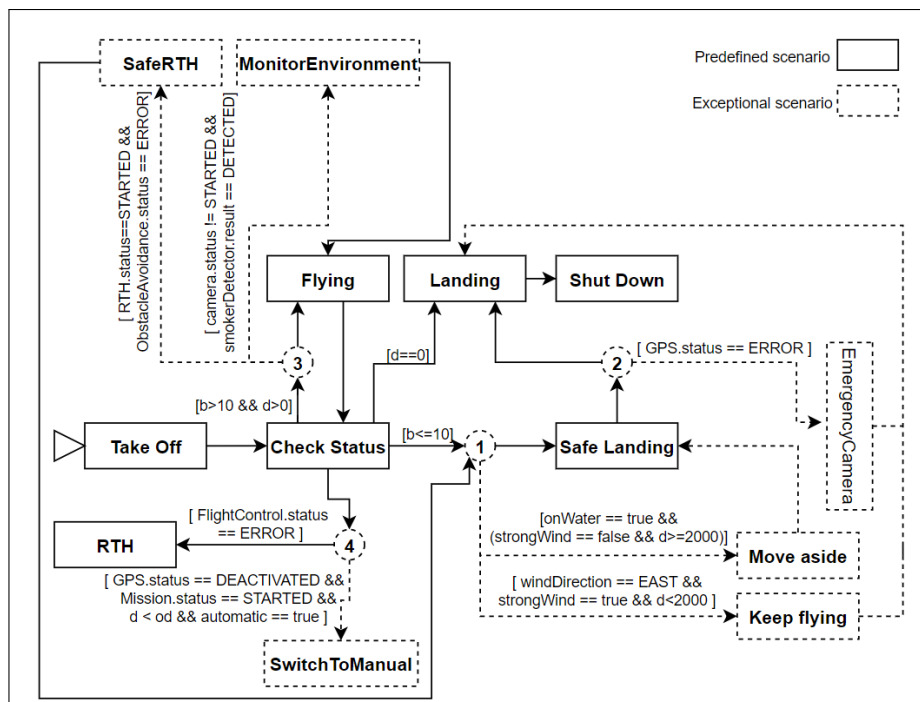
Figure 16 – Simulation of the monitored environment.



Source – Prepared by the author

*SwitchToManual*, *MonitorEnvironment* and *EmergencyCamera* are the new modeled scenarios, which are described as follows along with their corresponding specification in the DRES-ML.

Figure 17 – Scenarios of example application



Source – Prepared by the author



### 7.2.1.1 Exceptional Scenarios Specification

#### 7.2.1.1.1 *KeepFlying*

The purpose of the *KeepFlying* exceptional scenario is to enable the drone to continue flying towards the destination when there are favorable conditions instead of a safe landing operation. This situation, it is understood as favorable conditions when the strong wind moves towards the destination location (east), and the distance to the destination location is less than 2000 meters, enabling the drone to reach its destination. And after this new behavior is realized, the drone can perform its predefined behavior flow.

Figure 18 shows *KeepFlying* exceptional scenario specified using DRES-ML. The conditions for triggering the adaptation should be implemented in the Given clause. Therefore, to compose the conditions shown in Figure 17, two logical expressions were needed using the conjunctive operator "and", enabling the implementation of three conditional resource expressions.

Figure 18 – Keep flying exceptional scenario modeling with DRES-ML

```
Exceptional Scenario Keepflying
Given: ( ( Drone.distance from DESTINATION region < 2000 m ) and
        ( ( Wind.speed ≥ 5 m/s ) and ( Wind.direction == EAST ) ) )
When: SafeLanding STARTS
Then: execute Around goDestination
      goDestination :
        while ( Drone.distance from DESTINATION region ≠ 0 m )
          Drone.direction DESTINATION region
```

Source – Prepared by the author

The *Relative distance* conditional expression is used to compare the drone's distance to a destination region ("DESTINATION") with a distance in meters (2000) using a comparative operator ("less than"). To compare the wind speed, the *wind speed* conditional expression is used, which also uses a comparative operator ("equal to") and a speed value in m/s. The *wind direction* conditional expression allows implementing the condition that verifies the wind direction, then it is passed an operator ("equal to") and a direction value ("EAST").

The When clause is implemented with the safe landing start event (indicated

by the *Safelanding* scenario) to represent that the interception must be performed when this event is invoked (see joint point 1). Note that if the exception scenario is executed, it replaces the predefined event. Therefore, the adaptation strategy has to be of type "around", as indicated in the When clause. In addition, this clause links to an adaptive behavior script called "goDestination".

Finally, the adaptation script is implemented using command resources and statements. The modeled script must perform a repetition of the command that makes the drone to fly towards the destination region while it does not arrive at the destination. This modeling was performed using the *While* statement.

The *while stop* condition was implemented with *relative distance* conditional expression, in which the distance relative to the destination region ("DESTINATION") is compared to 0 meters using the equality operator "not equal to". And drone while's body was specified with a *maneuver direction* command, which makes the drone to maneuver the flight towards the destination region ("DESTINATION").

Thus, the adaptation entitled "goDestination" collaborates with the objective (ii) of the application, enabling the drone to reach its destination.

#### 7.2.1.1.2 *SwitchToManual*

A proper operation of the GPS is necessary for the drone to follow the controller's flight plan since the waypoints are defined with latitude, longitude, and altitude, the information provided by that sensor. Thus, another situation that can cause an RTH operation is when the drone has a faulty connection to the GPS. Therefore, the drone should be gliding for a while or even perform some random movements until a good signal is reestablished. Based on that, the *SwitchToManual* scenario was defined, causing the pilot to take control of the drone and preventing the drone from performing the RTH when it is close to the destination (see Figure 17).

The implementation of SwitchToManual using DRES-ML syntax was shown in Figure 19. The condition that guards this scenario is implemented inside of the *Given* clause using the associated expressions that handles comparisons of the status of the GPS and flight resources, and comparative expression to the relative distance of drone.

To intercept the RTH scenario (join point 4), the *return to home starts* event is implemented in the *When* clause. In order to override the execution of the RTH

Figure 19 – Switch to Manual exceptional scenario modeling with DRES-ML

```

Exceptional Scenario SwitchToManual
Given: ( ( GPS.status == ERROR ) and ( ( Drone.distance from DESTINATION region < ORIGIN region ) and
      ( AUTOMATIC FlightControl.status == STARTED ) ) )
When: RTH STARTS
Then: execute Around turnManual
      turnManual :
        START MANUAL FlightControl

```

Source – Prepared by the author

predefined scenario with the adaptation called "turnManual", it was used the adaptation type "Around" in *Then* clause.

The script in the *Then* clause defines the *start manual flight control* as a necessary adaptation. Thus, the drone control changes to the manual, enabling the pilot to take over the flight and to conduct it in the safest way possible. Thus, this adaptation collaborates so much with the objective (i) because the pilot can enable the continuity of the environment monitoring.

#### 7.2.1.1.3 SafeRTH

The Return to Home (RTH) operation (*RTH* scenario) is a useful drone protection feature. When a drone control error occurs (for example, signal lost), it ascends to the pre-defined RTH height and starts to flying back straight to the initial location (home point). However, if the height has not been adjusted correctly to avoid tall obstacles, such as trees, antennas, and others, and the obstacle avoidance sensor does not work correctly, the drone can collide.

Thus, the *SafeRTH* exceptional scenario performs a safe landing operation to guarantee the drone's safety in the situation where there is a malfunction in the collision sensor during the RTH operation. It is worth mentioning that this adaptation can bring new situations that require other adaptations, such as *MoveAside* or *Keepflying* (as can be seen in Figure 17).

The modeling of this described scenario using DRES-ML syntax can be seen in Figure 20. The *Given* clause was implemented using a logical expression containing inner expressions about the status of the RTH and the collision sensor, associating them by an "and" operator.

The first internal expression compares, using the "equal to" as comparison

Figure 20 – SafeRTH exceptional scenario modeling with DRES-ML

```

Exceptional Scenario SafeRTH
Given: ( ( RTH.status == STARTED ) and ( ObstacleAvoidance.status == ERROR ) )
When: Drone.maneuvers *
Then: execute Around newSafeLand
      newSafeLand :
        START SafeLanding

```

Source – Prepared by the author

operator, the status of the *return to home* operation with the "STARTED" status. This condition validates whether the drone is performing a return home operation. The last internal expression checks whether the status of the obstacle sensor is equal to the error value, representing the situation in which that resource is faulty. These expressions represent the implementation of the guard for SafeRTH.

As can be seen, join point 3 (see Figure 17) allows interception in the predefined Flying scenario. This is modeled within the When clause using the drone maneuver direction event without defining a specific direction (wildcard "\*"). This strategy is important because, whatever the direction of the drone's maneuver is, the intercept point can be used.

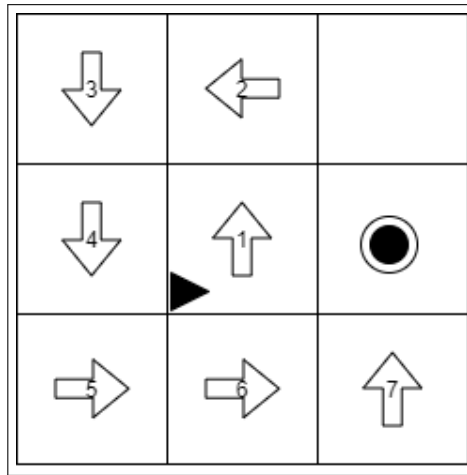
In the Then clause, "Around" is used to prevent that the drone continues maneuvering and make it to perform the adaptive behavior modeled within the script named "newSafeLand". The script that defines adaptive behavior forces only starting a safe landing command, assuming that making an safe landing is safer for the drone than it keep performing flight maneuvers towards home without monitoring the collision sensor. Therefore it reduces the loss of drones, objective (ii) of the application.

#### 7.2.1.1.4 MonitorEnvironment

To carry out an efficient monitoring environment, the smoke sensor is used together with a visual monitoring (camera) so that the pilot can view possible fires during the drone's mission. When the drone detects a region with fire through the smoke sensor, it takes some pictures of the environment for a further analysis of the firefighters and provides visual hints to assist in the location for any local combat operation. However, when a malfunction occurs with the camera, the drone must fly around the fire source position. These steps make it possible to restrict the area using GPS data.

In this scenario, it was assumed that the fire detection is carried out from the source to the destination. Thus, Figure 21 illustrates the movements necessary to restrict the fire area used in the adaptation. The triangle represents the region where the smoke sensor detected the fire, while the circle indicates the end of the adaptive behavior. Besides, each numbered arrow indicates the direction and order of the required movements.

Figure 21 – Necessary adaptation movements



Source – Prepared by the author

Figure 22 shows the *MonitorEnvironment* exceptional scenario's implementation using the DRES-ML syntax. The condition is implemented, in the Given clause, using the status of the camera actuator and the result of smoker detector sensor expressions, both associated by the operator "and". The first one is compared to the "DETECTED" value using an equality operation ("equal to"), and the last one is compared to the "STARTED" value using an unequal operator ("not equal to").

The join point 3 (see Figure 13) is the interception of the *Flying* scenario, however, *MonitorEnvironment* determines the drone's maneuver event for a specific direction "EAST". In addition, this interception is realized before the *Flying* scenario, thus, the adaptation strategy "Before" is used in the Then clause and the adaptation script called the "framework" is indicated. The necessary adaptation are implemented using the drone maneuver commands passing the necessary directions (such as shown in Figure 21). This adaptation allows monitoring to be carried out even in such an emerging situation, therefore it collaborates with the objective (i) of the application.

It is worth mentioning that it was assumed that the drone is flying from west

Figure 22 – MonitorEnvironment exceptional scenario modeling with DRES-ML

```

Exceptional Scenario MonitorEnvironment
Given: ( ( SmokerDetector.result == DETECTED ) and ( Camera.status ≠ STARTED ) )
When: Drone.maneuvers EAST
Then: execute Before framework
      framework :
        UAV.direction NORTH
        UAV.direction WEST
        UAV.direction SOUTH
        UAV.direction SOUTH
        UAV.direction EAST
        UAV.direction EAST
        UAV.direction NORTH

```

Source – Prepared by the author

to east and that these commands are appropriate for that context situation being able to perform the expected adaptation (demarcate the area containing the fire).

#### 7.2.1.1.5 EmergencyCamera

Although the camera is used to support monitoring the flight region, it can also be used as an instrument to assist in locating a lost drone. Based on this premise, the *EmergencyCamera* exceptional scenario was created.

When the drone needs to realize a safe landing and has a problem with the GPS (as shown in the condition in Figure 17), localizing, it may be challenging. Therefore, photos or videos that the camera can provide indications to assist in locating the drone. This scenario applies the emergency mode that reduces the drone's battery consumption, turns on the camera directed either to the origin or to the destination locations through the gimbal, depending on which one is closer.

Figure 31 shows the implementation of this scenario using the DRES-ML. The *When* clause contains a conditional expression that handles GPS status, comparing it to the error status ("ERROR") using the comparison operator ("equal to").

As can be seen in Figure 17, that the join point 2 intercepts after the execution of the SafeLanding exceptional scenario. Therefore, it is modeled using the *safe landing event starts* inside the *When* clause, and "After" is specified in the *Then* clause, representing the type of adaptation. Besides, in the last clause, it is indicated which script represents the adaptive behavior by name, in that situation, called "helperCamera".

Figure 23 – EmergencyCamera exceptional scenario modeling with DRES-ML

```

Exceptional Scenario EmergencyCamera
Given: ( GPS.status == ERROR )
When: SafeLanding STARTS
Then: execute After helperCamera
      helperCamera :
        START Camera
        START MANUAL gimbal
        START EnergySavingMode
        if ( Drone.distance from ORIGIN region < DESTINATION region )
          YAM Gimbal.rotation to 180°
        else
          YAM Gimbal.rotation to 0°

```

Source – Prepared by the author

Finally, it was modelled the script in terms of command and flow statements. Thus, they are executed to start the camera, start the manual gimbal and start the energy saving mode. To allow the camera to point to the nearest region between origin and destination, it is used the *if-else* statement with the condition being modeled using the *comparative relative distance* expression. It checks if the drone's distance from the origin is less than the destination's distance. If the drone distance to the origin is smaller, the drone should point the camera to the west using gimbal rotation event. In this case, the drone rotates the camera to 180 degrees on the yam axis. Otherwise, the drone rotates to 0 degrees also on the yam axis. It is worth mentioning that it was assumed that the drone always initiates its mission in the west and has the destination in the east, as stated in the problem description. This adaptation tries to maximize the change of the drone to be tracked in a situation where the drone is lost, therefore it assists in the objective (ii) of the system.

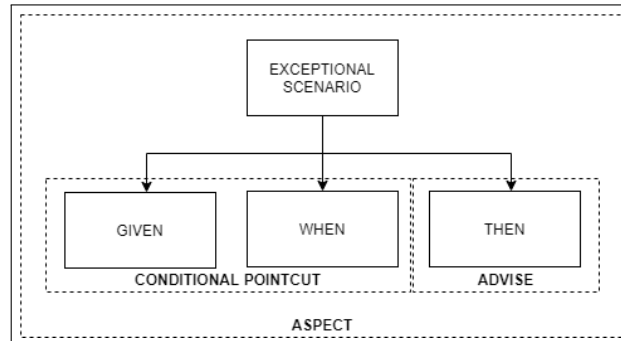
#### 7.2.1.2 Wrapper Generator

In this section, it is presented an implementation of a Model-to-Text generator to the Dragonfly drone simulator (MAIA et al., 2019b) from DRES-ML. Due to this, the output artifact is a wrapper code implemented using AOP.

Figure 24 shows the mapping between the BDD concepts used in the DRES-ML and the AOP technique for creating wrappers for the Dragonfly tool. The state of the context (Given), along with the occurred event (When), represents the *conditional*

*pointcut*, the adaptive behavior (Then) represents a *advice*.

Figure 24 – Correlation between DRES-ML and AOP structures.



Source – Prepared by the author

Once defined how the DRES-ML concepts can be translated into the concepts of the wrapper, it was possible to implement a script to perform an automated translation that generates executable wrapper Java code for the Dragonfly tool. That process was realized using the TextGen language that is integrated with the MPS JetBrains. That integration allows that TextGen traverses each modeled node from DRES-ML AST and transforms it into output text values implemented through templates.

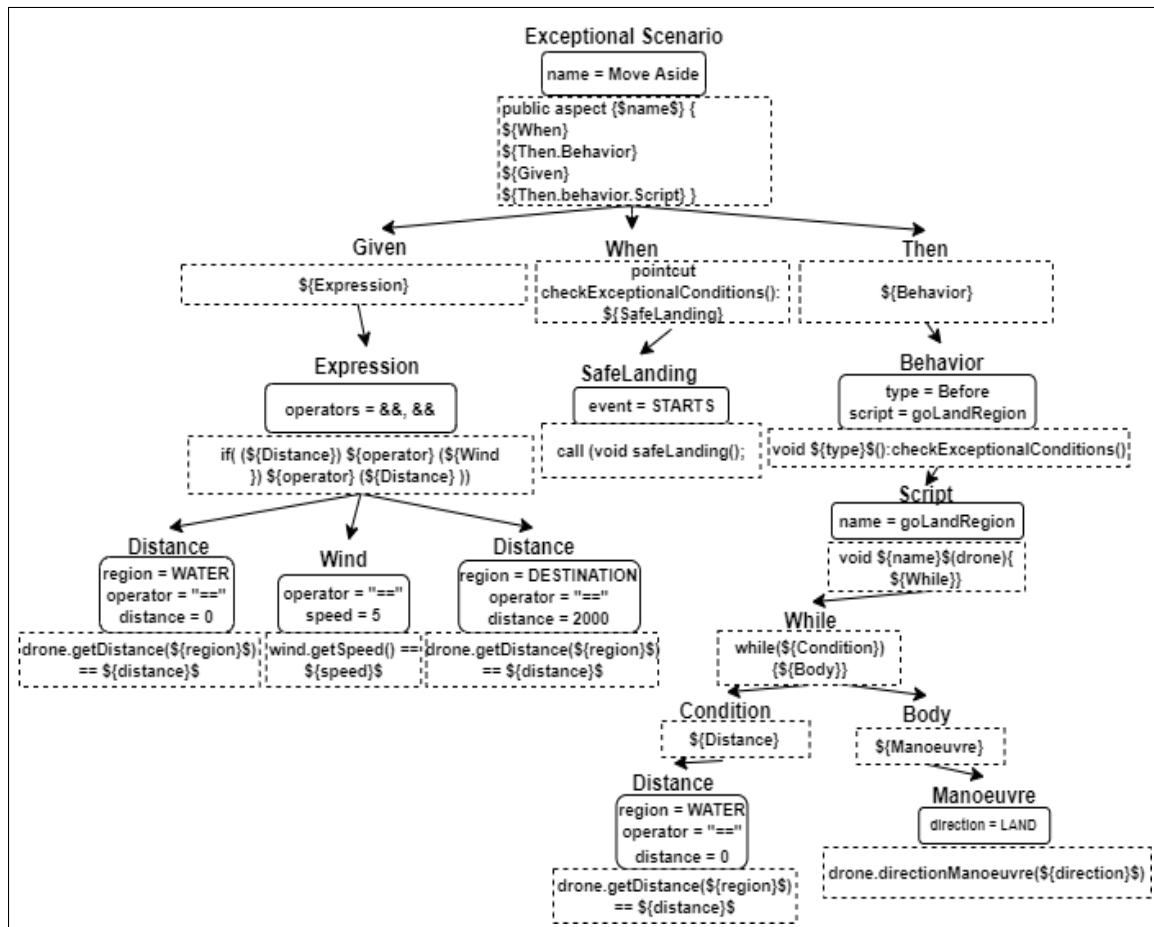
The *MoveAside* exceptional scenario implemented in DRES-ML is used to exemplify how TextGen performs to generate wrapper code. Figure 25 exhibits the modeled AST and the generator script for that example. The nodes, the child nodes, properties of AST are shown through regular boxes, while the scripts to produce the wrapper code using the Textgen language based on each in node are shown through dashed boxes.

The M2T process, through the implemented script, obtains the necessary information from the AST to create the target code using the Textgen language. Therefore, it provides in its syntax instructions for getting information from each node, such as *\$node.child* and *\$attributed node\$*, to obtain information from a child node and from attributes from current node, respectively. The values that do not follow this syntax are constants, such as "public aspect" in the Exceptional Script node.

The generator performs the transformation process using the defined template by the TextGen script by starting the execution from the root (Exceptional Scenario) and going through the tree according to the script specifications. That template constructs the header and delimits the body of the aspect object. The name and body



Figure 25 – AST and TextGen scripts for Move Aside exceptional scenario



Source – Prepared by the author

content of the aspect class are requested from attribute name, and Given, When and Then child nodes, respectively.

The sub tree of the AST that specifies the Given clause represents actual context of drone and environment forming the condition of advice. This is designed with conjunctive associations (logical expressions related by logical operators && ) of expressions of drone resources in relation to distance of regions and wind speed. These expressions contain templates that use the Dragonfly API and values specified in the attributes of each resource node to define the conditions understandable by the tool.

The *Safe Landing* event is the join point to the adaptive behavior of *Move Aside*, thus the *When* node contains the template that declares the pointcut and it child specifies the signature of the method from tool's API that represents this event.

The Then node contains the device type specification (before, after or around) and specifies instructions for the drone to perform the expected adaptation. This new

behavior was translated using *while* statement (repetition structure) to execute command for the drone repetitively, while it is not flying over a land region. The repetition stop condition was built using a method defined in the tool's API that returns the drone's distance to a region passed by parameter as a String. That distance is obtained through value defined in current node from the region attribute (WATER). Furthermore, for the operator (==) and the value for comparison (0) are obtained from the same node through the values specified for the operator and the distance attributes. In addition, the body of statement contains maneuver commands to land. It is translated by an API method of the tool causing the drone to maneuver to a position that approaches the passed region (LAND).

Figure 26 – Generated move aside wrapper.

```
public aspect MoveAside{
    pointcut checkExceptionalConditions(): call (void safeLanding());
    void before (): checkExceptionalConditions(){
        if(drone.getDistance("WATER") == 0 &&
           environment.getWindSpeed() < 5 &&
           drone.getDistance("DESTINATION") >= 2000)
            goLandRegion();
    }
    void goLandRegion(Drone drone){
        while(drone.getDistance("WATER") == 0){
            drone.directionManeuver("LAND");
        }
    }
}
```

Source – Prepared by the author

After the M2T process has been carried out, an artifact is produced containing the *MoveAside* exceptional scenario using the wrapper technique and using API commands of the target tool (Figure 26). Thus, the artifact can be compiled and executed together with the Dragonfly simulator. The simulation of the drone running on the tool both the normal and the adaptive behavior of *MoveAside* was recorded<sup>2</sup>.

An complete generator was implemented in order to translate exceptional scenarios previously presented in Section 7.2.1.1. It was created to transform the exceptional scenario modeled with DRES-ML to a wrapper code that can be executed in the DragonFly simulator. The transforming code, also using the Texgen language,

<sup>2</sup> <https://cutt.ly/shDDP3Z>

can be found in the code repository<sup>3</sup>.

Figure 27 represents the corresponding wrapper for the *KeepFlying* exceptional scenario. The conditional pointcut is defined by the *When* and *Given* clauses. The former represents the call of the safe landing method of the tool's API (join point), while the latter implements the conditional expression of pointcut inside the *if*'s guard. They check whether the wind direction is to the east and its speed is greater or equal than 5 m/s, and the distance of the drone from destination is smaller than 2000 meters (meaning 2 cells of environment panel of the Dragonfly). These conditions are implemented by the accessible methods *environment.getWindDirection* - returning a wind direction string value, *environment.getWindSpeed* - returning a speed wind integer value and *drone.getDistance* - returning an integer value of the drone's distance to a specified region, from environment and drone entities of the simulator.

The *Then* clause specifies the advice that implements the adaptive behaviors. The statements needed for adaptation are grouped within a method that is called within the body of the advice. The adaptation is implemented using the *While* structure and the *drone.directionManeuver* method, while receives parameters directing the drone to the destination location. Besides, that clause specifies the advice type that is implemented at the beginning of the advice signature. The regular and adaptive behaviors were recorded in the simulator<sup>4</sup> to make it clear how they work.

Figure 27 – Generated Keep flying wrapper

```
public aspect KeepFlying{
    pointcut checkExceptionalConditions(): call (void safeLanding());
    void around(): checkExceptionalConditions(){
        if(enviroment.getWindDirection() == "EAST" &&
            enviroment.getWindSpeed() >= 5 &&
            drone.getDistance("DESTINATION") < 2000)
            goDestination(drone);
    }
    void goDestination(Drone drone){
        while(drone.getDistance("DESTINATION") != 0){
            drone.directionManeuver("DESTINATION");
        }
    }
}
```

Source – Prepared by the author

Figure 28 shows the generated wrapper of *SafeRTH* exceptional scenario. It

<sup>3</sup> <https://github.com/lucasvieira123/DSL-Dragonfly/tree/DSLv.5>

<sup>4</sup> <https://cutt.ly/EjYltxh>

uses an around advice in the join point that performs drone maneuvers for any direction (*drone.directionManeuver(\*)*). The conditions of the pointcut are implemented using a methods of the drone's API that get the state of the *return to home* operation and the state of the collision sensor. They verify whether it is performing the operation to return to the home point (*drone.isReturningToHome()*) and check if the collision sensor is failing (*drone.getCollisionSensorState() == "FAILURE"*). The adaptation represented by the advice implements a forced landing through the *safelanding()* method. The execution of this wrapper was also recorded<sup>5</sup>.

Figure 28 – Generated SafeRTH Wrapper

```
public aspect SafeRTH{
    pointcut checkExceptionalConditions(): call (void directionManeuver(*));
    void around(): checkExceptionalConditions(){
        if(drone.isReturningToHome() &&
           collisionSensor.getState() == "FAILURE")
            newSafeLand(drone);
    }
    void newSafeLand(Drone drone){
        drone.safeLand();
    }
}
```

Source – Prepared by the author

Figure 29 shows the generated wrapper from *SwitchToManual* exceptional scenario. The *RTH STARTS* event in DRES-ML is represented by *ReturnToHome* method call, building the pointcut implementation. To validate whether the GPS resource is in an error status, it is implemented using a GPS API method (*drone.getGPSSState()*) that is compared to "FAILURE". In addition, methods of the drone API are used to check the distance to a defined region (*drone.getDistance()*) and to check whether drone is running an automatic flight, (it is used the *isAutomatic()* method). The adaptive behavior is facilitated by the method already implemented in the drone API, *setIsAutomatic(false)*, forcing the manual pilot control. The execution of this wrapper can be viewed for a better understanding<sup>6</sup>.

Figure 30 illustrates the wrapper created from the exceptional scenario *MonitorEnvironment*. The pointcut is implemented using the *directionManeuvre()* method included in the drone API. In order to check if the status of the camera actuator is

<sup>5</sup> <https://cutt.ly/mjYPuXR>

<sup>6</sup> <https://cutt.ly/kjU2eUr>

Figure 29 – Generated SwitchToManual Wrapper

```

public aspect SwitchToManual{
    pointcut checkExceptionalConditions(): call (void returnToHome());
    void around(): checkExceptionalConditions(){
        if(drone.getGPSState() == "FAILURE" &&
           drone.getDistance("DESTINATION") < drone.getDistance("ORIGIN") &&
           drone.isAutomatic())
            turnToManual(drone);
    }
    void turnToManual(Drone drone){
        drone.setAutomatic(false);
    }
}

```

Source – Prepared by the author

not started, it is used the *getCameraState()* method of the drone entity and compared whether its value is different from "ON". In addition, to perform the framework flight looking for fire, call the manoeuvre method (*drone.directionManoeuvre()*) were used indicating in the parameter the direction obtained from the drone resource attribute. The execution of this wrapper was also recorded to facilitate the understanding<sup>7</sup>.

Figure 30 – Generated MonitorEnvironment Wrapper

```

public aspect MonitorEnvironment{
    pointcut checkExceptionalConditions(): call (void directionManoeuvre(*));
    void before(): checkExceptionalConditions(){
        if(drone.getCameraState() != "ON" &&
           drone.getSmokerSensorResult() == "DETECTED")
            executeFramework(drone);
    }
    void executeFramework(Drone drone){
        drone.directionManoeuvre("NORTH");
        drone.directionManoeuvre("WEST");
        drone.directionManoeuvre("SOUTH");
        drone.directionManoeuvre("SOUTH");
        drone.directionManoeuvre("EAST");
        drone.directionManoeuvre("EAST");
        drone.directionManoeuvre("NORTH");
        drone.directionManoeuvre("NORTH");
    }
}

```

Source – Prepared by the author

The *EmergencyCamera* exceptional scenario has the generated wrapper represented in Figure 31. As already known, the *SafeLanding starts* event is imple-

<sup>7</sup> <https://cutt.ly/bjU87xk>

Figure 31 – Generated EmergencyCamera Wrapper

```

public aspect EmergencyCamera{
    pointcut checkExceptionalConditions(): call (void safeLanding());
    void after(): checkExceptionalConditions(){
        if(drone.getCameraState() == "FAILURE")
            helperCamera(drone);
    }
    void helperCamera(Drone drone){
        drone.setCameraState("ON");
        drone.setGambialState("ON");
        drone.setEconomyMode(true);
        if (drone.getDistance("ORIGIN") < drone.getDistance("DESTINATION")){
            drone.setGambialDirection("WEST");
        }else {
            drone.setGambialDirection("EAST");
        }
    }
}

```

Source – Prepared by the author

mented in a pointcut of the *safeLanding()* join point and the verification of the GPS status inside the Given clause is mapped to the condition of the advice implemented through the *drone.getCameraState()* method. To implement the commands to start the camera and the automatic gimbal, it is necessary to use the modifiers *setCameraState()* and *setGimbalState()* passing as parameter "ON" for each resource and, to start the drone's Energy Saving mode, it is necessary to use the *setEconomyMode(true)* method. The *setGimbalDirection()* method of the drone entity is capable of indicating the desired rotation in yam axis for which the degree of rotation is implemented as directions. It was also recorded the execution of this wrapper in the Dragonfly simulator<sup>8</sup>.

<sup>8</sup> <https://cutt.ly/8jU89Dv>

## 8 CONCLUSION AND FUTURE WORKS

The applicability of the drone in different tasks and environments has soared with the advance of drone technologies. In addition, the increasing level of automation reduced the need for a pilot intervention. However, there are lots of uncertainties that cannot not initially be predicted at design time and that generates exceptional situations during the drone use. Thus, drones could expand their degree of autonomy through self-adaptive capabilities to deal with those exceptional situations.

Based on that, this work proposed an approach to model exceptional situations and self-adaptive behaviors for drone-based applications. Initially, the domain expert diagnoses the possible exceptional scenarios and the corresponding behaviour adaptive strategies. After that, the domain expert implements the exceptional scenarios and the self-adaptive behaviours using the high level abstractions provided by the domain-specific language DRES-ML. Finally, the user can export the modelled scenarios for a target platform using engines that map the domain language to the language used on the target platform.

The DRES-ML is based on techniques such as behavior-driven development (BDD) and aspect-oriented programming (AOP). A domain analysis was carried out to survey the main concepts related to the domain to increase the expressiveness of the language. The DRES-ML was based on the JetBrains Meta Programming System (MPS), a well known language workbeanches example. That gives support for language implementation, definition of a modeling environment and realization of the M2T process.

To check the applicability of the proposed DSL a proof of concept was carried out using an application for monitoring forest fires using a drone. There were modelled five exceptional scenarios. And, an engine to transform the scenario specifications into the wrapper code used in the Dragonfly simulator. Thus, it was possible to verify the applicability of the proposed approach by modeling exceptional scenarios and adaptive behaviors using the domain-specific language, and transforming them in a artifact that is executable in a drone simulator.

## 8.1 ACHIEVEMENTS

This research produced two papers, one published in the 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self Managing Systems (SEAMS) (MAIA et al., 2019b) and another one in the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (MAIA et al., 2019b). Also, it developed the Dragonfly simulator<sup>1</sup>.

Besides, other publications were produced not directly related to this work; However, they improved the researcher's foundation, such as (DAMASCENO et al., 2019); (SOUZA et al., 2019) (ALVES et al., 2020); (COSTA et al., 2020), (SOUZA et al., 2020) and (GADELHA et al., 2020).

## 8.2 LIMITATIONS

It is possible to cite as current limitations of this work:

- The validation was performed only with a single target platform, thus it is necessary to try to implement the M2T process for other drone platforms;
- Validation of the usability of the language was not carried out, since only the author of this work used the language. Therefore, empirical experiments with other users are necessary to evaluate the language usability;
- No other technique, besides aspect-oriented programming, has been tested to include adaptive behavior at runtime. It is necessary to investigate other application strategies, such as dependency injection, APIs, design patterns, among others, to analyses how the DSL can be used with other drone platforms;
- Only one example application has been used as a proof of concept. Then, for a deeper analysis of the DSL, more examples of drone applications should be modelled.

## 8.3 FUTURE WORK

For future work, it is expected to conduct a user experiment to the analyze usability DRES-ML, checking his/her ability to understand, validate, modify, and even develop solutions through the approach. In addition, some properties will also be

<sup>1</sup> <https://github.com/DragonflyDrone/Dragonfly>



analyzed, such as *Productivity* - the effectiveness of using the proposed solution; *Coverage* - the ability of language to express all or a subset of the specific domain; *Completeness* - refers to the degree to which a language can express programs that contain all the information needed to execute them, and so on. Finally, the applicability on other target platforms should also be analyzed with other techniques for including adaptive behaviors at runtime.

## BIBLIOGRAPHY

- AHMAD, M. First step towards a domain specific language for self-adaptive systems. In: IEEE. **2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)**. [S.l.], 2010. p. 285–290.
- ALVARES, F.; RUTTEN, E.; SEINTURIER, L. Behavioural model-based control for autonomic software components. In: IEEE. **2015 IEEE International Conference on Autonomic Computing**. [S.l.], 2015. p. 187–196.
- ALVES, L. V.; MELO, R. T. de; COSTA, L. F. da; ROCHA, C. L.; ERIKO, W. d. O.; CAMPOS, G. A. de; SOUZA, J. T. de. An agent program in an iot system to recommend plans of activities to minimize childhood obesity. In: IEEE. **2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)**. [S.l.], 2020. p. 674–683.
- ARCAINI, P.; MIRANDOLA, R.; RICCOBENE, E.; SCANDURRA, P. A pattern-oriented design framework for self-adaptive software systems. In: IEEE. **2019 IEEE International Conference on Software Architecture Companion (ICSA-C)**. [S.l.], 2019. p. 166–169.
- BARESI, L.; GUINEA, S.; TAMBURRELLI, G. Towards decentralized self-adaptive component-based systems. In: **Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems**. [S.l.: s.n.], 2008. p. 57–64.
- BOZHINOSKI, D.; RUSCIO, D. D.; MALAVOLTA, I.; PELLICCIONE, P.; TIVOLI, M. Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters. In: IEEE. **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2015. p. 801–806.
- BRABERMAN, V.; D'IPPOLITO, N.; KRAMER, J.; SYKES, D.; UCHITEL, S. Morph: A reference architecture for configuration and behaviour self-adaptation. In: **Proceedings of the 1st International Workshop on Control Theory for Software Engineering**. [S.l.: s.n.], 2015. p. 9–16.
- BRAMBILLA, M.; CABOT, J.; WIMMER, M. Model-driven software engineering in practice. **Synthesis lectures on software engineering**, Morgan & Claypool Publishers, v. 3, n. 1, p. 1–207, 2017.
- BROY, M.; KIRSTAN, S.; KRCMAR, H.; SCHÄTZ, B. What is the benefit of a model-based design of embedded software systems in the car industry? In: **Emerging Technologies for the Evolution and Maintenance of Software Models**. [S.l.]: IGI Global, 2012. p. 343–369.
- CALINESCU, R.; GERASIMOU, S.; JOHNSON, K.; PATERSON, C. Using runtime quantitative verification to provide assurance evidence for self-adaptive software. In: **Software Engineering for Self-Adaptive Systems III. Assurances**. [S.l.]: Springer, 2017. p. 223–248.
- CAMPAGNE, F. **The MPS language workbench: volume I**. [S.l.]: Fabien Campagne, 2014. v. 1.

CECIL, J. A conceptual framework for supporting uav based cyber physical weather monitoring activities. In: IEEE. **2018 Annual IEEE International Systems Conference (SysCon)**. [S.l.], 2018. p. 1–8.

CHHETRI, M. B.; LUONG, H.; UZUNOV, A. V.; VO, Q. B.; KOWALCZYK, R.; NEPAL, S.; RAJAPAKSE, I. Adsl: An embedded domain-specific language for constraint-based distributed self-management. In: IEEE. **2018 25th Australasian Software Engineering Conference (ASWEC)**. [S.l.], 2018. p. 101–110.

CONCEPT of Operations for Drones: A risk based approach to regulation of unmanned aircraft. 2015. Disponível em: <<https://www.easa.europa.eu/document-library/general-publications/concept-operations-drones>>.

COSTA, L. F. da; MELO, R. T. de; ALVES, L. V.; ROCHA, C. L.; ARAUJO, E. W. de O.; CAMPOS, G. A. L. de; SOUZA, J. T. de; TRIANTAFYLIDIS, A.; ALEXIADIS, A.; VOTIS, K. et al. Smart algorithm for unhealthy behavior detection in health parameters. In: IEEE. **2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)**. [S.l.], 2020. p. 654–663.

COSTIOU, S.; KERBOEUF, M.; CAVARLÉ, G.; PLANTEC, A. Lub: a dsl for dynamic context oriented programming. In: **Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies**. [S.l.: s.n.], 2016. p. 1–9.

CULLEN, A.; WILLIAMS, B.; BERTINO, E.; ARUNKUMAR, S.; KARAFILI, E.; LUPU, E. Mission support for drones: A policy based approach. In: **Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications**. [S.l.: s.n.], 2017. p. 7–12.

DAMASCENO, A.; FERREIRA, A.; GAMA, E.; MORAES, J. P. R.; ALVES, L. V.; BARBOSA, M. H.; CHAGAS, M. L.; FREIRE, E. S. S.; CORTÉS, M. I. A landscape of the adoption of empirical evaluations in the brazilian symposium on human factors in computing systems. In: **Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems**. [S.l.: s.n.], 2019. p. 1–11.

ERDELJ, M.; NATALIZIO, E. Uav-assisted disaster management: Applications and open issues. In: IEEE. **2016 international conference on computing, networking and communications (ICNC)**. [S.l.], 2016. p. 1–5.

ERDWEG, S.; STORM, T. V. D.; VÖLTER, M.; TRATT, L.; BOSMAN, R.; COOK, W. R.; GERRITSEN, A.; HULSHOUT, A.; KELLY, S.; LOH, A. et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. **Computer Languages, Systems & Structures**, Elsevier, v. 44, p. 24–47, 2015.

FOWLER, M. **Inversion of control containers and the dependency injection pattern (2004)**. 2004.

FOWLER, M. Language workbenches: The killer-app for domain specific languages. 2005.

GADELHA, R.; VIEIRA, L.; MONTEIRO, D.; VIDAL, F.; MAIA, P. H. Scen@ rist: an approach for verifying self-adaptive systems using runtime scenarios. **Software Quality Journal**, Springer, p. 1–43, 2020.

GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education India, 1995.

Ganek, A. G.; Corbi, T. A. The dawning of the autonomic computing era. **IBM Systems Journal**, v. 42, n. 1, p. 5–18, 2003.

GHARIBI, M.; BOUTABA, R.; WASLANDER, S. L. Internet of drones. **IEEE Access**, IEEE, v. 4, p. 1148–1162, 2016.

GHOSH, D. **DSLs in action**. [S.l.]: Manning Publications Co., 2010.

GOMES, R.; STRAUB, J.; JONES, A.; MORGAN, J.; TIPPARACH, S.; SLETTEN, A.; KIM, K. W.; LOEGERING, D.; FEIKEMA, N.; DAYANANDA, K. et al. An interconnected network of uas as a system-of-systems. In: IEEE. **2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)**. [S.l.], 2017. p. 1–7.

GROHER, I.; VOELTER, M. Aspect-oriented model-driven software product line engineering. In: **Transactions on aspect-oriented software development VI**. [S.l.]: Springer, 2009. p. 111–152.

HAM, Y.; HAN, K. K.; LIN, J. J.; GOLPARVAR-FARD, M. Visual monitoring of civil infrastructure systems via camera-equipped unmanned aerial vehicles (uavs): a review of related works. **Visualization in Engineering**, SpringerOpen, v. 4, n. 1, p. 1, 2016.

HAREL, D.; THIAGARAJAN, P. Message sequence charts. In: **UML for Real**. [S.l.]: Springer, 2003. p. 77–105.

HASSANALIAN, M.; ABDELKEFI, A. Classifications, applications, and design challenges of drones: A review. **Progress in Aerospace Sciences**, Elsevier, v. 91, p. 99–131, 2017.

HOPPE, M.; BURGER, M.; SCHMIDT, A.; KOSCH, T. Dronos: a flexible open-source prototyping framework for interactive drone routines. In: **Proceedings of the 18th International Conference on Mobile and Ubiquitous Multimedia**. [S.l.: s.n.], 2019. p. 1–7.

JAHAN, S.; WALTER, C.; ALQAHTANI, S.; GAMBLE, R. Adaptive coordination to complete mission goals. In: IEEE. **2018 IEEE 3rd International Workshops on Foundations and Applications of Self\* Systems (FAS\* W)**. [S.l.], 2018. p. 214–221.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, IEEE, v. 36, n. 1, p. 41–50, 2003.

KICZALES, G.; COADY, Y. **Aspectc**. [S.l.]: Online publishing, URI <http://www.cs.ubc.ca/labs/spl/projects/aspectc.html>, 2001.

KICZALES, G.; HILSDALE, E. Aspect-oriented programming. sigsoft softw. eng. **Notes**, v. 26, n. 5, p. 313, 2001.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. Getting started with aspectj. **Communications of the ACM**, ACM New York, NY, USA, v. 44, n. 10, p. 59–65, 2001.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of aspectj. In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 2001. p. 327–354.

KOUNEV, S.; BROSIG, F.; HUBER, N. **The descartes modeling language**. [S.l.]: Universität Würzburg, 2014.

KŘÍKAVA, F. **Domain-specific modeling language for self-adaptive software system architectures**. Tese (Doutorado), 2013.

LUO, Y.; YU, Y.; JIN, Z.; LI, Y.; DING, Z.; ZHOU, Y.; LIU, Y. Privacy-aware uav flights through self-configuring motion planning. 2020.

MAIA, P.; VIEIRA, L.; CHAGAS, M.; YU, Y.; ZISMAN, A.; NUSEIBEH, B. Cautious adaptation of defiant components. 2019.

MAIA, P. H.; VIEIRA, L.; CHAGAS, M.; YU, Y.; ZISMAN, A.; NUSEIBEH, B. Dragonfly: a tool for simulating self-adaptive drone behaviours. In: IEEE. **2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.], 2019. p. 107–113.

MAIER, M. W. Architecting principles for systems-of-systems. **Systems Engineering: The Journal of the International Council on Systems Engineering**, Wiley Online Library, v. 1, n. 4, p. 267–284, 1998.

MOD, U. Joint doctrine note 2/11 the uk approach to unmanned aircraft systems. **UK MoD The Development, Concepts and Doctrine Centre, SWINDON, Wiltshire**, 2011.

MOREIRA, A. Aspect-oriented software development. 2005.

NIU, H.; GONZALEZ-PRELCIC, N.; HEATH, R. W. A uav-based traffic monitoring system-invited paper. In: IEEE. **2018 IEEE 87th Vehicular Technology Conference (VTC Spring)**. [S.l.], 2018. p. 1–5.

NORTH, D. et al. Introducing bdd. **Better Software**, v. 12, 2006.

ODERSKY, M.; ALTHERR, P.; CREMET, V.; EMIR, B.; MANETH, S.; MICHELOUD, S.; MIHAYLOV, N.; SCHINZ, M.; STENMAN, E.; ZENGER, M. **An overview of the Scala programming language**. [S.l.], 2004.

OREIZY, P.; GORLICK, M. M.; TAYLOR, R. N.; HEIMHIGNER, D.; JOHNSON, G.; MEDVIDOVIC, N.; QUILICI, A.; ROSENBLUM, D. S.; WOLF, A. L. An architecture-based approach to self-adaptive software. **IEEE Intelligent Systems and Their Applications**, IEEE, v. 14, n. 3, p. 54–62, 1999.

PERERA, T.; PRIYANKARA, A.; JAYASINGHE, G. Unmanned arial vehicles (uav) in smart agriculture: Trends, benefits and future perspectives. Uva Wellassa University of Sri Lanka, 2019.

RAHMES, M.; CHESTER, D.; HUNT, J.; CHIASSON, B. Optimizing cooperative cognitive search and rescue uavs. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything**. [S.l.], 2018. v. 10643, p. 106430T.

ROBERGE, V.; TARBOUCHI, M.; LABONTÉ, G. Fast genetic algorithm path planner for fixed-wing military uav using gpu. **IEEE Transactions on Aerospace and Electronic Systems**, IEEE, v. 54, n. 5, p. 2105–2117, 2018.

ROMANOVSKY, A.; ISHIKAWA, F. **Trustworthy cyber-physical systems engineering**. [S.l.]: CRC Press, 2016.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. **ACM Trans. Auton. Adapt. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. 2, maio 2009. ISSN 1556-4665. Disponível em: <<https://doi.org/10.1145/1516533.1516538>>.

SHETTY, S.; NEEMA, S.; BAPTY, T. Model based self adaptive behavior language for large scale real time embedded systems. In: IEEE. **Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004**. [S.l.], 2004. p. 478–483.

SILVA, J. P. S. da; ECAR, M.; PIMENTA, M. S.; GUEDES, G. T.; FRANZ, L. P.; MARCHEZAN, L. A systematic literature review of uml-based domain-specific modeling languages for self-adaptive systems. In: **Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems**. [S.l.: s.n.], 2018. p. 87–93.

SILVAGNI, M.; TONOLI, A.; ZENERINO, E.; CHIABERGE, M. Multipurpose uav for search and rescue operations in mountain avalanche events. **Geomatics, Natural Hazards and Risk**, Taylor & Francis, v. 8, n. 1, p. 18–33, 2017.

SOUZA, J. T. de; CAMPOS, G. A. L. de; COSTA, L. F. da; MELO, R. T.; WERBET, E.; ROCHA, C.; ALVES, L. V. An artificial agent to recommend activities to minimize childhood obesity problems in an iot system. In: SBC. **Anais da VII Escola Regional de Computação Aplicada à Saúde**. [S.l.], 2019. p. 139–144.

SOUZA, J. T. de; CAMPOS, G. A. L. de; ROCHA, C.; WERBET, E.; COSTA, L. F. d.; MELO, R. T. de; ALVES, L. V. An agent program in an iot system to recommend activities to minimize childhood obesity problems. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2020. p. 654–661.

SPINCZYK, O.; GAL, A.; SCHRÖDER-PREIKSCHAT, W. Aspect++ an aspect-oriented extension to the c++ programming language. In: **Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications**. [S.l.: s.n.], 2002. p. 53–60.

STEINBERG, D.; BUDINSKY, F.; MERKS, E.; PATERNOSTRO, M. **EMF: eclipse modeling framework**. [S.l.]: Pearson Education, 2008.

VOELTER, M.; BENZ, S.; DIETRICH, C.; ENGELMANN, B.; HELANDER, M.; KATS, L. C.; VISSER, E.; WACHSMUTH, G. **DSL engineering: Designing, implementing and using domain-specific languages**. [S.l.]: dslbook. org, 2013.

VOGEL, T.; GIESE, H. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In: IEEE. **2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.], 2012. p. 129–138.

WANG, H.; ZHAO, H.; ZHANG, J.; MA, D.; LI, J.; WEI, J. Survey on unmanned aerial vehicle networks: A cyber physical system perspective. **IEEE Communications Surveys & Tutorials**, IEEE, 2019.

WERMELINGER, M.; YU, Y. Analyzing the evolution of eclipse plugins. In: **Proceedings of the 2008 international working conference on Mining software repositories**. [S.l.: s.n.], 2008. p. 133–136.

XU, R.; YU, P.; ABRAMSON, M. J.; JOHNSTON, F. H.; SAMET, J. M.; BELL, M. L.; HAINES, A.; EBI, K. L.; LI, S.; GUO, Y. Wildfires, global climate change, and human health. **New England Journal of Medicine**, Mass Medical Soc, 2020.

YU, Y.; BARTHAUD, D.; PRICE, B. A.; BANDARA, A. K.; ZISMAN, A.; NUSEIBEH, B. Livebox: A self-adaptive forensic-ready service for drones. **IEEE Access**, IEEE, v. 7, p. 148401–148412, 2019.

ZHANG, R.; HOLVOET, T.; SONG, B.; PEI, Y. Uavs vs. pirates: An anticipatory swarm monitoring method using an adaptive pheromone map. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM New York, NY, USA, v. 14, n. 4, p. 1–31, 2020.

## APPENDIX



## APPENDIX A – Abstract Syntax Resources

Table 1 presents the abstract syntax of the resources, presenting the type of expression, the elements of the syntax and an example. For a visualization of this table in another perspective, can be found on this <<https://cutt.ly/OjfYo85>>.

**Table 1 – The abstract syntax of the resources**

Resource	Type	Abstract Syntax	Example instance
Drone-Maneuver-Direction	Conditional Expression	<EqualityOperatorEnum> <DirectionEnum>	<b>(Drone.Direction == NORTH)</b>
	Command Expression	<DirectionEnum>	<b>Drone.Direction NORTH</b>
	Event Expression	<DirectionEnum>	<b>Drone.Maneuvers NORTH</b>
Drone-Maneuver-Direction-ToRegion	Conditional Expression	<EqualityOperatorEnum> <RelativePositionEnum>	<b>(Drone.Direction ≠ DESTINATION region)</b>
	Command Expression	<RelativePositionEnum>	<b>Drone.Direction ORIGIN</b>
	Event Expression	<RelativePositionEnum>	<b>Drone.Maneuvers WATER</b>
DroneCurrent-Position	Conditional Expression	<EqualityOperatorEnum> <RelativePositionEnum> <Waypoint>	<b>(Drone.position == Latitude:-3.78; Longitude:-38.55; Altitude:5.3; )</b>
DroneRotate	Conditional Expression	<DroneAxesEnum> <GeneralOperatorEnum> Integer	<b>(PITCH Drone.rotation &gt;= 15°)</b>
	Command Expression	<DroneAxesEnum> Integer	<b>ROLL Drone.rotation to 45 °</b>
	Event Expression	<DroneAxesEnum> Integer	<b>YAM Drone.rotation turns to 90 °</b>
DroneSpeed	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(Drone.speed &lt;= 2.5 m/s)</b>
	Command Expression	Integer	<b>Drone.speed to 1.2 m/s</b>
	Event Expression	Integer	<b>Drone.speed goes to 3.0 m/s</b>

DroneSpeed Mode	Conditional Expression	<EqualityOperatorEnum> <ControlSwitchEnum>	<b>(Drone.speed <math>\neq</math> AUTOMATIC)</b>
	Command Expression	<ControlSwitchEnum>	<b>Drone.speed to MANUAL</b>
	Event Expression	<ControlSwitchEnum>	<b>Drone.speedMode goes to AUTOMATIC</b>
Drone Acceleration	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(Drone.acceleration &gt; 0.3 m/s<sup>2</sup>)</b>
	Command Expression	Integer	<b>Drone.acceleration to 0.1 m/s<sup>2</sup></b>
	Event Expression	Integer	<b>Drone.acceleration goes to 0.2 m/s<sup>2</sup></b>
Drone Acceleration Mode	Conditional Expression	<EqualityOperatorEnum> <ControlSwitchEnum>	<b>(Drone.acceleration == AUTOMATIC)</b>
	Command Expression	<ControlSwitchEnum>	<b>Drone.acceleration to AUTOMATIC</b>
	Event Expression	<ControlSwitchEnum>	<b>Drone.acceleration goes to MANUAL</b>
DroneAltitude	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(Drone.altitude &gt; 53 m)</b>
	Command Expression	Integer	<b>Drone.altitude to Integer m</b>
	Event Expression	Integer	<b>Drone.altitude goes to 80 m</b>
Motor	Conditional Expression	<EqualityOperatorEnum> <MotorStatusEnum>	<b>(Motor.Status == STOPPED )</b>
	Command Expression	<MotorActionEnum>	<b>START Motor</b>
	Event Expression	<MotorEventEnum>	<b>Motor STARTS</b>
	StatusEnum	"STARTED" "STOPPED" "ERROR"	
	Command	"START" "STOP"	

	Even- tEnum	"START" "STOP" "**"	
Mission WayPoint	Conditional Expression	<EqualityOperatorEnum> <1>*<n><WayPoints>	<b>(Mission.waypoints == Latitude:-3.78; Longitude:-38.55; Altitude:5.3; )</b>
	Command Expression	<1>*<n><WayPoints>	<b>Mission.waypoints Latitude:-3.78; Longitude:-38.55; Altitude:5.3;</b>
Mission	Conditional Expression	<EqualityOperatorEnum> <MissionStatusEnum>	<b>(Mission.Status ≠ PAUSED)</b>
	Command Expression	<MissionActionEnum>	<b>RESUME Mission</b>
	Event Expression	<MissionEventEnum>	<b>Mission UPLOADS</b>
	Sta- tusEnum	"READY_TO_START"  "READY_TO_UPLOAD"  "UPLOADING"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	
		"START"  "PAUSE"  "CANCEL"  "RESUME"  "RECOVER"	
	Even- tEnum	"UPLOADS"  "STARTS"  "PAUSES"  "CANCELS"  "RESUMES"  ***	
ReturnToHome HomePoint	Conditional Expression	<EqualityOperatorEnum> <Waypoint>	<b>(RTH.Status ≠Latitude:-3.78; Longitude:-38.55; Altitude:5.3; )</b>
	Command Expression	<Waypoint>	<b>RTH.homePoint Latitude:-3.78; Longitude:-38.55; Altitude:5.3;</b>

	Event Expression	<Waypoint>	<b>Mission.homePoint goes to Latitude:-3.78; Longitude:-38.55; Altitude:5.3;</b>
ReturnToHome HomePoint CurrentPosition	Command Expression	"CURRENT_POSITION"	<b>RTH.homePoint CURRENT POSITION</b>
	Conditional Expression	<EqualityOperatorEnum> <RTHStatusEnum>	<b>(RTH.Status == ERROR)</b>
ReturnToHome	Command Expression	<RTHActionEnum>	<b>RESUME RTH</b>
	Event Expression	<RTHEventEnum>	<b>RTH PAUSES</b>
	Sta- tusEnum	"READY_TO_START"  "READY_TO_UPLOAD"  "UPLOADING"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	
	Command	"START"  "PAUSE"  "CANCEL"  "RESUME"  "RECOVER"	
	Even- tEnum	"UPLOADS"  "STARTS"  "PAUSES"  "CANCELS"  "RESUMES"  ***	
EnergySaving ModeLow Warning	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(EnergySaving- Mode.lowBatteryWarning &lt;= 15 %)</b>
	Command Expression	Integer	<b>EnergySaving- Mode.lowBatteryWarning 15 %</b>
EnergySaving ModeVery LowWarning	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(EnergySaving- Mode.veryLowBattery Warning ≠ 15%)</b>

EnergySaving Mode	Command Expression	Integer	<b>EnergySaving- Mode.veryLow BatteryWarning</b> 15 %
	Conditional Expression	<EqualityOperatorEnum> <EnergySavingMode- StatusEnum>	<b>(EnergySaving- Mode.Status ≠ STOPPED)</b>
	Command Expression	<EnergySavingModeActio- nEnum>	START <b>EnergySavingMode</b>
	Event Expression	<EnergySavingModeEven- tEnum>	<b>EnergySavingMode PAUSES</b>
	Sta- tusEnum	"READY_TO_START"  "READY_TO_UPLOAD"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	
	Command	"START"  "PAUSE"  "STOP"  "RESUME"	
	Even- tEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
SafeLanding	Conditional Expression	<EqualityOperatorEnum> <SafeLandingStatusEnum>	<b>(SafeLanding.Status == RESUMED)</b>
	Command Expression	<SafeLandingActionEnum>	START <b>SafeLanding</b>
	Event Expression	<SafeLandingEventEnum>	<b>SafeLanding</b> STARTS
	Sta- tusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	

	Command	"START"  "PAUSE"  "STOP"  "RESUME"	
	EventEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
Landing	Conditional Expression	<EqualityOperatorEnum> <LandingStatusEnum>	<b>(Landing.Status ≠ ERROR)</b>
	Command Expression	<LandingActionEnum>	<b>PAUSE Landing</b>
	Event Expression	<LandingEventEnum>	<b>Landing CANCELS</b>
	StatusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	
	Command	"START"  "PAUSE"  "STOP"  "RESUME"	
	EventEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
TakeOff	Conditional Expression	<EqualityOperatorEnum> <TakeOffStatusEnum>	<b>(TakeOff.Status == READY_TO_START)</b>
	Command Expression	<TakeOffActionEnum>	<b>RESUME TakeOff</b>
	Event Expression	<TakeOffEventEnum>	<b>TakeOff PAUSES</b>

	Sta- tusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"	
	Command	"START"  "PAUSE"  "STOP"  "RESUME"	
	Even- tEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
FlightControl	Conditional Expression	<ControlSwitchEnum> <EqualityOperatorEnum> <FlightControlStatusEnum>	(AUTOMATIC <b>FlightControl.Status == STOPPED</b> )
	Command Expression	<ControlSwitchEnum> <FlightControlActionEnum>	MANUAL <b>FlightControl PAUSE</b>
	Event Expression	<ControlSwitchEnum> <FlightControlEventEnum>	MANUAL <b>FlightControl STARTS</b>
	Sta- tusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "STOPPED"	
	Command	"START"  "STOP"	
	Even- tEnum	"STARTS"  "STOPS"  ***	
GimbalRotation	Conditional Expression	<AxesEnum> <GeneralOperatorEnum> Integer	(ROLL <b>Gimbal.rotation</b> $\neq$ 10 °)
	Command Expression	<AxesEnum> Integer	ROLL <b>Gimbal.rotation to 30 °</b>
	Event Expression	<AxesEnum> Integer	YAM <b>Gimbal.rotation turns to 45 °</b>
Gimbal	Conditional Expression	<ControlSwitchEnum> <GeneralOperationEnum> <GimbalStatus>	(MANUAL <b>Gimbal.Status &lt;= CALIBRATING</b> )

	Command Expression	<GimbalAction> <ControlSwitchEnum>	CALIBRATE AUTOMAIC <b>Gimbal</b>
	Event Expression	<ControlSwitchEnum> <GimbalEvent>	AUTOMATIC <b>Gimbal</b> *
	StatusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "STOPPED"  "CALIBRATING"	
	Command	"START"  "STOP"  "CALIBRATE"	
	EventEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
	Conditional Expression	<EqualityOperatorEnum> <CameraStatusEnum>	<b>(Camera.Status == STOPPED)</b>
Camera	Command Expression	<CameraActionEnum>	PAUSE <b>Camera</b>
	Event Expression	<CameraEventEnum>	<b>Camera</b> RESUMES
	StatusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "PAUSED"  "STOPPED"  "RESUMED"	
	Command	"START"  "PAUSE"  "STOP"  "RESUME"	
	EventEnum	"STARTS"  "PAUSES"  "STOPS"  "RESUMES"  ***	
	Conditional Expression	<EqualityOperatorEnum> <Waypoint>	<b>(Camera.focusPoint ==</b> Latitude:-3.78; Longitude:-38.55; Altitude:5.3;)
Camera FocusPoint			



Payload	Command Expression	<Waypoint>	<b>Cam- era.focusPointLatitude:- 3.78; Longitude:-38.55; Altitude:5.3;</b>
	Event Expression	<Waypoint>	<b>Camera.focusPoint goes to Latitude:-3.78; Longitude:-38.55; Altitude:5.3;</b>
	Conditional Expression	<EqualityOperatorEnum> <PayloadStatusEnum>	<b>(Payload.Status ≠ CANCELED)</b>
	Command Expression	<PayloadActionEnum>	<b>UNLOAD Payload</b>
	Event Expression	<PayloadEventEnum>	<b>Payload LOADS</b>
	StatusEnum	"READY_TO_START"  "CANCELED"  "LOADED"  "UNLOADED"  "ERROR"	
	Command	"CANCEL"  "LOAD"  UNLOAD"	
	EventEnum	"CANCELS"  "LOADS"  UNLOADS"  ***	
	BatteryCapacity	Conditional Expression	<GeneralOperatorEnum> Integer
	BatteryVoltage	Conditional Expression	<GeneralOperatorEnum> Integer
Battery	BatteryCurrent	Conditional Expression	<GeneralOperatorEnum> Integer
	BatteryPercentage	Conditional Expression	<GeneralOperatorEnum> Integer
	Conditional Expression	<EqualityOperatorEnum> <BatteryStatus>	<b>(Battery.Status == ERROR)</b>
	StatusEnum	"NORMAL"  "ERROR"	

GPS	Conditional Expression	<EqualityOperatorEnum> <GPSStatusEnum>	<b>(GPS.Status ≠ DEACTIVATED)</b>
	Command Expression	<GPSSActionEnum>	<b>ACTIVATE GPS</b>
	Event Expression	<GPSEventEnum>	<b>GPS CALIBRATES</b>
	StatusEnum	"READY_TO_START"  "ERROR"  "ACTIVATED"  "DEACTIVATED"  "CALIBRATING"  "NEED_CALIBRATION"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  "CALIBRATES"  ""	
GPSResult	Conditional Expression	<EqualityOperatorEnum> <GPSResultEnum>	<b>(GPS.result == STRONG_SIGNAL)</b>
		"STRONG_SIGNAL"  "WEAK_SIGNAL"	
RelativeDistance	Conditional Expression	<RelativePositionEnum> <GeneralOperatorEnum> Integer	<b>(Drone.distance from WATER &gt;= 250 m)</b>
Comparative Relative Distance	Conditional Expression	<2>*<2> <Relative-PositionEnum > <GeneralOperatorEnum>	<b>(Drone.distance from LAND &lt;= than distance to DESTINATION)</b>
ScalarDistance	Conditional Expression	<Waypoint> <GeneralOperatorEnum> Integer	<b>(Drone.distance from Latitude:-3.78; Longitude:-38.55; Altitude:5.3; == 300 m)</b>
Obstacle Avoidance Sensor	Conditional Expression	<EqualityOperatorEnum> <ObstacleAvoidanceSensorStatusEnum>	<b>(ObstacleAvoidance.Status == AUTO LANDING)</b>
	Command Expression	<ObstacleAvoidanceSensorActionEnum>	<b>GO HOME ObstacleAvoidance</b>
	Event Expression	<ObstacleAvoidanceSensorEventEnum>	

	StatusEnum	"AUTO_LANDING"  "WAIT"  "GO_HOME"  "ERROR"  "ACTIVATED"  "DEACTIVATED"	
	Command	"AUTO_LANDING"  "WAIT"  "GO_HOME"  "DEACTIVATED"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Obstacle Avoidance SensorResult	Conditional Expression	<EqualityOperatorEnum> <Obstacle AvoidanceSensorResult> "DETECTED"  "NON-DETECTED"	(ObstacleAvoidance.result == NON-DETECTED)
IMU	Conditional Expression	<EqualityOperatorEnum> <IMUStatusEnum>	(IMU.Status == NEED_CALIBRATION)
	Command Expression	<IMUActionEnum>	ACTIVATE IMU
	Event Expression	<IMUEventEnum>	IMU DEACTIVATES
	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Accelerometer	Conditional Expression	<EqualityOperatorEnum> <AccelerometerStatusEnum>	(Accelerometer.Status == DEACTIVATED)
	Command Expression	<AccelerometerActionEnum>	CALIBRATE Accelerometer
	Event Expression	<AccelerometerEventEnum>	Accelerometer *

	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
	Conditional Expression	<EqualityOperatorEnum> <GyroscopeStatusEnum>	<b>(Gyroscope.Status ≠ ERROR)</b>
Gyroscope	Command Expression	<GyroscopeActionEnum>	<b>CALIBRATE Gyroscope</b>
	Event Expression	<GyroscopeEventEnum>	<b>Gyroscope DEACTIVATES</b>
	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
	Conditional Expression	<EqualityOperatorEnum> <CompassStatusEnum>	<b>(Compass.Status == DEACTIVATED)</b>
Compass	Command Expression	<CompassAction>	<b>CALIBRATE Compass</b>
	Event Expression	<ComapassEventEnum>	<b>Compass DEACTIVATES</b>
	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	Conditional Expression	<EqualityOperatorEnum> <CompassStatusEnum>	<b>(Compass.Status == DEACTIVATED)</b>

	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Magnetometer	Conditional Expression	<EqualityOperatorEnum> <MagnetometerStatusEnum>	<b>(Magnetometer == NEED_CALIBRATION)</b>
	Command Expression	<MagnetometerActionEnum>	ACTIVATE <b>Magnetometer</b>
	Event Expression	<MagnetometerEventEnum>	<b>Magnetometer *</b>
	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Barometer	Conditional Expression	<EqualityOperatorEnum> <BarometerStatusEnum>	<b>(Barometer.Status ≠ DEACTIVATED)</b>
	Command Expression	<BarometerActionEnum>	DEACTIVATE <b>Barometer</b>
	Event Expression	<BarometerEventEnum>	<b>Barometer ACTIVATES</b>
	StatusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Hygrometer	Conditional Expression	<EqualityOperatorEnum> <HygrometerStatusEnum>	<b>(Hygrometer.Status == ERROR)</b>
	Command Expression	<HygrometerActionEnum>	DEACTIVATE <b>Hygrometer</b>

	Event Expression	<HygrometerEventEnum>	<b>Hygrometer</b> DEACTIVATES
	Sta-tusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
	Even-tEnum	"ACTIVATES"  "DEACTIVATES"  ***	
HumidityLevel	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(Humidity.result &gt;= 5 %)</b>
Thermometer	Conditional Expression	<GeneralOperatorEnum> Integer	<b>(Thermometer.Status &gt; 45 °C)</b>
	Command Expression	<ThermometerActionEnum>	<ThermometerActionEnum> <b>Thermometer</b>
	Event Expression	<ThermometerEventEnum>	<b>Thermometer</b> <ThermometerEventEnum>
	Sta-tusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"	
	Even-tEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Tempera-tureLevel	Condi-tional	<GeneralOperatorEnum> Integer	<b>(Temperature.result &lt;GeneralOperatorEnum&gt;Integer °C)</b>
Smoker Detector	Conditional Expression	<EqualityOperatorEnum> <SmokerDetectorStatusEnum>	<b>(SmokerDetector.Status == ERROR)</b>
	Command Expression	<SmokerDetectorActionEnum>	CALIBRATE <b>SmokerDetector</b>
	Sta-tusEnum	"NEED_CALIBRATION"  "DEACTIVATED"  "ACTIVATED"  "ERROR"	

	Command	"ACTIVATE"  "DEACTIVATE"  "CALIBRATE"	
Smoker Detector Result	Conditional Expression	<EqualityOperatorEnum> <SmokerDetector- ResultEnum> "DETECTED"  "NON-DETECTED"	<b>(SmokerDetector.result == NON-DETECTED)</b>
Flight	Conditional Expression	<EqualityOperatorEnum> <FlightStatusEnum> "STOPPED"  "ON_GROUND"  "IN_FLIGHT"  "ERROR"	<b>(Flight.Status ≠ ON_GROUND)</b>
Lights	Conditional Expression	<EqualityOperatorEnum> <LightsStatusEnum>	<b>(Lights.Status == DEACTIVATED)</b>
	Command Expression	<LightsActionEnum>	<b>ACTIVATE Lights</b>
	Event Expression	<LightsEventEnum>	<b>Lights DEACTIVATES</b>
	Sta- tusEnum	"ACTIVATED"  "DEACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"	
	Even- tEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Landinggear	Conditional Expression	<EqualityOperatorEnum> <LandinggearStatusEnum>	<b>(LandingGear.Status == ERROR)</b>
	Command Expression	<LandinggearActionEnum>	<b>DEACTIVATE LandingGear</b>
	Event Expression	<LandinggearEventEnum>	<b>LandingGear ACTIVATES</b>
	Sta- tusEnum	"ACTIVATED"  "DEACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"	

	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
WindSpeed	Conditional Expression	<GeneralOperatorEnum> Integer	(Wind.speed => 8 m/s)
WindDirectionEnum	Conditional Expression	<EqualityOperatorEnum> <DirectionEnum>	(Wind.DirectionEnum == NORTH)
Anemometer	Conditional Expression	<EqualityOperatorEnum> <AnemometerStatusEnum>	(Anemometer.Status == ERROR)
	Command Expression	<AnemometerActionEnum>	DEACTIVATE Anemometer
	Event Expression	<AnemometerEventEnum>	Anemometer DEACTIVATES
	StatusEnum	"ACTIVATED"  "DEACTIVATED"  "ERROR"	
	Command	"ACTIVATE"  "DEACTIVATE"	
	EventEnum	"ACTIVATES"  "DEACTIVATES"  ***	
Emergency StopMode	Conditional Expression	<EqualityOperatorEnum> <EmergencyStopModeStatusEnum>	(EmergencyStopMode.Status ≠ READY_TO_START)
	Command Expression	<EmergencyStopModeActionEnum>	PAUSE EmergencyStopMode
	Event Expression	<EmergencyStopModeEventEnum>	EmergencyStopMode CANCELS
	StatusEnum	"READY_TO_START"  "ERROR"  "STARTED"  "PAUSED"  "CANCELED"  "RESUMED"  "READY_TO_START"	
	Command	"START"  "PAUSE"  "CANCEL"  "RESUME"	



	EventEnum	"STARTS"  "PAUSES"  "CANCELS"  "RESUMES"	
WayPoints	Concrete	<1>*<3>Integer	<b>Latitude:-3.78;</b> <b>Longitude:-38.55;</b> <b>Altitude:5.3;</b>
EqualityOperatorEnum	Enumeration	"=="    ≠	
InequalityOperatorEnum	Enumeration	">"    "<"    ">="    "<="	
GeneralOperatorEnum	Enumeration	"=="    ≠    ">"    "<"    ">="    "<="	
DirectionEnum	Enumeration	"NORTH"    "SOUTH"    "EAST"    "WEST"    "NORTH_EAST"   "SOUTH_EAST"    "NORTH_WEST"   "SOUTH_WEAST"	
AxesEnum	Enumeration	"PITCH"    "ROLL"    "YAM"	
Control-SwitchEnum	Enumeration	"AUTOMATIC"    "MANUAL"	
RelativePositionEnum	Enumeration	"OBSTACLE"    "DESTINATION"    "ORIGIN"    "WATER"    "LAND"	

Source - Prepared by the author