POLYTECHNIC OF BARI

Department of Electrical and Information Engineering

Master's degree in Computer Engineering

Master's degree thesis

Design and development of a framework for variability management in Self-Adaptive Systems through goal derivation and refinement

Supervisors	Graduating student
Prof. Marina Mongiello	Sacco Domenico
Dr. Francesco Nocera	

Academic year 2017/2018

INDEX

Abstract

In the last years, the increasing number of subjects in which software has been applied and the increasing complexity of software requirements lead to the search for design methods of versatile, reliable, flexible and adaptable software, which can stand changes due to the external environment. The Self-Adaptiveness is the capability of a system to modify its behaviour and/or its internal structure basing on the perception that the system has of the environment. A Self-Adaptive Software modifies its behaviour to satisfy the requirements. The Software Engineering community and the SEAMS (Symposia on Software Engineering for Adaptive and Self-Managing Systems) made efforts to formally define process models allowing to create Self-Adaptive Software. One of the most popular models in the architectural development of Self-Adaptive Systems is the MAPE-K architectural model, which has many common elements with the closed loop control systems' architecture.

The purpose of this work is to design and implement a framework to assist the creation of Self-Adaptive Systems. The realized framework allows to define the input variables, their type, the preconditions, the conditions and the goals achieved by the Self-Adaptive System to be implemented. The systems implemented through the framework can modify their behaviour in relation to the different users, whose information are stored in the knowledge base. The system allows also to change the controller's configuration at run-time, giving the possibility to reconfigure the controller for every new input.

The framework uses the MAPE-K model as architectural reference, implementing the Monitor, the Analyzer and the components of the knowledge base allowing their functionality. The OWL ontologies are used to define the preconditions and the conditions from the inputs received by the system. Two kinds of OWL individuals have been used: the first is variable and depending on the user and is used to define the preconditions, the second is generical and independent form the user and is used to define the conditions. The knowledge base was implemented as a relational database queried with SQL. The information defining the Self-Adaptive System to be implemented are entered through a web interface, which creates the base ontology and the system's knowledge base.

Different examples of Self-Adaptive Systems have been realized with the developed framework, these examples were tested and evaluated through the *Real Time Model Simulations*, making quantitative and quantitative analysis. The evaluation metrics used are the ones belonging to the control theory such as stability, accuracy, robustness and settling time. The behaviour of the system during the controller reconfiguration at run-time was studied as well. Statistical measures of the settling time have been presented, using the confidence intervals of the normalized Gaussian distribution. The systems generated using the framework are stable, accurate, robust and the controller can successfully reconfigure itself at run-time.

1. Introduction to Self-Adaptive Systems

1.1 Preview

In the last years, complexity of software systems rose noticeably. Consequently, the research for new methods of design, development and management of software systems intensified.

Features of increasing importance in such articulate and complex software systems as today's ones are versatility, reliability and flexibility. Besides the previous ones, another important feature increasingly required in the software systems is the capability to autonomously adapt to changes and solve exceptions without the human intervention (Self-Adaptiveness) [1][36][37].

1.2 Definition of Self-Adaptive System

It is defined as Self-Adaptive a system capable of modifying its behaviour and/or its structure in relation to the perception of the environment, the system itself and its requirements [1].

Self-Adaptive Systems can also be defined as systems capable of performing the right action basing on the knowledge of what is happening in the system, the goals to achieve and the requirements of the stakeholders [38].

The main autonomic features of a software are generally expressed with the Self-* notation [1][2][37], some of them are:

- **Self-monitoring,** one of the fundamental properties required in a Self-Adaptive system, is defined as the capability of a system to monitor its own state.
- **Self-configuration**, defined as a system's capability to choose the right control parameters for its own functioning.
- **Self-optimization**, capability of the system to choose the optimal configuration in a context.
- **Self-protection**, capability of the system to protect itself from external threats.

- **Self-healing**, capability of a system to repair its faults or to modify its internal configuration to allow service continuity even in case of exceptions and failures.

Self-Adaptiveness can be saw as a higher level of the properties before described, the highest degree of autonomy a system can achieve.

Self-Adaptiveness is an important feature in many software applications, some examples are: Self-Adaptive user interfaces, embedded systems, mobile and ad hoc networks, multi-agent systems, autonomous robotic units, reliable computation, peer-to-peer systems, sensor networks, Service Oriented Architectures and ubiquitous computation [2].

1.3 Self-Adaptive Systems classification

A valid criterion to classify Self-Adaptive Systems is to define the aspects that are functional for the autonomous adaptation operations [1].

1.3.1 Goals

Goals are divided between the ones related to the system's life cycle and the ones related to the scenarios in which the system is involved. A formal way to describe the system's goals is to divide the goals into sub-goals and then associate quantitative values to the attributes, each associated to a sub-goal.

- **Evolution:** it defines whether the system's goals are changing during its lifetime and how they might change. This feature might space between a system with invariable goals to a system in which goals are defined at run-time.
- **Flexibility:** it defines if the uncertainty level related to the goal, it can be rigid, constrained or not constrained.
- **Duration:** it depends on goal's validity during system's lifetime, it can be temporary or permanent. Permanent goals might reduce the degree of flexibility of a system.

- **Dependency:** indicates whether the systems goals are correlated between them or not. Correlations between goals increments system's complexity, especially if the correlated goals are conflictual.

1.3.2 Changes

At each context change, a Self-Adaptive System decides whether to adapt or not and how to perform the required adaptation. A good definition of context might be [3]: "any information accessible to the system that is able to modify its behaviour". Context is composed of the system itself, the environment and the actors. The environment can be defined as "anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation" [4]. Change might be caused by one of these three components or by a combination of them.

- **Source:** external or internal to the system, in the last case it can be in the infrastructure, in the middleware or in the application level.
- **Type:** Functional (a change in the system purpose), non-functional (for instance a change in the required level of performance or reliability) or technological (at hardware level, with new/replaced components or at software level, with component updates)
- **Frequency:** how often the considered change happens.
- **Anticipation:** it defines if it is possible to predict the change and if its relating issues might be solved before it takes place.

1.3.3 Mechanisms

Mechanisms define the system's reaction towards changes, particularly which behaviour is assumed by the system to face them.

- **Type:** adaptation can be related to the parameters, to the system structure or to a combination of those two types.

- **Autonomy:** the level of external intervention required during the adaptation process, it can vary from complete autonomy to the necessity of an external intervention (performed by a human or another system).
- **Organization:** adaptation operations can be performed by one or more system's components.
- **Scope:** indicates whether the adaptation is local or affects the whole system.
- **Duration:** how much time is required for the adaptation to perform
- **Timeliness:** indicates if the predefined time required for the adaptation is guaranteed or not.
- **Triggering**: if the change is determined by an event or by the time.

1.3.4 Effects

Indicates the effect that adaptation has on the system, in particular:

- **Criticality:** impact on the system if the adaptation procedure is not successful
- **Predictability:** defines if the Self-Adaptation consequences can be predicted or not.
- **Overhead:** shows how much the adaptation affects the overall system's performance.
- **Resilience:** defines the system's capability to continue functioning correctly during the adaptation operations.

Self-Adaptive systems can be also divided into open or closed. Open Self-Adaptive systems allows the introduction of new behavioural models and adaptation operations at run-time, while closed Self-Adaptive Systems do not support such functionality.

1.4 Technologies used in Self-Adaptive Systems

There are different approaches used in Self-Adaptive System's implementation. In this paragraph will be shown some technologies related to Self-Adaptive Systems and the advantages and disadvantages related to each of them.

1.4.1 Aspect Oriented Programming

Aspect Oriented Programming, introduced in [5], is another programming paradigm alongside procedural programming and Object-Oriented Programming. AOP was created to solve problems that cannot be handled by the other two programming paradigms.

Aspects are defined as decisions related to design, they differ from components because components are properties that can be clearly defined and incapsulated inside the system, like methods in procedural programming and objects in OOP. The aspects, in the contrary, are system properties that cannot be incapsulated with such clarity (memory access, number of instances of the same object that are transmitted inside a network, etc...).

AOP defines distinctively components, aspects and relations between them. Components and aspects are also defined in different languages.

Components can be defined as classes in an OOP language, or methods in the procedural programming. Aspects are with one or more dedicated languages, to express clearly each property defined in the requirements. Once all aspects and components have been implemented, aspects and components are connected to each other, the result is a single program.

Relationship with Self-Adaptive Systems

In the Self-Adaptive Systems design, two critical aspects are the exceptions management and the graceful degradation.

The management of unexpected situations is one of the key aspects in Self-Adaptive Systems and AOP can make such management easier, by reducing the design and modification time. Graceful degradation requirements can be implemented in an easier way using AOP as well.

In a Self-Adaptive System performing adaptation operations at run-time, the monitoring of the program components can cause a problematic overhead, using AOP, an optimized software can be produced, reducing the effects of such problem.

Dynamic Aspect Oriented Programming

The DAOP (Dynamic Aspect oriented Programming) [22] allows to extend the software components at run-time and is therefore applicable to the Self-Adaptive Systems that perform Self-Adaptation operations at run-time.

As explained in [23], the traditional AOP performs the “weaving “ operation, described as the process of composition of the program aspects, during compilation or loading phases.

A “weaving” during the compilation phase allows to create an optimized code and a solid control on the code, since it must be accepted by the compilator.

If the weaving operation is performed during the loading phase, the source code is not required, and the modifications are implemented through the classes instead.

Advantages and disadvantages

There are different cases in which usage of Aspect Oriented Programming can allow a substantial increase in performance and flexibility, for example [6], [7] and [8].

In the AOP can be difficult to individuate which are the aspects, how to define the connections between them and especially how to implement in the chosen language(s) all the aspects that are being considered.

In DAOP another problem is related to the overhead at execution time, which might cause a significative performance degradation in a system operating at real time.

1.4.2 Context Oriented Programming

As previously specified, a context is defined as the combination of environment, software system and actors interacting with it. Context Oriented Programming is related to the code level instead of the architectural one.

In Context Oriented Programming are expressed all the possible behavioural changes that a system can perform basing on the context in which the software is.

In [9] the concept of “layer” is defined, which represents a group of behaviours different between them and dependent on the context changes. The layers can be declared inside the classes of each module they’re affecting the behaviour (layer-in-class pattern) or independently (class-in-layer pattern).

In a multithread-based Self-Adaptive application, context can be the same for all threads or each thread can have its own context.

Another division in the COP (Context Oriented Programming) can be performed on the policy about the behavioural variations that are adopted. In particular, the behavioural change duration can be undefined or last until the context changes. The first solution can create systems in which the behaviour is difficult to predict in long term.

Using the COP does not exclude the possibility to use AOP as well, the last can be used to implement the Self-Monitoring feature in the designed system. COP can be used to implement the system’s adaptability, by creating a layer hierarchy, where each layer is used to describe a variable behaviour of the system.

Advantages and disadvantages

Context Oriented Programming brings advantages to systems in which all the different behaviours have the same importance to the application logic, in this case, describing all the behavioural variations of the application together is more intuitive.

Another advantage in a multithread environment is the possibility to define a different context for each thread, allowing a higher degree of flexibility.

An advantage of COP compared to the AOP is the possibility to remove the effects of a behavioural change as soon as its purpose has been served; on the contrary, in many systems using DAOP, the behavioural variation may affect the system for an undefined amount of time

One of the main issues with COP is defining the constraints between layers, defining which behavioural variations contained in the layers may or may not be adopted for each circumstance. It is necessary, for many systems, to define dependencies between behavioural variations, for instance a mutual exclusion between two different kinds of behaviour.

Other issues related to COP are triggering the behaviours described in the layers by events and to formally describe the layers and the behaviours contained into them, making the system able to understand the meaning of each behaviour. Formal description of layers can be important, allowing the system to decide which behaviour to adopt, basing the decision on the information the system has on it.

1.4.3 Fuzzy logic and rule-based systems

It is defined as fuzzy set a class which boundaries are not marked definitively [10]. For instance, a class **C** such as “x is big number” or “the temperature x is too hot”. Each element is associated to a value defining the degree of membership of the element to the class **C**, such value can be a real number in the range [0,1]. This value is given by a function called *membership function*, which expresses the degree of membership of an element in relation to a certain class.

Relationship with Self-Adaptive Systems

In Self-Adaptive Systems, fuzzy logic can be used to design more efficient controllers, which can be able to better manage the environment intervention over the system, keeping it in a stable configuration and increasing its performance.

An approach used in the AGC (Automatic Generation Control) is proposed in [11], where fuzzy logic is used to design an adaptive controller more performing than the standard PID controller. Using the membership functions, uncertainty over parameters is eliminated, allowing the definition of a more effective control function.

Fuzzy logic can be used to define the preconditions of the rules on which the behaviour of a Self-Adaptive System is based. An example is given in [12], in which an Action Repository is created, composed of triples (precondition, action and post-condition), each one representing a rule.

Those rules allow the system to make decisions, in the previous example, the decisions are executed with actions performed by a software installed on a mobile device.

Advantages and disadvantages

Fuzzy logic in Self-Adaptive systems can be more effective than traditional PID controllers, in some cases the last ones can be slower and their parameters more difficult to set dynamically.

Another advantage consists in the ease with which a Self-Adaptive Systems using fuzzy logic can perform a decision on the course of actions to take.

The main difficulty consists in defining the membership functions correctly. The membership functions are crucial for the functionality in Self-Adaptive Systems implementing fuzzy logic. In systems interacting with different users, the membership functions are likely to depend on each user profile.

1.4.4 Adaptive Featured Transition System and Self-Adaptive Automata

A Self-Adaptive Software System, as explained in [13], can be modelled as a set of programs, each of them with its own features, and a set of transitions from one program to another.

Labelled Transition Systems

Formal modelling of Self-Adaptive systems starts from the Labelled Transition Systems, defined by the tuple $\mathbf{<S,S_0,A,K,T,L>}$ where:

S is the set of system states, $\mathbf{S_0}$ the set of the initial states, **A** the action set, **K** the set of atomic prepositions (logical prepositions that can be true or false), $\mathbf{T \subseteq S \times S}$ representing the set of

transitions and the function **L: S → 2^k** having the purpose to associate each state with the prepositions in the set that are true for that state. Atomic prepositions are representing the difference between two programs (defined as set of *features*).

As defined by this model, the system changes from a certain configuration **S**, for which the property set **L(S)** is verified, to another state **S'**, for which **L(S')** property set is valid with the transition **T**. The transition **T** takes place performing the actions contained in **A.**

Although, LTS are not enough to formally describe a Self-Adaptive System exhaustively, it is also necessary to define at which conditions the transition from one configuration to another is allowed and how the features change following the configuration change.

Featured Transition Systems

An FTS is defined by the tuple **<S, S_0,A,K,T,L,f,g>**, that is adding to LTS the following parameters:

f is the set of the software features and **g: T x f x {true, false}** is a function associating to every transition **T** an expression, indicating if the transition from one state to another is allowed or not given a certain set of features. For example, a transition from one state to another can be forbidden in case the software or the environment don't have the features required for that transition.

An FTS can be reduced to an LTS associated to a particular "product", by keeping constant the set of features.

It is still necessary to make a distinction between the features related to the product and those related to the environment, [13] contributes to this aspect introducing the definition of Adaptive Featured Transition System, it follows a simplified description of such formal structure in the next paragraph.

Adaptive Featured Transition System

An A-FTS (Adaptive Featured Transition System) can be defined by the tuple **<S, S_0,A,K,T,L,f',g'>**, which is like an FTS but with these important variations:

f' now describes both environment and product features.

g': S x A x S -> (f' x f'x {true; false;}) is not a simple relation as **g: T x f x {true, false}** was, but a function representing how the system and environment configurations are changing for each existing transition.

Un FTS is an A-FTS where the features related to the environment are not variable.

The whole Self-Adaptive System activity will be modelled as an alternate sequence of macrostates (defined as current state, system configuration and environment condition) and actions (operations triggered by the change from one state to another).

Self-Adaptive Automata

Another formal model for Self-Adaptive Systems is suggested in [14] and is called Self-Adaptive Automata (SAA), such model divides the system in three parts:

- The **base system**, on which the adaptation is performed
- The **adaptation decision process**, which defines the behavioural changes the system will perform
- The **adaptation pattern**, which performs the adaptation operations issued by the adaptation decision process

A SAA can be described by the tuple **<S,S_0,T,T_0,H,A>** where:

- **S** is the set of states, **S_0** the initial state
- **H** is a set of labels
- **T: S x H -> S** Is a set of transitions functions, which partially maps the set **S**, **T_0** is defined as the initial transition function, starting from the initial state **S_0**.

- **A: S x T** represents the adaptation function, the one partially mapping the states with transition functions. The mapping is partial because not all the states are forced to change transition rules when the system changes its behaviour (there might be transition rules valid for all behavioural changes the system may face to perform the adaptations)

Transitions in SAA can be triggered by time or events

There are three important conditions in the SAA model's definition:

- Adaptation is performed instantaneously and in a single state, a limitation like the one in the A-FTS.
- Adaptation is performed by the system only when base system cannot perform any transition from the current state to the next state in its current configuration.
- The Self-Adaptive Automata is deterministic.

Advantages and disadvantages

The most evident advantage in modelling the Self-Adaptive Systems with an A-FTS is to use model checking theory to verify their properties.

SAA systems have as a basic feature the decomposition between the base system and the adaptation pattern, such distinction allows to change the adaptation pattern without affecting the base system and vice versa.

One of the most significant disadvantages in the formal models presented in this paragraph is the computational stress caused by the state explosion (the number of states increase exponentially with the number of the atomic propositions considered).

In A-FTS, the state explosion problem is even heavier due to the macrostates, more complex than the simple states, due to the transition functions, which are accounting also the conditions at which each transition takes place and the way it affects the system and the environment.

Another issue related to the formal description of Self-Adaptive Systems presented in this paragraph consists in the model limitations, in both SAA and A-FTS, which are based on instant behavioural modifications, such limit make the models simpler but also far from the behaviour assumed by Self-Adaptive Systems in a real context.

1.4.5 Control loops

A closed-loop control system is modelled in Self-Adaptive Systems through sensors, gathering the data coming from external sources, controllers, the units deciding the adaptations to perform and actuators, the units performing the adaptations operations acting on the system and on the environment.

A Self-Adaptive System can be modelled as a system consisting of one or more control loops, implementing the system adaptation by altering their transfer function [2].

In Self-Adaptive Systems a control ring can be interpreted as the sequence of *Monitoring*, *Analysing*, *Planning* ed *Executing (MAPE)* activities [15][16].

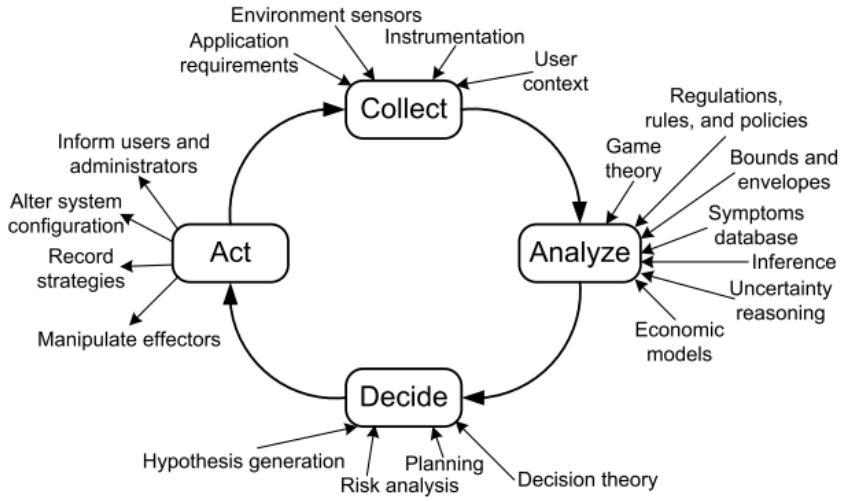

Figure 1.1: Control loop activities

This model is often too simple and is sometimes enriched to satisfy the design requirements, such as in [16], which introduces other two control loops. The first kind of ring is located within the MAPE system, the second makes the other control loops able to communicate between them.

Another method to include control loops in the design process of Self-Adaptive Systems is to extend the UML language with other elements, in order to make control loops explicit [17]. To face the lack of expressivity related to control loops in the traditional UML, new concepts are introduced.

At first, the concept of role: each component of the system can be a controller, a processor, a sensor or an actuator. Beside the role of each component, there are three kinds of interfaces a component can have: control interface (for controllers and processors), sensor interface or actuator interface.

The concept of strand is also defined: a strand links an interface to one or more interfaces and defines which of them are influenced by modifications performed on the starting interface. A strand can become a sequence linking two controllers, establishing that the activation of a controller, leads to the activation of the other.

Another concept introduced is the effect propagation, it indicates a modification on the performance in all the interfaces depending on a starting one if this is modified.

In this extended UML language is also introduced a function applied to each sensor's interface. This function has two possible values which are "control" or "environment", the first in case the controller can affect the values acquired by the sensor, the second in case this is not possible.

Advantages and disadvantages

The possibility to describe a system with one or more control loops allows to apply the elements of the control theory to the designed systems, allowing to formally demonstrate their stability and controllability.

Anyway, Self-Adaptive Systems of large sizes, which are modelled with a single control loop, suffer often of scalability issues. Beside the previous issue, is not possible in many cases to model a system with more control loops which are independent between them.

As explained in [17], other issues that might arise is to define the dependencies between interfaces correctly and to define which system components are affected by the control loop and which one are not.

Another problem related to the use of different controllers is their mutual affection, which can bring to a loop in which each controller activates the other, causing never-ending oscillations in the system.

1.4.6 Artificial Intelligence

The use of Artificial Intelligence is a solution for the implementation of many kinds of Self-Adaptive systems.

An example of artificial intelligence application in the implementation of Self-Adaptive Systems is given in [18], where the concept of Bayesian artificial intelligence is presented, which is defined as the use of inductive Bayesian techniques in the architectural development of a software related to the artificial intelligence.

Definition of Artificial Bayesian Intelligence

The Bayesian artificial intelligence [19] is based on the concept of uncertainty, the relationships between the variables of a systems have three distinct sources of uncertainty:

- Uncertainty due to measurements
- Uncertainty due to hidden variables affecting the system's phenomena
- Uncertainty of the natural laws in many contexts

Due to such sources of uncertainty, it becomes difficult to find and quantify an exact relationship of causality between two phenomena . A solution, disposing of a sufficiently large dataset, is given by the Bayes theorem:

Considering two events $\mathbf{e_1}$ and $\mathbf{e_2}$, the conditional probability of one is related to the conditional probability of the other by this equation:

$$\mathbf{P}(\boldsymbol{e_1}|\boldsymbol{e_2}) = \frac{\mathbf{P}(\boldsymbol{e_2}|\boldsymbol{e_1}) \mathbf{x} \mathbf{P}(\boldsymbol{e_1})}{\boldsymbol{P}(\boldsymbol{e_2})}$$

Thanks to this theorem, it is possible to quantify the causality relationship between two phenomena, this brought to the design of probabilistic models applicable to real contests, an example of such models are the Bayesian Networks.

Relationship with Self-Adaptive Systems

Bayesian artificial intelligence can be used in a Self-Adaptive Software to make run-time decisions, avoiding the burden to consider all possible situations during the design phase.

Bayesian networks are defined as acyclic graphs where nodes are undefined variables, each of them having a table of conditional probabilities which quantifies the effect of the previous nodes on the current node.

Such networks have already been used in causal probabilistic models to make predictions about the satisfaction of non-functional requirements, as exposed in [20].

An evolution of the Bayesian networks, the decision networks is presented in [19]. A decision network introduces the concept of “decisional node”, representing the possible alternatives that can be chosen given a certain decision to make.

The choice is made by quantifying the utility of each possible decision, called Expected Utility. Supposing that **d** is the set of the possible decisions and $\mathbf{UA_k}$ the expected utility of the decision $\mathbf{d_k}$ following an event **e**, the expected utility is defined as:

$$UA(d_k|e) = \sum_{xi \in X} U(x_i, d_k)\, x\, P(x_i|e, d_k)$$

Where **X** is the variable considered for the decision, x_i a generic value and $x_i \in X$ the range of the possible values assumed by **X**. The variable **e** represents the occurring of an event. $U(x_i, d_k)$ represents the utility of all possible states following the decisional alternative d_k. The alternative d_k with the highest value of **UA** is the one that is selected. In the implementation presented in [20], the variable **X** represents the degree of satisfaction of the non-functional requirements.

To the model previously presented follows the implementation of the Dynamic Decision Networks, that are decisional networks accounting time variance in their model. Dynamic Decision Networks (DDN) represent each node with various instances n_t, and t={0, ... ,n}.

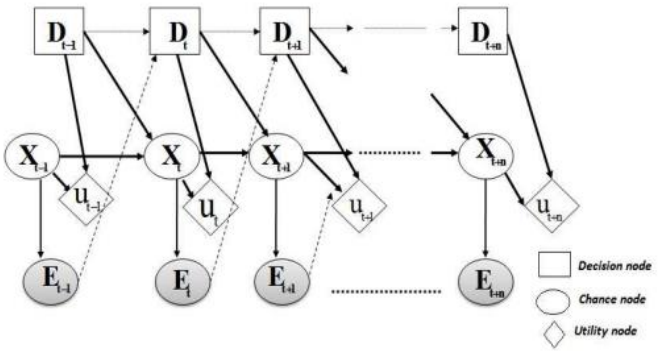

Figure 1.2 Dynamic Decision Network structure

Advantaged and disadvantages

The determination of the satisfaction degree of non-functional requirements, given its uncertainty, can be achieved through modelling the system as a Dynamic Decision Network, giving the system the possibility to decide which configuration to adopt basing on a clearly defined algorithm and a specified model.

An issue related to Dynamic Decision Network is to quantify the extent of the modifications that can trigger the system adaptation, in [18] the concept of “surprise” is defined as the Kullback-

Leibler divergency [21], estimating the divergency between the probability distribution defined by the model, which is the one expected (a priori) and the probability distribution defined by the gathered data (a posteriori). Although, even with such information at disposal, is still difficult to determine if an external modification is so extensive to require an adaptation or is just a transitory modification and an adaptation would result to be useless and expensive.

1.4.7 Semantic Web

Definition of Semantic Web

The Semantic Web can be defined as the part of the web that has a meaning for the machines [67]. Most of the web page contents can be understood only by human users but cannot be interpreted by the machines. An example that can explain this lack of comprehension by the machines is the semantic ambiguity given by the results of a search engine (for instance, the word “mouse” can define both the animal and the input device for computers).

The main purpose of the Semantic Web is to code inside web pages the meaning of their content, making it understandable by the machines as well. The goal of the Semantic Web is to turn the Web from an archive of documents linked between them by the hyperlinks to a distributed knowledge base [75].

Differences with the traditional World Wide Web

Since it was born in the 90s [68], the World Wide Web underwent a quick and huge growth, to the point of defining such growth as an “explosion”, although noticeable, this growth led to the creation of a network rich in contents understandable by the users but poor in contents understandable by the machines.

The traditional purpose of the World Wide Web is to provide to its human users HTML documents hyperlinked between them. The purpose of the Semantic Web is to create a network of contents that can be interpreted by the machines.

Semantic Web Features

Semantic Web is an extension of the traditional World Wide Web, characterized by a decentralized structure. The knowledge representation accessible to the machines consists of two main components [67], a collection of structured data and a set of inference rules.

To implement the two components previously presented and ultimately the distributed Knowledge Base that implements the Semantic Web, two main standards have been created: **XML** (eXtensible Markup Language) [69] and **RDF** (Resource Definition Framework) [70].

The XML standard

The standard XML was created to formally define the data structures belonging to a domain, creating the collection of data structures necessary to the Semantic Web. In the traditional World Wide Web, the HTML standard (HyperText Markup Language) can express only the way the information contained inside the web page should be displayed by the browser but does not formally define the data structures contained inside the page. The XML, being a formal language to describe data structures, can fill this absence.

An .xml file, written in XML language, is processed by a unit called parser, which verifies its correctness. XML documents have a tree-based structure, which starts from a root node and ends with the leaf nodes. The correctness of an XML document, meaning that it has the data it should and with the correct hierarchy, is defined through the DTD (Document Type Definition) files [72].

The RDF standard

The RDF standard in Semantic Web has as its main purpose to define the meaning of the elements formally described through XML. RDF defines such meaning in a triple set, each composed of a **subject**, a **property** and a **value**. The subject is the entity the triple refers to, the property represents what is related to the subject and the value is associated to the property

A simple example of an RDF triple in the human language is defined by an elementary sentence composed by subject-verb-object: “The barber shop is closed on Monday” or “The stadium is closed tomorrow”.

Subject, entity and value are each identified by a unique identification string called **URI** (Universal Resource Identifier), this eliminates the issues related to ambiguity (for instance, the issue presented with the word “mouse” in the previous example is avoided, since the animal will have a **URI** and the input device for computers another one).

An example about how effectively the RDF standard can be used is Dbpedia [71], which consists of a database that can be interrogated using RDF queries to retrieve specific information like the current USA president or the capital of Germany.

The RDF standard has a key role in the Knowledge Representation of the Semantic Web and gave life to the OWL language, the most popular language in the Web Ontologies description, which will be presented in the next paragraph.

1.4.8 Web Ontologies

Definition of ontology in Computer Science

The word “ontology” comes from the philosophy. The word describes a branch of the metaphysic focused on the study of the existence, particularly in determining which entities exist and how the world is structured [74].

In Computer Science, this word has a different meaning; ontology is defined as a document or a file formally defining the relationships between terms [67].

The ontology is formed by two elements, the **taxonomy** and the **inference rules set**. Ontologies are used to define exhaustively the elements and the properties belonging to a certain domain. From the practical point of view, it is useful to define with the same language both the taxonomy and the set of inference rules, using elementary triples in which each of the component is identified by an URI. OWL language was created to satisfy such requirements [73].

OWL language was created by the World Wide Web Consortium from the Description Logic [75], which is a set of formalisms used to represent the logic knowledge belonging to a domain. The Description Logic is based on three elements, that are the **concepts** (also defined as **classes**), the **individuals**, defined as specific instances of the classes and the **roles**, which represent the relationships between individuals. OWL describes with the same language the domain (**taxonomy**) and the axioms related to the domain (**the inference rules**), calling the first set **Terminology Box** and the second one **Assertion Box**.

Application of the ontologies in problem solving

Ontologies can be useful to define in a transparent way the elements of a domain, making easier to implement its related functionalities. Ontologies can also make web searches more accurate and associate semantic information to HTML pages, allowing their understanding by both users and machines, thanks to the ontology contribution.

Ontologies in Self-Adaptive Systems

Ontologies can contribute in different ways to the design of Self-Adaptive Systems. A fundamental contribution is given by the possibility to create agents which can communicate between them even if not specifically designed for doing so, using the ontologies as common ground for communication, they just need to share the mean of communication and the data format to start communicating.

Using ontologies is a solid framework for creating multi-agent systems, which are Self-Adaptive systems consisting of different agents communicating between them.

Another crucial aspect related to the ontologies is the possibility to justify the assertions that are made through the operations of “reasoning” (the extractions of inference rules from the ontology). The capability to explain how such rules were inferred can help to determine the reliability of the information acquired. Using ontologies gives the possibility to implement Self-

Adaptive Systems reasoning over a problem and explaining also the reasoning operations performed for determining a problem's solution.

Another useful application of ontologies in the Self-Adaptive Systems, particularly to those using the MAPE-K architectural model, is to detect in the Execution phase which services can be used to reach a defined goal. It is possible to decompose the plan generated by the Planner in different micro-services, for each micro-service a search unity can find the appropriate web service to be executed.

Issues related to the ontologies

As specified in the beginning of this paragraph, an ontology is implemented through a set of triples, defining the entity, the relationships and the properties of a domain. One of the main issues is to create such set of triples, that should be based on a common acceptance of the used vocabulary and of the logic related to the considered domain.

Another issue is to unify different ontologies belonging to different domains, an example of such difficulty is given by the words that have different meaning in different domains; when such domains are being expanded so much that they intersect, this kind of issue appears evident.

The ontologies of vast domains can consist of a very high number of terms, in such cases the algorithms that are being used for the reasoning operations can cause significant computational stress.

2. Self-Adaptive Systems in Software Engineering, limits and challenges

2.1 Software process models in Self-Adaptive Systems

2.1.1 Requirements definition and refinement

Requirements definition is a crucial phase in software development. The requirements engineering's purpose is to develop methods to define in a complete and exhaustive way the requirements of a software system.

One of the most used methodologies in requirements engineering is the one based on the software's goals, the *Goal Oriented Requirements Engineering* (GORE) [31]. Such requirements engineering defines the software requirements starting from the goals that the system needs to achieve.

In Self-Adaptive Systems is necessary to define both functional requirements typical of traditional software (which are defined through the goals that the software needs to achieve) and requirements describing the adaptation features that needs to be automated; it is crucial, in requirements engineering, to have an appropriate notation to define such requirements.

Requirements definition using GORE approach is divided in three phases:

- Requirements choice
- Goal analysis and requirements refinement
- Assignment of the responsibilities to the agents

Requirements definition is divided in two phases, in the first phase the requirements are defined in a generical way through the goals that need to be achieved, in the second phase the requirements are formally verified over their completeness and consistency.

Requirements definition

In the Self-Adaptive Software domain, it is necessary to choose requirements notations able to fully describe the system's functionalities.

A promising language for such task is KAOS (Knowledge Acquisition in autOmated Specification) [32]. Such language uses four models, which all combined can create the requirements document. The four models are goals, responsibilities, objects and operations.

The **goal model** is a direct acyclic graph, in which the upper nodes are the generical goals, which are divided in sub-goals more and more specific, until they cannot be divided any further. The sub-goals may be related between them by AND/OR associations. In the first case, it is implied that any of the associated sub-goal can be achieved to achieve the goal described in the upper node, in the second case, all the sub-goals need to be completed to achieve the goal described in the upper node. This feature, to define goal's achievement with different alternatives, makes KAOS a suitable method to describe accurately many Self-Adaptive Systems already at requirements level, without having to face the issue of describing different behavioural alternatives at the lower levels of the system's implementation, such as at the code level.

The **responsibilities model** describes to which agents is assigned the responsibility to achieve each sub-goal. In this case, is defined as an agent an entity that can be both human or software, interacting with the system. A valid criterion to determine when to stop the goals decomposition in sub-goals in the goal model is when the sub-goal responsibility is already completely assigned to a single agent.

The **object model** represents the entities, the agents and the associations, these last elements are defining independent entities that are part of the system but cannot perform any operations, as the agents do.

The last model, the **operations model**, describes the actions made by the agents when they interact with the system's objects. Operations can be composed basing on the data flow, making the agents to communicate between them with the input/output data flow, or event triggered, where an agent's behaviour will trigger another agent's behaviour.

Another promising strategy to define and describe exhaustively the requirements in Self-Adaptive Systems is the framework for requirements definition called i* [33], which is divided in two main components: SD (Strategic Dependency) and SR (Strategic Rationale). The SD describes the dependencies between the actors. The SR describes the stakeholder's interests and how those interests are interacting with the environment and the system. The key feature in i* is the emphasis on why the requirements are the ones described, considering the integration of the system in the organization more than the operations that the system must perform, which are defined in the second phase of requirements definition.

A critical problem in Self-Adaptive Software requirements definition is the uncertainty caused by the environment's alterations, which can change the system's properties, consequently modifying its requirements. A popular language which takes in account this kind of uncertainty is RELAX [35].

The main purpose of relax is to define under which conditions, which of the non-functional requirements can be neglected (reducing their priority or not considering them at all). This feature makes possible the creation of Self-Adaptive Software able to focus on functional requirements under critical conditions, allowing the continuity of service and neglecting some of the non-functional requirements.

Requirements definition

Functional and non-functional requirements of a Self-Adaptive System can change during the System's life cycle. In the case of run-time Self-Adaptive Software, requirements modification at run-time is a particularly difficult task.

A possible solution is to create a system that can monitor itself and access to its own software requirements at run-time, modifying them when it is necessary. The Software capability to monitor itself is called *software reflection*.

From this specific necessity of the Self-Adaptive Software, Software Engineering must adapt to design systems with dynamical requirements, which can be modified at run-time. This brought

to the creation of the *requirements reflection* [34], a software design method in which requirements are defined as run-time entities.

In the traditional Software Engineering, the requirements and the code that implements them are very distant, both in terms of time and semantics. In a run-time Self-Adaptive System, it is important for the system to have, beside the capability to adapt itself, a feature defined as "introspection", which defines the knowledge that a system has of its own internal characteristics.

An approach used to solve the weak coupling between the requirements and the code implementing them is the creation of a "metamodel", which helps the system to determine and modify its features. In [34] an approach in which the software architecture adapts at run-time depending on the changing requirements is exposed (figure 2.1).

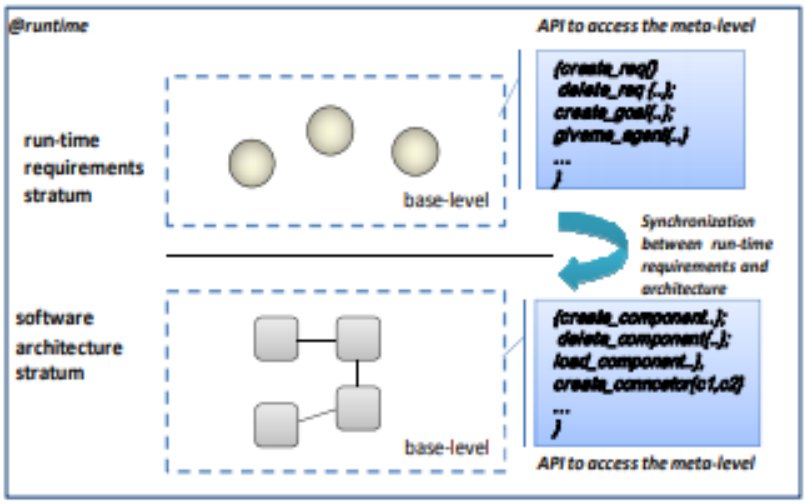

Figure 2.1: The link between requirements and architecture as presented in [34]

Requirements refinement and evaluation

The traditional requirements engineering has a second phase in which requirements are validated, in this phase is determined whether the requirements should be modified or not for the system to work correctly. In Self-Adaptive Software development, where requirements are dynamic, a complete requirements validation is not always possible.

Frameworks as i* and languages as RELAX promote the creation of systems in which requirements are redefined and validated at run-time rather than during the design phase.

In some software systems the architecture remains the same with every adaptation operation performed, in such cases the metamodel is used to check the requirements consistency and correctness.

In many software systems, the adaptation operation brings changes in the system architecture, modifying the way components are connected between them (as happened in the Aspect Oriented Programming). Architecture's modification requires also its validation, besides the one of the new requirements that caused the adaptation in the first place. It is necessary to verify that the software can fulfil its requirements with its new architecture.

2.1.2 Software architecture in Self-Adaptive Systems

Traditional architectural approaches

[37] states that is possible to adopt traditional architectural approaches also in Self-Adaptive Systems. Software components are divided in autonomous, partially autonomous, controllers, auxiliary components and conventional components. The core demonstration of the previous statement follows: given that Self-Adaptive Systems are "compositional" systems, for which the composition operator is consistent (its algebraic properties are valid), therefore the traditional architectural paradigms can be used for their design.

Following the previous statement, from the Goal Oriented Requirements Engineering is possible to develop a Self-Adaptive system's architecture in the same way it is developed for traditional software systems. Although, as [36] underlines, there are divergencies between the necessities of Self-Adaptive systems and the ones of the traditional software systems which can reflect on their architectural design:

- Environment and system monitoring components to make decisions
- Components to determine which decision to make

- Components to perform the adaptation operations

In [31] the architecture is defined with specialized agents that have specific tasks and communicate between the (Multi-Agent Systems).

As concerns the exception management, in a similar manner with what happens with traditional systems that are not Self-Adaptive, the management of all possible exceptions is performed at design time by the developer. But this can be done only if all the possible errors, their symptoms (the behaviour characterizing them) and their solutions are known during the design phase. Sometimes such knowledge cannot be possessed because of the uncertainty related to the environment in which the system operates, or it is too much costly to define and manage all the possible exceptions in the software.

Dynamic architectural approaches

As seen in Aspect Oriented Programming, it is possible to design the run-time Self-Adaptive system architecture as a set of components interacting between them, which interaction is defined at run-time, such operation is defined *weaving* and describes the tailoring of the system's components.

An architectural approach like the weaving is the C2 [39], in which the components, instead of communicating through object, are communicating using connectors and asynchronous messages.

In the C2 Architecture, a hierarchy between the system components is established. In such architecture, each component is only aware of those located at the upper level. This hierarchy allows to develop systems able to handle the architectural changes better, because the dependencies between the components are reduced and related only to the upper level, while the lower levels modifications are ignored.

Asynchronous messages exchanged between the components can be requests or notifications. Notifications go up in the hierarchy, while notifications go to the opposite way. Each component

has an upper and a lower interface and can filter the requests or the notifications by choosing its asynchronous messages forwarding policy (broadcast, multicast or unicast).

The C2 architecture follows these rules:

- Each component is not aware of the ones at the inferior levels (independency between sublayers)
- Communications are only performed through messages
- Since messages are asynchronous, each component might have its own thread, hence the designed software is multithread.
- Shared memory spaces are not allowed in the architectural design
- Architectural implementation is separated from its design, which means that it is not necessary to avoid the use of shared memory spaces in the final implementation or shared threads between the components, if such implementations are allowing an optimal functionality of the system.

C2 architectural model has many advantages, some of them are the independence from the number of layers, the number of threads and the programming language. Another advantage is the separation between the architectural model and its implementation.

Although, C2 architectural components are not completely independent between them, since each component is affected by the ones at the upper level. If a component at the upper level is modified, all the interfaces forwarding requests to it must be modified as well. In order to limit the impact on the system of such dependencies on the architecture's versatility, C2 introduces the concept of "request translation". The request translation is implemented through a mapping procedure for each component between a generical request template and the component's interface, this makes easier to modify the interface of the components in case of an architectural modification.

Reflective architectural approaches

An architectural approach matching with the requirements of Self-Adaptive Software as concerns the *software reflection* and dynamic architecture is the reflective architecture [40].

Such architectural paradigm has as its main goal to clearly define the self-representation of the system's architecture. The software reflection is defined as the perception that the system has of its own architecture.

The necessity to have a self-representation in a Self-Adaptive System's architecture, especially at run-time, is due to the fact that such systems, to perform the adaptations operations they require, may change the way the components are interacting between them, both from the topological point of view (how the components are linked between them) and the strategic point of view (how the links between the components are used).

In the previous paragraph it was already analysed how is possible to design Self-Adaptive software using traditional or dynamic approaches, but these approaches are still limited. One of the most significant limits is defined in [40] as the *Implicit Architectural Problem* (IAP).

The IAP defines the problems related to the software architecture implementation, which consists in the dispersion at the code level of the software architecture and of the consequent architectural design decisions. From this problematic, it derives that the software architecture is not independent from the code as it should, this limits the maintenance and the reusability of the software. The non-functional requirements related to the architecture are also negatively affected by the IAP, since they are hard to satisfy and modify when is necessary.

To avoid the IAP is necessary to define the non-functional requirements through a set of classes inside the architecture itself, instead of implementing them dispersedly at the code level.

A system with a reflective architecture is divided in two levels, the first is the base architecture level and the second the architectural meta-level. The base architecture level describes the system's base architecture, implementing the non-functional requirements as classes, to avoid the IAP. The meta-level is composed by meta-objects, which are the representations of the system policies over the objects in the architectural level. The meta-objects describe both the software policies and the communication policies between the software components.

Metamodels in software architecture

The main purpose of a metamodel is to represent the architectural features, in a way such the software itself can observe and modify its architecture at run-time.

It is necessary to establish a two-way relationship between the architecture and its metamodel, in which the metamodel correctly describes the architecture and in the other way around, the modifications performed on the metamodel are reflecting on the software architecture.

The metamodel is monitored and manipulated by the system's adaptation logic [41]. The adaptation logic is divided in two parts: the monitoring code, which defines when is necessary to perform an adaptation and the reconfiguration code, which defines the instructions the system follows to perform the adaptations.

The set of constraints and the way in which the systems adapts are defined as *adaptation contract*. In a closed Self-Adaptive system, the adaptation contract is static. On the contrary, in an open Self-Adaptive System, the adaptation contract changes during the system's lifetime. An open Self-Adaptive system can face unexpected situations, which were not defined during the design phase.

One of the required elements for the creation of an open Self-Adaptive system is the possibility to dynamically load different *adaptation contracts*, basing on the current system's requirements. Although, is also necessary to regulate the adaptation operations to maintain the system's architecture coherent.

In [41] the system's metamodel is a graph, which has as nodes the interfaces and as vertices the links between the interfaces. Each vertex is labelled with the features that the system can have. In this case the adaptation process is the transition between a graph to another one and the control over the architectural cohesion is made by assuring that each configuration assumed by the system is always a graph.

There are more articulated metamodels, which are focusing on the system's behaviour rather than its architecture, or metamodels describing both, such as the Lancaster model [42]. This model is composed of two elements: the components and the metamodel. Each component has

a meta-interface. The metamodel is composed of four objects: the one for the interfaces, the one for the architecture, the one for the interception and the one for the resources to allocate.

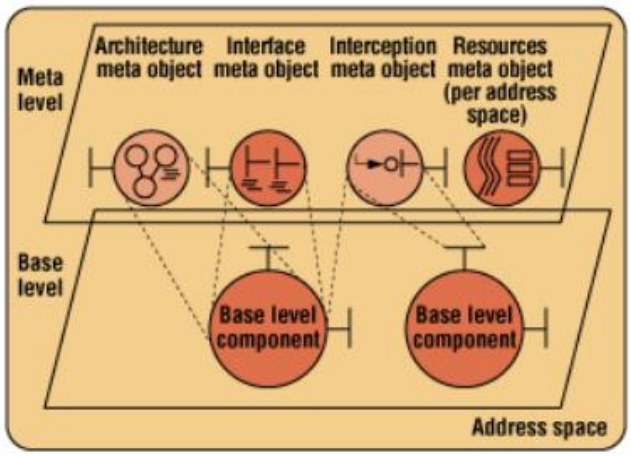

Figure 2.2: The Lancaster metamodel presented in [42]

The **interfaces object** is related to the external part of the components and how these should be linked between them. The **architectural object** is related to the internal composition of the components and is divided in two parts: a graph which determines the interaction between the components and a set of constraints. The architectural object is the one already implemented in [41].

The definition of the architectural constraints can be explicit, by using a dedicated language to define the wanted and unwanted changes in the architecture or implicit, with a control system that verifies if a certain architecture is wanted or not.

The **interception object** is necessary for integrating new functionalities in the system dynamically, specifying the actions to perform before and after the integration of the new functionalities. The last object, the **resource object**, manages the system resources and the tasks, resources can be also defined as primitives or complex.

An example of architectural metamodel for Self-Adaptive Systems: the RAINBOW framework

RAINBOW [47] proposes an architectural model for Self-Adaptive Systems oriented to reuse. One of the most distinctive traits of this framework is the decoupling between the system and the set of adaptation mechanisms. The decoupling between these components is achieved through conceiving the system as a single control loop, where the adaptation logic is the controller and the system is the controlled unit.

The architecture of a system designed using the RAINBOW framework rules is the following one:

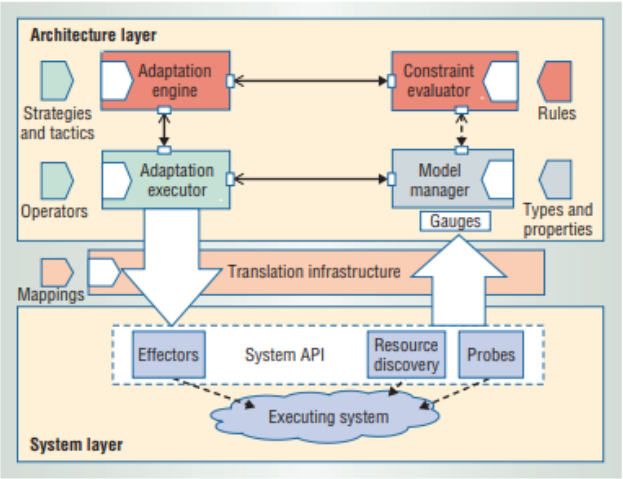

Figure 2.3: RAINBOW framework presented in [47]

It is evident the decoupling between the system layer and the adaptation logic layer. The architecture is divided in three layers, which are hierarchically organized. The highest layer is the architecture, the translation layer and the system layers are following.

The system layer contains the system to adapt, sensors and effectors. The sensors are used to monitor the system, the effectors to perform the adaptation operations. The translation layer has the purpose to map the information in the real system with the ones of its architectural metamodel, which is in the architectural layer. The architecture layer, beside the "**model manager**", which manipulates the architectural model, has other three units: the **constraint evaluator**, which evaluates when to trigger the adaptation mechanisms, the **adaptation engine**, which decides which adaptation procedure to perform and the third one, the **adaptation executor**, which defines the instructions to be followed for the adaptation to take place.

As specified in [47], the degree of reusability of the adaptation logic depends on the degree of similarity between the systems, in values to analyse, properties to guarantee and system structure. The system, translation and architecture layers, having properties which are common in all Self-Adaptive Systems, are highly reusable. Also, the Systems which are already existing and without adaptation logic can be integrated in a RAINBOW framework and become Self-Adaptive Systems; it is necessary to equip such systems with an interface composed of sensors and effectors if these are missing.

2.1.3 Methods of software development in Self-Adaptive Systems

To create a software process model suitable for the Self-Adaptive Systems, it is first necessary to distinguish between the activities of the software process that are carried out during development phase and those carried out during the execution phase.

Beside this, it is important to face another issue, consisting in the uncertainty due to the delaying of some of the design decisions from the development phase to the execution phase. From this delay, it follows that the software process model is not composed of separated phases in sequence any longer but consists of a more evolutionary approach.

Redistribution of the software process components

A development model for Self-Adaptive Systems was introduced in [43], where is also introduced a method to divide the operations to perform in the development phase from the ones the system should perform during the execution phase.

The following picture describes the software life cycle presented in [43]:

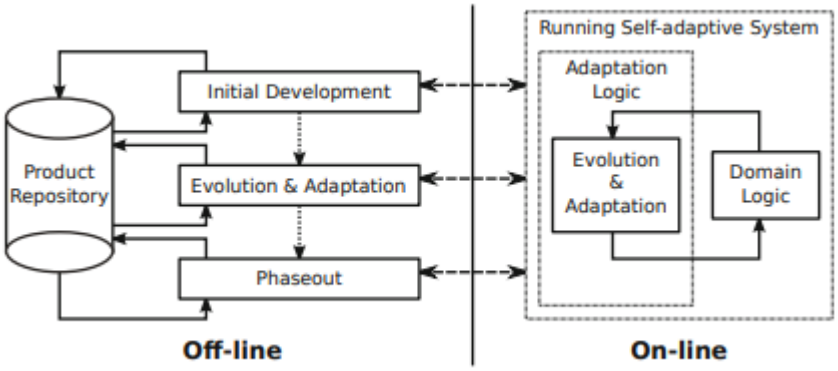

Figure 2.4: Software life cycle described in [43]

The figure is divided in two parts, on the left are located the operations belonging to the traditional software process and on the right the operations specific of the Self-Adaptive system's process model.

In [43] the SPEM language [44] (Software and Systems Process Engineering Meta-Model Specification) is extended to adapt the software process model to the Self-Adaptive Systems. The abstract entities describing roles, activities and products are associated to a super-class, called "process element", which associates to each element costs, benefits, dependencies between one or more elements and an indicator to define whether the process belongs to the activities developed on-line or off-line.

The innovation in the approach exposed in [43] is to define the dependencies between the activities that compose the software process since the earlier stages of the process model.

Another innovation exposed is to define, for each process activity, advantages and disadvantages of being placed in the development or execution phases.

A feature required in the development of a software process model for Self-Adaptive Systems is the support to the design decisions that are made. [43] offers a solution based on the VBSE (Value Based Software Engineering), associating a numerical value to each element of the software process, in a way such that is possible to decide about the design alternative to consider, that is the one with the highest total value.

The value added by each process is defined as the measure with which the single process element contributes to satisfy the goals set by the stakeholders. Associating a value to each process element helps the designer of the system to decide, for example, which elements of the software process should belong to the on-line or the off-line categories.

SMARDT method for Self-Adaptive System design

The SMARDT method (Specification Methodology applicable to Requirements, Design and Testing) is presented in [45] as an evolution of the V-model [46] and is a software process in which for each requirements development stage a testing stage is associated to it. The main issues with the V-model are about the coherence between the phases of the software process and the manual definition of the testing cases.

The SMARDT method introduces an algorithm which determines automatically the testing cases. The software process phases, based on the V-model, consists in a requirements refinement and a testing phase. Each of the previously described phases is defined as a layer. The layers are, in order of implementation: *reflection object*, *logic*, *generic technical concept* and *concrete technical concept*.

The issue of defining the test cases automatically is solved by using an algorithm. The test case derivation takes place in three phases. The first phase is to build the activity diagram, the second to trace the covering paths of such diagram and the third to set the initial conditions belonging to a predefined range, the test is then executed for each of these values, comparing the output results with the ones expected.

The other main issue of the V-model, as describe before, is to assure the consistency between the test cases belonging to the different layers. In [45] this issue is solved by deriving the test cases of a layer directly form the ones belonging to the upper layer. The consistency between the layers allows to apply the test cases of a layer to all its lower layers.

In [45] the SMARDT method is used to design a control system for a self-driven car. The system adapts the controlling parameters to find those optimal for the car to run on a designed circuit.

The designed software uses a closed control loop, which monitors the car's functioning parameters and modifies them basing on the results.

2.1.4 Verification and Validation (V & V) in Self-Adaptive systems

In the development of traditional software systems, the testing phase is one of the final phases before the release, and one of the costliest in terms of time and resources. Self-Adaptive Systems can, for many aspects, be associated to closed loop control systems. Such systems are initially tested in Software Engineering with a model of the system running on a simulation, then the testing in the following phase consists in the fully implemented system running on a simulation and in the final phase the fully implemented system is tested by making it interact with the real environment, the users and the other systems.

Self-Adaptive Systems evaluation

Looking after the testing procedures for closed loop systems, in [48] is introduced a methodology to test Self-Adaptive Software, particularly the ones adopting the MAPE-K architectural paradigm. The testing procedures exposed in [48] are of three different kinds: *one-way*, where the output of the tested component is compared with the one produced by an "oracle", which is the expected output, *in-the-loop*, where the components are tested in a simulation and *on-line*, which tests the interaction between the software and the real environment.

The key concept expressed in [48] is that the testing procedure, if performed in a too advanced stage of the software process in a Self-Adaptive System, makes costly to determine, to verify and to correct the errors that are individuated. To solve this issue, [48] proposes an earlier test methodology, using the *Run-time Model*, which consists in a software model influenced by the environment and the adaptation logic. Using the *Run-time Model,* it is possible to test the MAPE-K software components without implementing the whole controlled system.

Using software simulations, *one-way* and *in-the-loop* testing procedures are used on the single MAPE-K components. When the controlled system is fully implemented, it is integrated in the MAPE-K architecture and other tests follow to evaluate the behaviour of the software in response to the environment and the controller. This last testing procedure is the one defined as *on-line* testing.

Self-Adaptive Systems errors detection

Because of the peculiar features of Self-Adaptive Systems, it is possible the insurgence of exceptions which do not exist in traditional software systems, this phenomenon is particularly evident in context-aware systems. Such systems base their adaptation logic on rules, which can be triggered by modifications of the environment or of the system.

As summoned in [49], there are two main kinds of errors in context-aware systems: errors caused by indeterminism and errors caused by instability. Errors that are caused by indeterminism rise from situations when the system must choose between more than one rule to activate in its current state. Errors caused by instability happen when the system's behaviour is affected by the speed of the rule execution and/or the speed of the context alterations.

The testing procedure proposed in [49] uses an *Adaptation Model*, which accounts the modifications the system underwent and detects the situations of error. The *Adaptation Model* is a finite state machine, using transition rules defined as $R \subseteq S \times C \times S \times A \times N$, where S is a state, C the condition to move to another state S, A the action that is triggered by the transition and N a natural number which establishes the rule R priority, avoiding situations in which the system cannot choose which rule to execute in a certain state, which is the base for the first kind of errors presented in [9], the ones caused by indeterminism.

2.1.5 Quality control for the final users

There are many aspects in which the interaction between the users and the Self-Adaptive Systems differs from the interaction between the users and the traditional systems. A simple

and empirical way to test a software system is to test it with a limited number of users who can evaluate it through a survey, one example is the evaluation procedure for the Self-Adaptive software developed in [12]. The quality evaluation should also account the user acceptance of the system, his trust, his intention to use it in the future and advise it to other people.

One of the issues related to quality control in which Self-Adaptive software differ from the traditional ones is that the Self-Adaptive Software are interacting with the user using data that are being given by the user itself. Therefore, is necessary to guarantee the user's trust, assuring him that the use of his data in the application will not bring harm to him. If user's trust is not guaranteed, he may input partial or incorrect data, compromising the user experience.

To formally verify and measure the acceptance of a Self-Adaptive technology in a software, is possible to use the metrics given by the UTAUT (Unified Theory of Acceptance and Use of Technology) [50]. The UTAUT analyses four factors:

- **Performance Expectancy:** indicated the degree with which the users are expecting the system to be helpful to achieve their goals.
- **Effort Expectancy:** indicating how much effort implies the system's usage.
- **Social Influence:** this factor determines how much the usage of the system is valued by society.
- **Facilitating Conditions**: how much the systems and the infrastructures already available are making easier for the user to use the system.

To the previous factors, also the **trust** should be added, defined as the attitude of the user to trust the use that the Self-Adaptive System will do of his personal data. Trust also reflects on how much the user will value the advices given by the system.

2.1.6 Maintenance and evolution of Self-Adaptive Systems

In the beginning of the Software Engineering, the cascade model defined the maintenance operation as the final phase of software life cycle, allowing only small, corrective modifications [51]. In the following years, it became evident that such role for the maintenance phase was

reductive and that the maintenance operations could be triggered not only by the discover of software errors but also by the new requirements the stakeholder could establish. For this reason, evolutionary approaches rose in the 90s, although such approaches still had limits related to the software architecture. A modification of the software architecture still requires an analysis of its impact on the system, beside the redesign and the re-implementation of the new software.

In a Self-Adaptive System, because of the changes in the requirements or in the context, sometimes is necessary to modify the system's architecture, often even at run-time, it follows that in such cases, the software development paradigms previously used in software systems maintenance cannot be applied efficiently.

In spite the many differences between the traditional systems and the Self-Adaptive ones, some types of maintenance operations are common between them, such as the "corrective" operations, to eliminate the bugs and the "adaptive" operations, defined in [52] as the maintenance operations triggered by the change of the data types elaborated by the system or the data elaboration process. [52] provides also a simple but exhaustive classification of the maintenance operations.

Another divergence takes place between the maintenance operations that can occur in the open Self-Adaptive Systems and the ones occurring in the closed ones. Open Self-Adaptive Systems can deal with unexpected situations (modifications in the software or the environment that were not designed to face). Closed Self-Adaptive systems on the contrary, have a predefined set of adaptation operations. In closed Self-Adaptive Systems, the expansion/alteration of the predefined set of adaptation is a maintenance operation.

The maintenance phase in run-time Self-Adaptive Systems undergoes a partial shift of the operations related to the exception management, which are moved to the deployment phase. This shift takes place because a run-time Self-Adaptive System must detect error situations and adapt itself to avoid them. But this shift is only partial because error analysis is not perfect and the correction for error situations where no suitable adaptation is to be found still belongs to the maintenance phase.

2.2 Limits and challenges in Self-Adaptive Systems

In spite the progress that were made in Self-Adaptive Systems, there are still many problems and challenges ahead, in this paragraph some of them are presented.

2.2.1 Issues related to design

Conflictual goals in a Self-Adaptive System force to realize a trade-off, sometimes is difficult to formalize and quantify the utility of the user goals in order to find the right trade-off point to avoid a conflict.

Approaches used to create Self-Adaptive Systems used until now are very heterogeneous, underlining the necessity to create decentralized implementation techniques, which are efficient and predictable.

The prediction of the effects caused by the adaptation operations is still a difficult task. It is necessary to develop more advanced predictive models.

2.2.2 Issues related to performances

Monitoring a system to make possible its adaptation brings inevitably an overhead, sometimes such overhead is so strong that makes the benefits that Self-Adaptation brings to the system useless.

Overhead caused by the adaptation operations in very dynamic systems, such as the ones implemented on mobile platforms, is often so high to cause performance problems.

2.2.3 Issues related to safety and security

An anomalous behaviour performed by the software to deal with an unexpected situation can bring harm or unsafety and consequently be a liability for the developers. Operations performed

by the software to adapt itself might sometimes bring harm to the system's security and should be stopped when is necessary.

2.2.4 Self-Adaptive Systems challenges

One of the most difficult challenge related to the Self-Adaptive Systems is to explicit the control loops in its architecture, allowing the system designers to use control theory to their own advantage, this subject was already discussed in [17], where an UML extension is presented to allow the description of the system's architecture making control loop explicit.

There is the need to experiment more methods to deal with uncertainty and unpredictability in the large Self-Adaptive Systems, establishing methods to determine how and when a system might face an exception (Proactive Failure Prediction). Related to such field, approaches like UBF (Universal Basis Functions) and SEP (Similar Events Prediction) [24] have been already experimented.

Another need is to create middleware and frameworks that allows to implement the Self-Adaptive features in software systems more easily, reducing the burden carried out by the implementation. An example of a widely used framework for implementing Self-Adaptive Systems is RAINBOW [25][26], which has been already exposed.

Another challenge is to find ways to easily design Self-Adaptive Systems or introducing the Self-Adaptiveness in systems already existing. Some studies have already been done such as [27], [27] and [29].

Another challenge is to enhance the human-machine interaction in Self-Adaptive systems, including the users in the system control chain and finding non-invasive solutions to efficiently gather user data to personalize the experience of the different users. Promising approaches uses knowledge bases queried with OWL and RDF languages, an example is presented in [30].

It is also necessary to ensure that the behavioural modifications do not cause architectural modifications that can compromise the system's functionality. A promising solution for this problem is the implementation of a unit dedicated for this monitoring purpose, defined as

Architecture Evolutionary Manager (AEM) [4]. Such unit can monitor the architecture during the adaptation operations, preventing the system from reaching an undesirable architecture.

2.3 Self-Adaptive Systems application examples

2.3.1 Self-Adaptive Systems for cloud computing

Cloud computing is a promising reality nowadays and is applied to many subjects such as sensor networks management, streaming of multimedia contents, data management, data storage and more recently, even gaming.

In the last years many issues related to the centralized cloud computing architectures have been underlined. Centralized cloud computing architectures generally consists of a central node with a very high computational capability and many leaf nodes, which send the data that need to be elaborated (or the request for a content) and receive a response form the central node. Such architectures often suffer of low scalability and ineffective use of network resources.

To face such issues, new cloud architectural paradigms have been developed such as the Edge Computing [53] and the Fog Computing [54]. These architectures consist in different layers and the computational power is more distributed between the components. An example of such multilayer architecture in cloud computing is exposed in [55]:

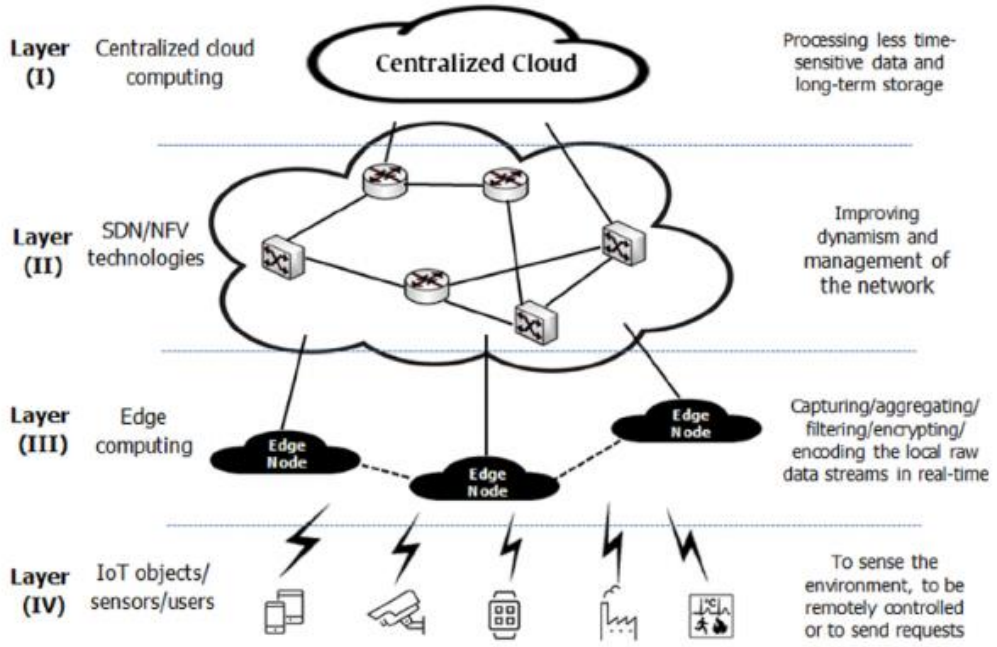

Figure 2.5: Edge computing architecture example presented in [55]

In [55] the network, which is the second layer from the highest one, can be rearranged basing on the system's current requirements, representing an element which can be adapted.

The Self-Adaptiveness applications to this kind of systems allows to preserve the *Quality of Service* and the *Quality of Experience* in spite the network variability. Other benefits related to the use of a Self-Adaptive approach are the sparing of computational resources, the sparing of network resources (resulting in the disposal of a higher bandwidth) and the energy saving, being the distributed computing system able to delegate costly computational operations from mobile devices (with energy constraints since powered by batteries) to nodes which can benefit of a constant energy supply. Thanks to the Self-Adaptiveness it is also possible to personalize the user experience and balance the network load of the nodes in case the network configuration undergoes to some modifications (which can be due to adding/removing a node or other error situations).

In order to implement the Self-Adaptation, is necessary to monitor the metrics related both to the network and the computational load on each node. In the kinds of systems presented in

[55], are often used virtual machines and containers in the nodes to guarantee a higher degree of flexibility. Containers are more flexible than virtual machines and they can even migrate from one node to another when is required. Given the necessity to manage the virtual machines and their containers, it becomes necessary to monitor also the metrics related to them (container size, used memory, percentage of used CPU cycles…). The system should also be able to react to events such as the adding or removal of a node or any other kind of structural modification the network might face. Another important requirement is that the system monitoring must not be excessively invasive, bringing a significant overhead.

By using the monitored data in a cloud computing system and combining them with predictive models, it is also possible to estimate the future network's requirements [56] and to configure the network structure adequately. A cloud computing architecture implementing the Self-Adaptiveness can effectively respond to error situations, manage the network resources autonomously by balancing the computational load and adapt itself to the single users, offering them a better service.

2.3.2 Self-Adaptive Systems for remote medical assistance

In [57] is introduced the design and the implementation of a Self-Adaptive System which purpose is the medical supervision of distant patients, detecting anomalous situations and calling the emergency number when necessary. The system architecture is the one of a *Service Based System*, which have applications in many fields and consists in composing third-parties services to deliver more complex ones.

Self-Adaptiveness application to remote medical assistance systems brings to significant advantages, such as the capability to define emergency procedures based on different alternatives, applying the robustness of Self-Adaptive Systems to critical situations related to the medical field.

The procedures defined in [57] are composed of simpler services, defined as atomics and complex services, which originate from the composition of atomic services. To each service is assigned a description, a cost in terms of quality of service and a statistical fail rate. With such

information, when a work procedure requires a certain quality of service or a certain success rate, the right microservices are chosen to compose the main service.

Another application example of Self-Adaptive Systems to the medical field is shown in [58]. In the clinical field, a Self-Adaptive System should not only account the patient's needs but also the ones of the organization the system is operating within. Particularly, the general goal of a medical software is to offer the best treatment to the patients at the lowest cost for the medical structure.

Using a software system in the medical field requires also that the information displayed for the patients' treatment are appropriate to the knowledge level and the authentication level of the system's user (which can vary from a nurse to a specialized surgeon). A software system for patient treatment must also account the reactions that patients have during the treatment and their past clinical story, adapting itself.

The system presented in [58] implements patient diagnosis through the SWRL (Semantic Web rules) and for each operation performed in the work flow also accounts the results of similar operations that have been performed before.

2.3.3 Self-Adaptive Systems for industrial automation

In big industrial production lines, human intervention to solve unexpected situations which slow or stops the production can be only partially effective, both because the big number of components involved in the production and the high number of possible unexpected situations. To increase the production efficiency to its full potential is not only necessary to automatize the production process, but its management, the inspection of the machines and the management of unexpected situations.

In [59] is presented a production line using the CPS (Cyber physical Systems) to automatize the production process and other two software units: A Self-Managed system to manage the production process and a Self-Adaptive system to detect and solve the issues related to the production.

The system presented in [59] is composed of three units. The first, defined as *smart machine agent*, works with the CPS, analysing the production materials, the produced elements and allowing the communication between the machines. The second unity is the *Self-Managed system*, which divides the work load in sub processes and identifies which machines can execute them and at the lowest time and cost, defining the workflow that the production line must follow. The third unit is the *Self-Adaptive System*, composed of two modules, the first to identify the errors and the anomalous situations and the second to solve such situations.

The errors related to the production line belong to four type of events, which are, sorted by increasing importance: primitive events (related to sensors), basic events (related to the single resources), complex events (related to a section of the production line) and critical events (related to the whole production line). The second module of the Self-Adaptive System, which is the one solving the anomalous situations once they have been individuated by the first one, has two types of intervention, the local one for localized events (type A, for primitive and basic events) and the global one for the events affecting the whole production line (type B, for complex and critical events).

3. Project and framework description

3.1 Project goal

3.1.1 Current project

The goal of this thesis is to design and implement a tool, which makes the creation of Self-Adaptive Systems easier. In particular, the implemented tool allows the user to define the constraints, the preconditions, the conditions and the goals of the Self-Adaptive System to be implemented.

A multi-user approach is supported, to adapt the system's behaviour to the different user profiles, which are stored in the Knowledge Base.

The tool allows to realize a system in which Self-Adaptation politics can be changed at run-time, allowing to modify the goal evaluation policy for each new input, reconfiguring the controller and consequently changing each goal evaluation.

3.1.2 Differences with the previous project

Differently from the previous project, in this one is necessary to adapt the framework to manage different kinds of ontologies, with different kinds of inputs, different goals and different evaluation politics. The management of different ontologies requires the decoupling between the ontologies and the code written to manage them.

Another difference with the previous project is the possibility to modify the weight used to evaluate the goals at run-time, allowing to change the Self-Adaptation politics. The values defined as ratings, which in the previous project were defining the coefficients in the formulas used for the goal evaluation, keep the same function in the current project. The numerical

value associated to each goal will be called **evaluation**, to distinguish it from the **rating**, which is the coefficient used in the goal evaluation formula.

Another divergence with the previous project is the way in which the information related to the determination of the users' preconditions are stored. In the previous project a JSON file was used, in the current project such solution would make the JSON file too large to be easily accessible, therefore the information related to each user are stored in a database.

3.2 Architecture description

3.2.1 Architectural model

The project architecture follows the MAPE-K paradigm, this project is focused on the creation of Monitor, Analyzer and the components in the Knowledge Base necessary for their functionality. In the design phase of each Self-Adaptive System generated with the tool the rating values are specified, which are the coefficients used to calculate the evaluation values related to each goal.

The knowledge base is implemented using the information stored in a database, which is composed of three tables. The first table defines the default rating values for each user, the second table contains the evaluation given to each goal for each user and the third table stores the information related to each user's preconditions ranges. The content of the knowledge base will be explained in a detailed way in a dedicated paragraph.

The input is given through a JSON object, solutions that comes handy because widely used, for instance in remote controlled devices implemented with Raspberry or in the web communications, such as in the AJAX requests.

The following diagram shows the system's architecture:

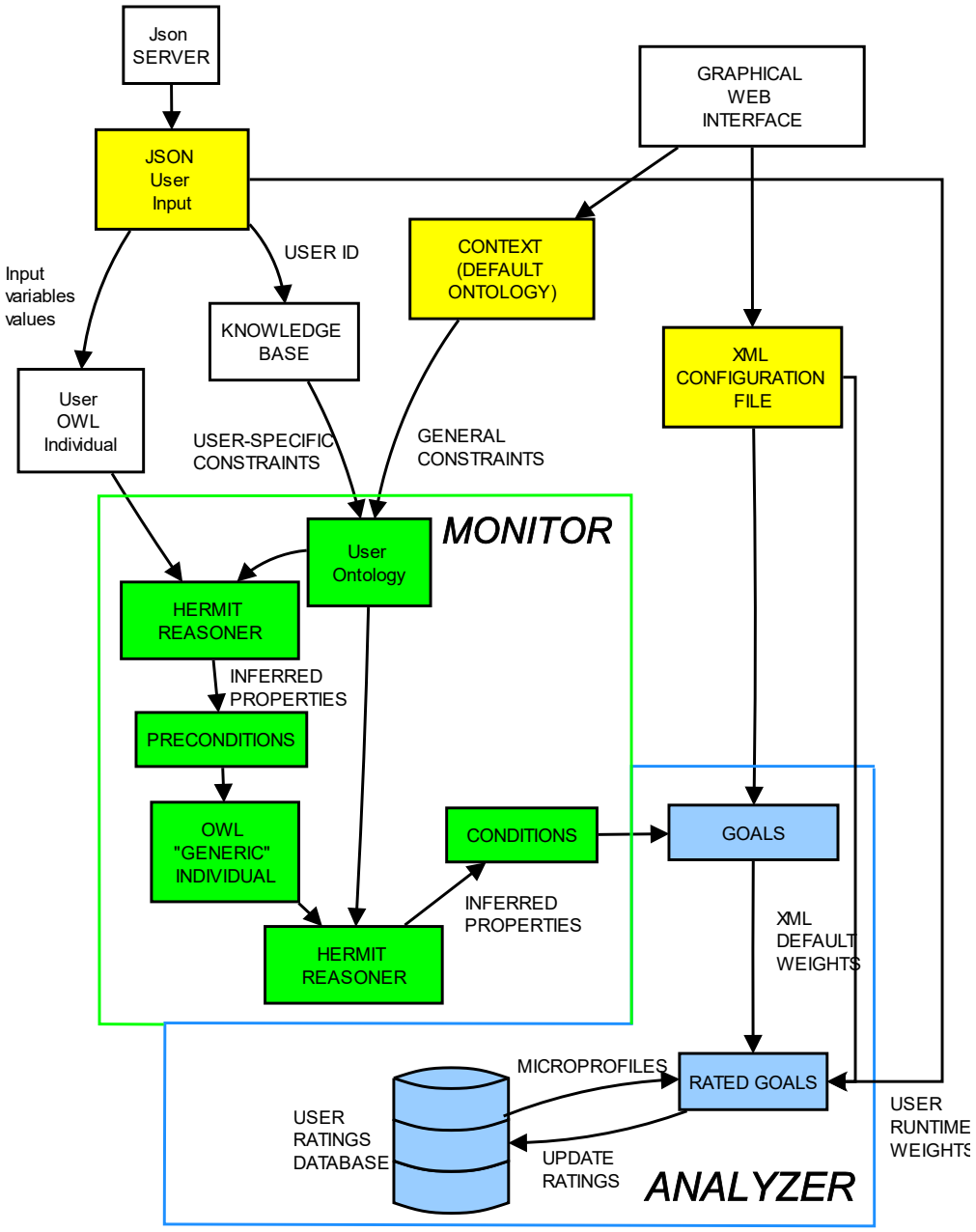

Figure 3.1: The system's architecture, the blocks representing the inputs are depicted in yellow (the JSON packet, the base ontology and the configuration file) , in green the modules composing the monitor and in blue the ones composing the analyser

3.2.2 Monitor

The monitor starts with the System base ontology, which represents the system's context, defined as the set of constraints and classes which are always present and independent from the different users' profiles. An OWL individual is created through the JSON input, with its parameters corresponding to the input values. From the context ontology, the ontology with the user's profile information is created using the user's information stored in the Knowledge Base.

The OWL individual is created with the data coming from the input and added to the user ontology. After this, the Hermit reasoner is used to determine the preconditions which are verified for the user input.

From the preconditions, an OWL individual named as "Generic" (called this way because user-independent) is created. This individual is added to the context ontology and a second reasoning operation with Hermit is performed, to determine the verified conditions from the preconditions.

3.2.3 Analyzer

The instructions about the next steps are stored in an XML configuration file, which describes the following aspects:

- Which goals are associated to the conditions.
- How the goals are evaluated, which ratings are considered for their evaluation and with which coefficients.
- The default rating values, which are used when it is impossible to retrieve their value.
- The elements belonging to the domain of each nominal variable.

Following the instructions stored in the configuration file, the goals are defined from the conditions that are true for the input, it is also possible that more than one condition is true at the same time.

By interrogating the XML configuration file, the user input and the knowledge base, a numerical value (the evaluation) is associated to each goal.

The evaluation of each goal is calculated as the total sum of the rating values defined in the system's model, each of them multiplied by a coefficient, which depends on the rating and the goal that is being evaluated.

$$\textbf{Goal evaluation} = \textbf{rating}_1 \textbf{ x weight}_1 + \textbf{rating}_2 \textbf{ x weight}_2 + \textbf{... rating}_N \textbf{ x weight}_N$$

In case it is not possible to calculate the evaluation associated to a goal because one or more rating values in its formula are missing, if enough user microprofiles are available, the rating is calculated with the Pearson Correlation, as explained in the paragraph about the microprofiles determination.

Once that each goal is evaluated, the values are stored in the user's database, so that they can be reused when it will be necessary to determine the user's microprofiles.

3.2.4 Rating values determination

The rating values are acquired from 4 sources, which are queried in the following order:

- The JSON input, allowing to modify the rating values at run-time for each input and changing the Self-Adaptation policies of the system accordingly.
- The record corresponding to the user in the table **_preferences**.
- If the rating values are not available in the previous two cases, the goal is directly evaluated using the Pearson Correlation, in case enough microprofiles are available (#users > N).
- The default weight is retrieved from the configuration file in case there are not enough microprofiles to evaluate the goal using the Pearson Correlation (#users < N)

3.2.5 Creation of a new system

The creation of a new system starts from the Web interface, which main purpose is to assist the user in the creation of the configuration file and the context ontology. Once the configuration file is defined, the user will use another section of the editor to create the new user's profiles for the system:

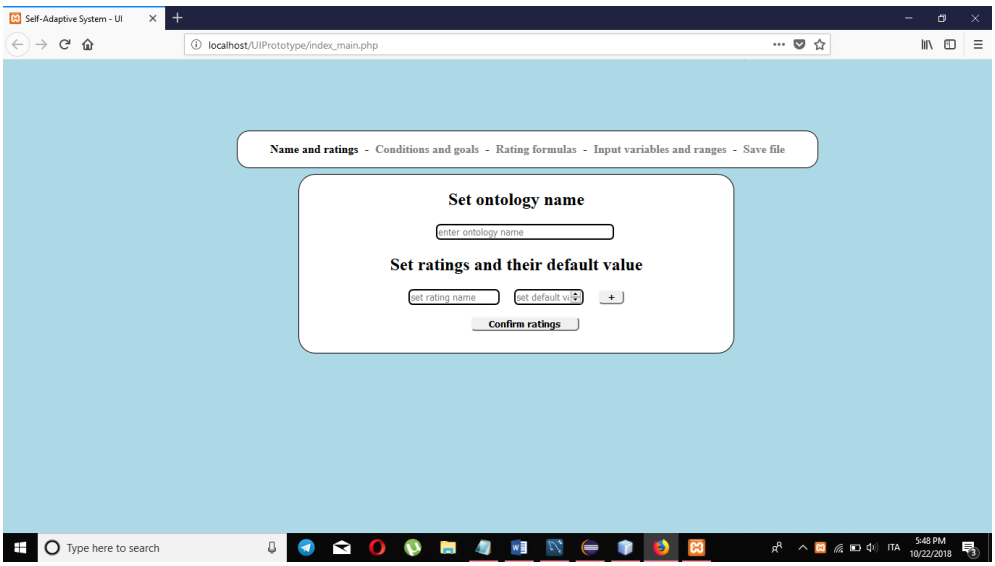

Figure 3.2: Aided creation of the configuration file and of the context ontology

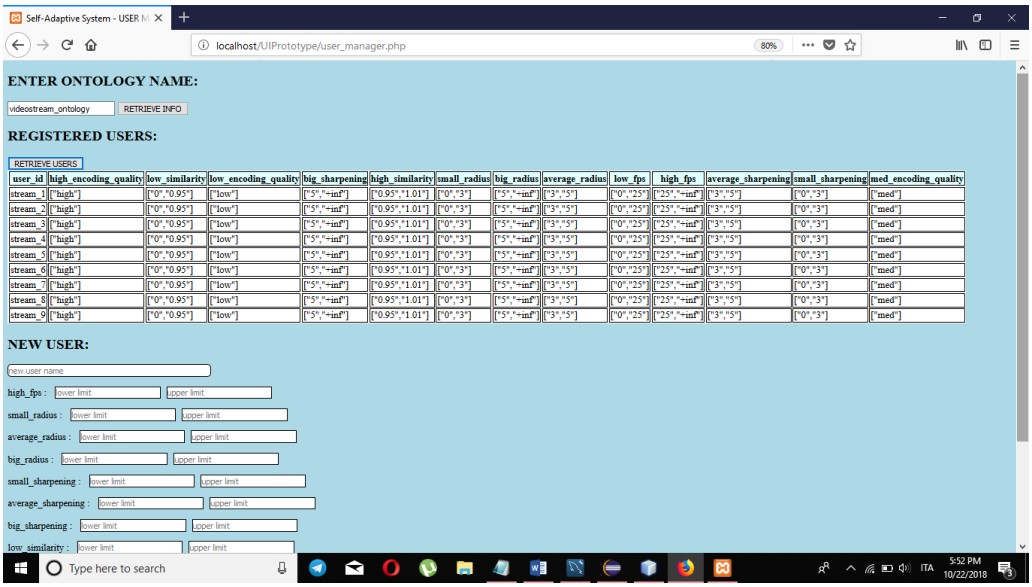

Figure 3.3: User profiles creation

After that the user profile data are entered in the database, a secondary Java program (stored in an executable .jar library) creates the context ontology. The program will read the information stored in the configuration file and will create the context ontology using the default ontology (an empty .owl file) as starting reference, which is the one specific for the system.

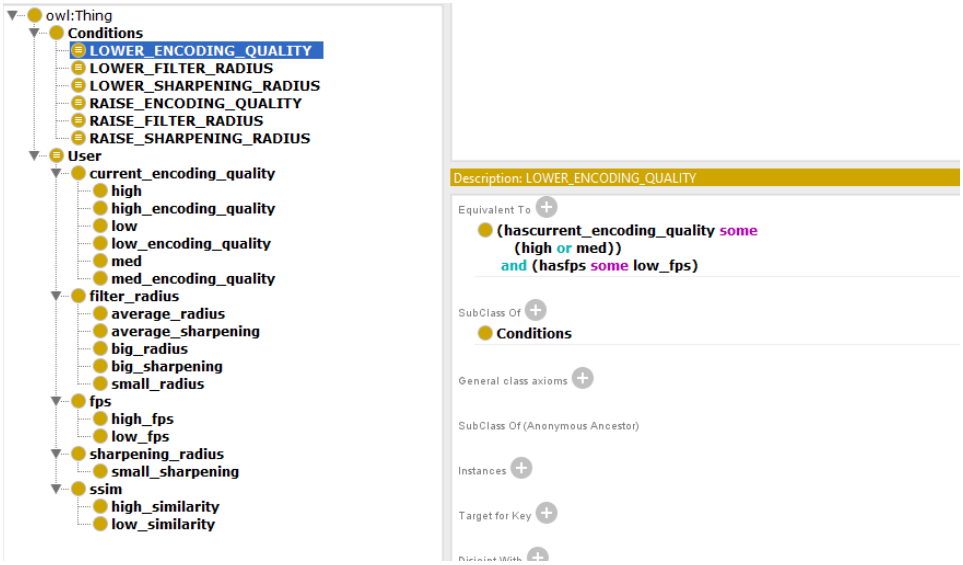

Figure 3.4: Context ontology created with the information stored in the configuration file, conditions have been entered through Protégé

Once that the conditions are defined as subclasses of the **Condition** class, the only thing left to do is provide the .json input to the program; the program will analyse each input defining which goals the system should execute and evaluating them.

4. Solution implementation

4.1 Used instruments

4.1.1 Tools and Integrated Development Environment

The project was developed almost entirely in Java, using SQL in the interactions with the database implementing the Knowledge Base. The IDE used is Eclipse. The Java version used is the 8. The ontologies are stored in the OWL/XML format and for their editing the graphical ontology editor Protégé was used. The web interface used for the aided creation of the configuration file was implemented using NetBeans. The HTML, CSS and JavaScript languages were used to create the web interface and the PHP language used to implement its interaction with the database.

4.1.2 Jar libraries

To implement the project, the following external .jar libraries were used:

- **gson-2.8.5**, used to convert with ease the JSON objects to HashMap ones when necessary.
- **Json-simple-1.1.1** , library used to parse the JSON objects acquired from the file simulating the client's input.
- **mysql-connector-java-8.0.12**, library used to make the program interact with the users' database.
- **org.semanticweb.HermiT**, library implementing Hermit, the reasoner used on the ontologies
- **org.semanticweb.owl.owlapi-3.4.4**, library managing the OWL files and their components
- **underscore-1.28 e underscore-lodash-1.24**, libraries to manage the HashMap objects

4.2 Development method

4.2.1 Input acquisition

The JSON input is simulated through a file called **user_input.json**, which contains a json objects array, each object representing a different input.

Each of the array element is passed in sequence as an input to the system through a client, at regular time intervals, simulating the input acquisition at a predefined sampling time. The input is then received by the server which constitutes the main program, listening to the same port number of the client and starting to process the inputs as soon as they arrive.

4.2.2 Implemented Java classes

To develop the system, a modular approach was chosen, dividing the model in specific subcomponents, each created to implement a block composing the architecture. A class is associated to each relevant element.

The implemented classes are the following:

- **mainClass**, containing the server acquiring the input packets
- **process_json_input,** containing the operation sequence performed over the ontologies to extract the goals and their evaluations.
- **OntologyBundle,** containing the ontology and all its associated elements used to load, store and manage it.
- **preconditionExtractor**, which task is to extract the preconditions derived from the JSON input and the user's information stored in the Knowledge Base
- **conditionExtractor**, which extracts the conditions that are true for that input
- **db_manager**, which manages the communication with the database containing the information about the range values determining the preconditions, the microprofiles and the rating values corresponding to each user.
- **goalRater**, which evaluates each goal.

4.2.3 Intermediate representations

To make the debug process easier, intermediate representations are created during the operations performed on the default ontology (which is the context, containing all the general constraints). The intermediate representations describe the state of the ontology during its processing, in particular:

- The context ontology.
- The user specific ontology.
- The user specific ontology with the OWL individual created from the input.
- The user specific ontology with the preconditions inferred with the Hermit reasoner.
- The user specific ontology with the inferred preconditions and the “Generic” OWL individual.
- The user specific ontology with the inferred conditions.

The intermediate representations consists are .owl files and can be explored using Protégé. After the generation of the intermediate representations, in the Analyse phase the goals related to the verified conditions are evaluated, following the instructions stored in the XML configuration file.

4.3 Faced problems and solutions

4.3.1 Adaptation to a generic ontology

A default ontology can have variables belonging to three basic types:

- **Integers**
- **Double**
- **Nominals**

The last type, the nominal variables, are the most difficult to handle. At first, it is necessary to establish the set of values that the nominal variable can assume. It is also possible that for a

given nominal variable, its subsets can belong to different classes defined as nominal variables as well.

To better explain the case previously exposed, an example related to the previous project can be made. For each user was necessary to set which were the work and the gym days of the week.

The closed set corresponding to the possible values of the nominal variable **Day** is the following:

Day = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}

There are two different preconditions associated to the Day variable, which are nominal types as well:

$$\mathbf{Prec_{day} = \{GymDay, Workday\}}$$

For each user it is necessary to describe in the Knowledge Base the subset of **Day** belonging to **Gymday** and the subset belonging to **Workday**

In the implemented project it was necessary to generalize and automatize the management of nominal variables as well. To perform such generalization, a class with the same name of the nominal variable is created in the context ontology, its children are called as the possible values that the nominal variable can have.

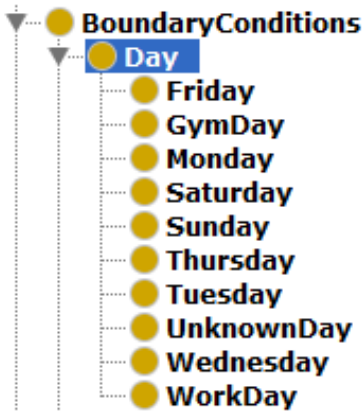

Figure 4.1: All the possible nominal values corresponding to the Day variable, which are all the possible input values for that variable

In the creation of the user-specific ontology, for each nominal variable, the classes with the precondition name will be created and these classes will have as their children the subset of the nominal variable that defines them. This can be explained better with the following figure:

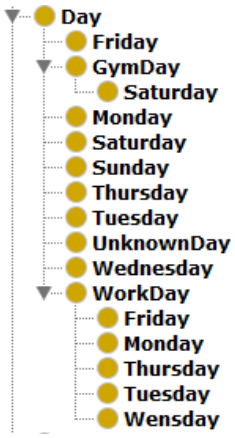

Figure 4.2: Definition of gym and work day preconditions for a user, this ontology was created using the information stored in the Knowledge Base and associated to a user, who goes to gym on Saturday and works from Monday to Friday

Other implementation difficulties were related to the Integer and Double types, which can describe different value ranges for a precondition's domain depending on the different users.

The ranges can belong to three different types:

- From -inf to a finite number
- From a finite number to +inf
- Between two finite numbers

In the Knowledge Base are stored also the information related to which kind of range and for which values each variable triggers a certain precondition.

To implement a solution for the integer and double variables having preconditions associated to their ranges, in the default ontology, for each numeric variable, a corresponding subclass of the "User" class is created. Each subclass has the name of a numeric input variable. Which can

be Integer or Double. Each of these subclasses have as many children as the preconditions associated to that variable. These classes will be filled entering the ranges specific for the user, retrieving the numerical limits of the range from the Knowledge Base:

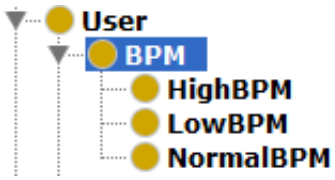

Figure 4.3: Section of the default ontology, which describes the numerical variable BPM (pulsations per minute) and its possible preconditions HighBPM, LowBPM and NormalBPM

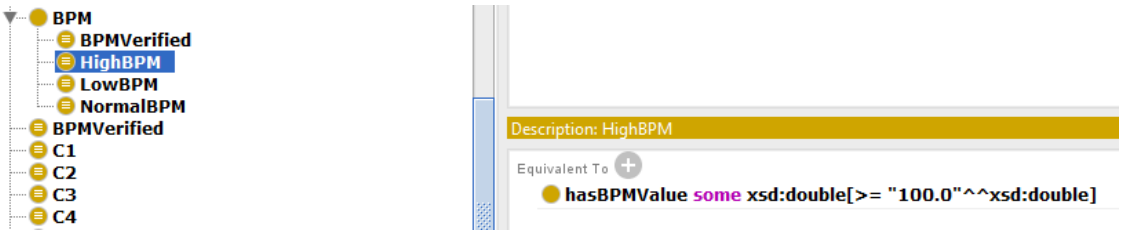

Figure 4.4: Section of the user specific ontology, with numeric values found in the Knowledge Base defining High, Normal or Low pulsations per minute

It is also necessary, to tell from double and integer input variables, to specify their type in the user-specific ontology, creating a DataProperty indicating the User class as domain and the range **xsd:double** if the variable is double or the range **xsd:integer** if the variable is integer.

Once that the preconditions and the conditions are extracted with the two reasoning operations performed by Hermit, one of the issues is to eliminate all the inferred classes derived from the reasoning procedure that are redundant and therefore useless. In order to do so, specific methods were implemented, which are controlling the superclass and the subclasses of each inferred class, eliminating the unnecessary ones.

It might also happen that the default ontology already contains an individual with the same user_id of the current user but created from a previous JSON input. If such residual individual

has properties as well, they might overlap with the ones of the new input, to prevent this, a specific method was implemented to reset the ontology before each use.

4.3.2 Conditions and goal association

Once that the conditions for a given user with a given input are determined, it is necessary to determine which goals are to be considered as well.

In the previous project, a table associating a set of goals to each condition was implemented. Although, in this project it was chosen to store the information about the goal set associated to each verified condition in the configuration file, alongside the formulas that are used to evaluate each goal.

One issue that revealed difficult to handle in the design phase was the overlap of different conditions, which can be true at the same time and might share some of the goals. The web interface and the source code of the ontology processors were implemented in a way to face such cases.

4.3.3 Evaluation policy configuration

The evaluation politics are defined in the XML configuration file, associating each goal to the weights that are to be considered for its evaluation.

When the rates that are necessary to calculate a goal are missing for one or more goals to be evaluated, the redundancy levels for rating retrieval that were shown before come in. The rating values, if missing from the input, are retrieved from the user record in the knowledge base, if they are missing from there too, there are two courses of actions: if there are enough users to determine the goal evaluation through the microprofiles, the goal is calculated using the Pearson Correlation; if there aren’t enough users for such operation, the default rating values stored in the configuration file are considered.

4.3.4 Web interface design and implementation

To easily create the configuration file, a web interface was implemented consisting in five main forms:

- Ontology name
- Rating names and default values
- Conditions and associated goals
- Determination of the weight associated to each rating for the goals' evaluation
- Definition of the input variables and their range

The web interface is composed of 6 main files:

- **Index_main.php**, corresponding to the page containing the HTML of the forms used for the configuration file creation.
- **UIStyle.css**, containing the pages styling elements.
- **UIScript.js**, containing the JavaScript code associated to the creation assistance interface.
- **Ajax.php**, which receives the information stored in the forms to fill the configuration file.
- **User_manager.php**, which is the web interface for the user profile creation in the database, the created user profiles define the numerical values for each range and the nominal variables values for each user's specific preconditions.
- **USERScript.js**, which contains the JavaScript associated to the profile creation page.
- **Ajax_USER.php,** which interacts with the database, creating new tables when necessary, storing new user profiles and retrieving the information contained in the existing ones.

The information contained in the **index_main.php** form is sent to the **Ajax.php** file through a POST XmlHttpRequest and then saved in the XML configuration file having the same name of the ontology. The form allows to define also the weights with which each rating is summed for the evaluation of each goal.

For each form section, starting from the first, once the data have been confirmed using the saving button, the next section opens, at the end of the form's compilation, the configuration file and the context ontology are created.

4.3.5 Database management

The database is composed of three tables for each Self-Adaptive System and the tables' name are depending on the ontology name:

- The **ontology_name** table contains all the ratings corresponding to each goal that have been calculated for each user (when those values are available).
- The **ontology_name + "_ranges"** table contains the information on the nominal, integer and double variables of each user profile, which are used to determine the preconditions, which are differing between the different users.
- The **ontology_name + "_preferences"** table contains all the specific rating values associated to each user, at the user creation its row in this table is empty because no value is still defined. The row contains the last rating values put in the JSON input and associated to the user, this table is the second source for the rating values after the JSON input and comes before the goal evaluation through the predefined values or the goal evaluation estimation through the microprofiles of the other users.

Two components of the system are interacting with the database implementing the Knowledge Base, which are:

- The **web interface** that interacts with the database by defining the new user's profiles, creating the **_ranges** table and the information related to each user
- The **Ontology processor** written in Java to process the input through the ontologies, which interacts with the database by extracting the user profile corresponding to the current user to determine the verified preconditions and the rating values (when available), calculate the microprofiles (when required) and updating the database each time the goals related to a user are evaluated.

The following figures are exposing the 3 tables related to the Video Encoder, explained in the paragraph 4.4.3:

user_id	set_low_encoding	set_high_encoding	set_average_encoding	set_average_radius	set_average_sharpening	set_high_sh
stream 1	9	3.4	6.2	3.4	5.2	NULL
stream 10	5	2	3	4	2	1
stream 11	5	1	1	2	2	5
stream 12	3	2	3	4	5	3
stream 13	4	3	1	2	2	2
stream 14	2	4	2	3	2	NULL
stream 15	4	1	3	2	4	1
stream 16	2	3	5	2	2	2
stream 17	4	4	1	2	2	NULL
stream 18	4	6	3	1	3	NULL

Figure 4.5: Section of the table containing the numerical values (the evaluations) associated to each user for each goal

user_id	rateQuality	rateNoiseReduction	rateFluidity
stream 1	1	2	3
stream 2	3	1	2
stream 3	1	5	1
stream 4	2	1	1
stream 5	2	3	1
stream 6	3	2	1
NULL	NULL	NULL	NULL

Figure 4.6: Table "_preferences" containing the rating values associated to each user

user_id	high_encoding_quality	low_similarity	low_encoding_quality	big_sharpening	high_similarity	small_radius	bi
stream 1	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 2	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 3	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 4	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 5	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 6	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 7	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 8	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
stream 9	["high"]	["0","0.95"]	["low"]	["5","+inf"]	["0.95","1.01"]	["0","3"]	["5
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NUL

Figure 4.7: Section of the "_ranges" table, containing the ranges associated to each variable for each user profile

4.3.6 Creation of the context ontology

The context ontology is created using a secondary program written in Java, which extracts the information contained in the configuration file and creates an ontology following its instructions, in particular:

- The nominal variables and their specific ranges values (with the name of the ranges as well).
- Integer and double variables with their ranges name.
- Creates an ObjectProperty for each variable, with **User** as domain and the variable class as range.
- Creates a class for each condition as a subclass of the **Conditions** class

To make the ontology fully operational at this point it is only necessary to define the conditions' logic in the context ontology.

For this task the Protégé editor can be used, defining the logical proposition characterizing each condition in the "Equivalent to" section. To set each condition the classes and the ObjectProperty generated by the secondary program can be used.

4.3.7 Microprofiles determination

In case one, some or all the rating values that are necessary to evaluate a goal are missing and there are enough users to evaluate the goal through the microprofiles (#users >N), the "**_preferences**" table is queried to retrieve the goals evaluations that are related to each user.

For each goal that cannot be evaluated because of the missing rating values, the Pearson Correlation between the current user "v" and each other user "u" is determined:

$$PC(u,v) = \frac{\sum_{G \in \mathcal{G}_u \cap \mathcal{G}_v} (r_{uG} - \bar{r}_u) \cdot (r_{vG} - \bar{r}_v)}{\sqrt{\sum_{G \in \mathcal{G}_u} (r_{uG} - \bar{r}_u)^2 \cdot \sum_{G \in \mathcal{G}_v} (r_{vG} - \bar{r}_v)^2}}$$

Figure 4.8 Pearson Correlation calculation formula, only the known goal values for both the user u and v are used

After calculating the Pearson Correlation with the other users, the Pearson Correlation is used to estimate the numeric value in which the evaluation consists:

$$\hat{r}_{uG} = \frac{\sum_{v \in \mathcal{N}(u)} PC(u,v) \cdot r_{vG}}{\sum_{v \in \mathcal{N}(u)} |PC(u,v)|}$$

Figure 4.9: Formula to calculate the goal evaluation estimation using the Pearson Correlation values

It is also possible to select, for the goal estimation, only the users in the database which have the highest Pearson Correlation values with the current user. Those users are defined as “neighbours” and are the users showing a behaviour more likely to the user for which the goal evaluation estimation is required.

Once each goal is evaluated, the values are stored in the database, so that the latest evaluation values are always available for each user if it will be necessary to extract the microprofile values again in the future.

4.4 Examples of implemented Self-Adaptive Systems

To show the effectiveness of the implemented framework, different examples have been created, each related to a problem that can be solved using Self-Adaptive Systems.

4.4.1 Content Distribution Manager

In this example it will be emphasised how this project can be also used for the resolution of one of the most problematic issues related to Self-Adaptive Systems, which is the description of conflicting goals. It might happen that for a Self-Adaptive System, performing the operations to satisfy a goal related to a required feature may affect another feature negatively. To implement this concept in the system's logic, in the rating evaluation formula, the rating of the features conflicting with the goal will have a negative coefficient. Consequently, the higher is the priority of the conflicting features, the lesser will be the conflicting goal evaluation.

The system on which this example is based is a Content Distribution Manager consisting in a server pool. The purpose of the CDM is to provide a kind of content, for instance breaking news, to its users. It might happen that due to an event or in a time of the year, the requests for the content delivered by the CDM experience a very high peak, characterized by a high number of requests in a short amount of time.

To face such situations, the system can decide between two options:

- Increase the number of servers in the pool, to satisfy the higher number of requests
- Lower the delivered content quality, sparing bandwidth (for instance distributing web pages with textual content instead of multimedia content)

Although, taking one of the two courses of actions has a cost: increasing the number of servers in the pool, more resources are used; on the other hand, lowering the content quality leads to a user experience degradation. Given the choices that the system can make, two rating values were defined:

- rateQuality: indicates the priority of the quality preservation over the distributed content

- rateResources: indicates the priority of the resources sparing (the number of servers allocated to the pool)

The input variables are:

- **Bandwidth**, double indicating the current bandwidth
- **Pool**, integer indicating the number of servers belonging to the pool and used to distribute the content
- **Quality**, nominal value indicating the content quality, which can be High or Low

Different users are corresponding to different kinds of Content Distribution Manager, which have:

- A variable number of available servers (some a few and other dozens)
- Different bandwidth requirements (some services require a smaller bandwidth than others)

The different conditions that may be verified are the following:

- **LESS_SERVERS**, when is possible to deallocate servers from the pool
- **MORE_SERVERS,** when is necessary to increase the number of servers allocated to the pool
- **CONTENT_TEXTUAL_MULTIMEDIA**, when is possible to raise the content quality
- **CONTENT_MULTIMEDIA_TEXTUAL**, when the content quality must be lowered

To each of these conditions is associated a goal having the same name. There can be situations where is possible, because of the available bandwidth, to release some servers or increase the quality of the contents shared by the CDM. The decision will be made considering the rating values given to the quality and to the preservation of resources.

Since the rating values can change at run-time, it is possible to reconfigure the system in real-time, modifying its Self-Adaptation policy from quality to resource preservation or vice versa.

4.4.2 Airplanes with variable geometry wings

The following example manages the configuration of the wings in airplanes with variable geometry wings. This kind of airplane can change its wing size basing on its current needs. To reach higher speeds the airplane can offer less resistance to the air reducing the wing size; in the phases of landing and take-off on the contrary, it can increase the wing size in order to benefit of a higher lift and manoeuvrability.

In this example the wing size has three possible configurations: small, medium or large.

The input variables are the following ones:

- Wind, double
- Altitude, integer
- Current speed, double
- Current wing size, nominal variable (with small, medium or large as possible values)

Each wing size configuration holds advantages and disadvantages: a small wing size decreases the airplane lift and manoeuvrability, but increases its maximum speed and reduces fuel consumption, on the contrary a higher wing size benefits the lift and manoeuvrability but reduces the maximum speed and increases fuel consumption.

To different users are corresponding different kinds of plane, each with its own parameters related to:

- Operational speed
- Operational altitude

Coherently to what stated previously, the ratings will be two:

- rateFuel: priority over fuel saving
- rateManouver: priority over manoeuvrability

Conditions are the following:

- **LOWER_WING_SIZE**, reduce the wing size
- **RAISE_WING_SIZE**, increase the wing size

To each condition the following goals are associated:

- **LOWER_WING_SIZE**: set_small_size, set_medium_size
- **RAISE_WING_SIZE**: set_medium_size, set_large_size

The system can decide whether to focus on manoeuvrability (always having the largest allowed wing size) or on fuel saving (always having the smallest allowed wing size) basing its behaviour on the user profile data. In this example the advantages related to the run-time reconfiguration are evident, allowing the airplane to change its behaviour at real-time, focusing on fuel saving or on manoeuvrability.

4.4.3 Video encoder

In this example is being considered one of the common problems faced by a video encoder, that is to manage the fluidity, quality and filter size of the encoded frames.

The encoder can change the encoding algorithm, apply filters for noise reduction and high-pass filters for the sharpening. It was chosen to classify each encoding algorithm with its resulting quality, defining it as Low, Medium or High.

The input variables are the following:

- **ssim** (double), variable between 0 and 1, which indicates the Structural Similarity Index, defining the encoded image quality comparing it to the one before the encoding
- **current_encoding_quality** (nominal) indicating the quality of the images generated by the encoder currently used (can be Low, Medium or High)
- **filter_radius** (integer) indicating the size of the averaging filter
- **sharpening_radius** (integer) indicating the size of the sharpening filter
- **fps** (double) frame per second produced by the video encoder

The use of an encoder algorithm generating frames of higher quality decreases the framerate or vice versa, choosing a lower quality algorithm increases the frame rate. The application of

filters with a wider range uses more computational power and therefore decreases the framerate as well.

Different users are corresponding to different video encoders, which have different parameters in terms of:

- **fluidity requirements** (for instance, video stream requires a higher framerate than standard video surveillance)
- **filter size** (higher resolutions will consider wider filters; the same filter size can be big for a video having a resolution of 800 x 600 but small for a video in 4K)
- **noise treatment** (film or other videos that are recorded in optimal conditions do not require as many noise filtering operations as videos taken by cameras, smartphones or generally recorded without optimal conditions).

The system priority can focus on three different elements, from which the ratings are derived:

- **rateFluidity:** priority to the framerate
- **rateQuality:** priority to the frame quality
- **rateNoiseReduction:** priority to the noise reduction operations on the frames

The system conditions and the goals associated to them are the following:

- **LOWER_ENGODING_QUALITY:** *set_medim_encoding, set_low_encoding*
- **RAISE_ENCODING_QUALITY:** *set_high_encoding, set_medium_encoding*
- **LOWER_FILTER_RADIUS:** *set_average_radius, set_low_radius*
- **RAISE_FILTER_RADIUS:** *set_high_radius, set_average_radius*
- **LOWER_SHARPENING_RADIUS:** *set_medium_sharpening, set_low_sharpening*
- **RAISE_SHARPENING_RADIUS:** *set_medium_sharpening, set_high_sharpening*

Basing on the rating values given by the user and on the current framerate and ssim, the system will decide which filters and which encoding quality to apply to satisfy the requirements set by the controller. As the previous examples, the system controller can be reconfigured at run-time.

4.4.4 Variable frequency Wireless transmitter

This example models a system composed by a transmitter and a receiver in which the communication frequency can be regulated basing on the distance between the two elements. Higher is the communication frequency, lower is the signal range [76], which is the distance it can cover. In urban-like environments, lower frequencies signals have more penetration power over the obstacles than higher frequency signals.

The purpose of this example is to implement a control system to use higher communication frequencies when the distance between transmitter and receiver is lower, exploiting as much bandwidth as possible and to use lower communication frequencies when the distance between transmitter and receiver is larger, preserving the signal strength.

The input variables are the following ones:

- **distance**, double variable indicating the distance between transmitter and receiver
- **frequency**, nominal variable indicating the communication frequency currently used, it can have three values: low, med or high.

Different users correspond to different transmitters. For instance, the signal range for a powerful radio transmitter is far higher than the one of a battery powered transmitter.

The Self-Adaptation policies, particularly the choice of the frequency to use depending on the distance, are defined by the priorities given to bandwidth and signal strength, therefore the rating values are defined as follows:

- **rateBandwidth**: priority over the bandwidth
- **rateStrength**: priority over the signal power

The system conditions and their associated goals are the following:

- **RAISE_FREQUENCY:** *set_high, set_med*
- **LOWER_FREQUENCY:** *set_med, set_low*

The choice of the communication frequency is influenced by the priority given to the bandwidth and the signal strength.

4.5 Examples classification

4.5.1 Self-Adaptive Systems classification

In [77] a paradigm for Self-Adaptive System classification is proposed, it is stated that a Self-Adaptive System can range between two kind of models. The first model represents the classic control systems, in which the controller uses continuous signals that modify the system's behaviour by altering its parameters (as happens with the PID controllers). The second model represents the control systems designed through the Software Engineering, in which the controller uses discrete signals to modify the system's behaviour acting on its architecture.

The classification elements for the Self-Adaptive Systems defined in [77] are:

- **Adaptation goal**, usually defined with a self-* property, which indicates the reason behind the Self-Adaptive System design.
- **Reference inputs**, which are the input variables that are chosen to describe the system's state.
- **Measured outputs**, which are the output variables of the control system that are measured.
- **Computed control actions**, indicating the kind of signals that are used for the control and their goal (for instance system control through parameter modification, process modification or architecture modification)
- **System structure**, which describes the way through which the system is controlled, that can be backpropagation, adaptive controllers (MRAC or MIAC) or reconfigurable controllers (controllers modifying the control algorithm depending on the system's needs)
- **Observable adaptation properties**, which are the target of the adaptation operations and define which properties are allowed thanks to the system Self-Adaptation.
- **Proposed evaluation**, which defines how the system evaluates its own behaviour.
- **Identified metrics**, the set of KPI (Key Performance Indexes) used to indicate the overall performances.

4.5.2 Classification application to the project examples

Since all the examples treated in this thesis are developed with the same tool, some elements characterizing them are common. Using the classification previously presented, the recurring elements are the following:

- **System structure**: identified as a MRAC adaptative controller, given the rating values that can be tuned at real-time to change the system's behaviour and its Self-Adaptation policies.
- **Proposed evaluation**: performed through a software simulation of the controlled system and of the environment.
- **Measured outputs**: the input variables are the same ones that are monitored in output, which makes this element equal to the **measured inputs.**
- **Adaptation goal**: each system has both the need to self-manage, choosing the correct values of the variable it must control and to self-optimize, reaching the optimal configuration defined by the priorities expressed in the rating values.

The following table shows the classification elements applied to each example:

Example	**Measured inputs/outputs**	**Computed control actions**	**Adaptation properties**	**Metrics**
Airplane with variable geometry wings	wing size, wind, speed, altitude	discrete signals for the parameters	aerodynamic resistance, manoeuvrability	wing size
Content Distribution Manager	server, quality, bandwidth	discrete signals for the architecture	resource allocation, content availability	bandwidth, server
Video Encoder	fps, sharpening, averaging, ssim, quality	discrete signals for the parameters	fluidity, frames quality	fps, ssim
Variable frequency wireless transmitter	distance, frequency	discrete signals for the parameters	available bandwidth, signal power	frequency

5. Platform testing

5.1 Testing methods

5.1.1 Testing operations on uncomplete Self-Adaptive Systems

The possibility to test the correctness and the effectiveness of a Self-Adaptive System since its first development stages, without having to wait for its completion, is important to detect and correct the software errors as soon as possible.

As Software Engineering states, the software error correction costs increase the more advanced the implementation stage is. Detecting the software errors as soon as possible can reduce the error correction costs.

The statements made previously can be applied to the Self-Adaptive Systems as well, in which the testing is essentially composed of three main steps. The first step is the testing of a system model in a simulation, the second step the testing of the implemented system in a simulation and the third step the testing of the implemented system in the real environment.

In Self-Adaptive Systems, the testing of the single components is particularly important, since the system is completely implemented and available only in the last development stages.

[48] presents a detailed enumeration of the possible testing procedures that can be performed on Self-Adaptive Software-Systems (SASS) which are also applicable to uncompleted systems. The model used as reference is the MAPE-K, which is also the architectural model used in the project related to this thesis.

A SASS can be divided in two main components: the system deciding and implementing the adaptation (Adaptation Engine) and the software adapting itself (Adaptive Software). A full testing of the SASS systems can take place only when both these parts are fully implemented and have been successfully integrated.

In the partial testing of SASS is possible to simulate the components that still must be implemented using a Run-time Model (RTM), which simulates the states reached by the controlled system and the environment. The Run-time model can be formally described as the entity defining the system's state and the transition rules between a state and the next one.

The testing procedures described in [48] are three:

- **One-way testing**
- **In-the-Loop testing**
- **On-line testing**

In the One-way testing the single components of the system are tested, their output is compared to the one given by an "oracle" with the same input. An oracle is a model able to present the correct output with a certain input. [48] proposes to apply the One-Way testing individually to the Analyzer unit and in sequence to the Monitor and Executor units. It is also suggested to make the testing cases as wide as possible because it might happen that, for a certain input, an error in the Monitor unit might be masked by an error in the Executor unit or vice versa. Extending the used test cases to a wide set, the probability to let such hidden errors undetected is severely reduced.

The In-The-Loop testing consists in the testing of the whole control loop, which is composed by each of the M-A-P-E units and an environment simulation, which, depending on the external factors and the Executor unit output, creates a new input in the next control loop iteration. The types of testing performed In-The-Loop can be black-box or grey-box.

In the black-box testing only the Planner and environment states are compared to the ones given as output by the oracle, which are representing the correct states. In the gray-box testing, the states of the environment, the Planner and the Analyzer are compared with their respective states generated by the oracle.

While the black-box testing can be performed since the earlier development stages, the gray-box testing requires the complete implementation of the Analyzer and the Planner units but is also able to detect errors with a higher level of detail.

In the In-The-Loop testing, the test is over sequences of inputs generated by the actions the system performs on the control environment and not over single inputs as in the previous testing procedures. Such input sequences are called in [48] Run Time Model Simulations (RTMS) and describe the combined behaviour of the environment, the Monitor, the Executor and the Adapting Software.

The third and last testing procedure type, the On-Line testing, requires the possibility to test the SASS system, fully implemented, in the real environment.

5.1.2 One-Way Testing in the designed system

In the Self-Adaptive Systems generated with the framework presented this thesis, the Analyzer unit has been implemented, allowing to compare the output of the single unit $\mathbf{S_i^A}$ with the expected output $\mathbf{S_O^A}$ given by the oracle, verifying if they are equal or not.

$$\mathbf{S_i^M \text{ -> } ANALYSIS \text{ -> } S_i^A}$$

$$\mathbf{S_O^M \text{ -> } ORACLE \text{ -> } S_O^A}$$

Since the Monitor and Analyzer units are the only ones implemented, it is not possible to perform the sequenced testing of Monitor and Executor. For the testing cases to take place, a fake executor was created, which acts on the environment in a predefined way depending on the goal to be reached.

$$\mathbf{S_1^P \text{ -> } EXECUTE \text{ -> } RTM \text{ -> } MONITOR \text{ -> } S_2^M}$$

$$\mathbf{S_O^P \text{ -> } ORACLE \text{ -> } S_O^M}$$

The output of $\mathbf{S_2^M}$ and $\mathbf{S_O^M}$ are finally compared between them. In the testing system the oracle is the ideal state which fulfils the system goals set by the current controller configuration, which is the state having the value of the error function equal to zero. The distance between $\mathbf{S_2^M}$ and $\mathbf{S_O^M}$ corresponds to the error function itself. If the difference between them is a neglectable quantity $\boldsymbol{\varepsilon}$, the two states can be considered the same.

5.1.3 In-The-Loop Testing system testing

Since the Planner was not implemented, the only In-the-Loop testing that can be performed is the black-box testing, which analyses the state of the environment. The analysed environment state is compared with the behaviour the system should have for the goals to be reached.

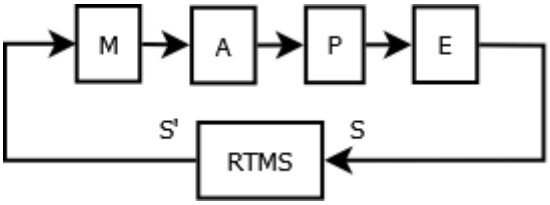

Figure 5.1: System control loop, through which the environment states are compared between them

5.1.4 Run Time Model and Run Time Model Simulation implementation

For each Self-Adaptive System generated with the tool, a Java class called **_RTM** was created. The **_RTM** Java classes are corresponding to the implementation of the different control loop elements, in particular:

- The **variables** of the **_RTM** classes are describing the system's state.
- The **execute_goal(String goal)** method describes the operations to perform to satisfy each goal through a switch-case construct, which constitutes an elementary implementation of the Executor unit.
- The **time_pass()** method defines the environment intervention over the variables considered by the system, giving also the possibility to implement interdependencies relationships between the system's variables. An example of such interdependencies is the implementation of this method in the video encoder example, in which the relationship between fps, encoding quality and size of the averaging and sharpening filter is implemented.

Through a limited number of iterations, an RTM sequence is created, and the resulting states assumed by the system are saved in a .csv log file, to be analysed after the simulations are

finished. The results of the simulations performed on the generated system are presented in the next paragraph.

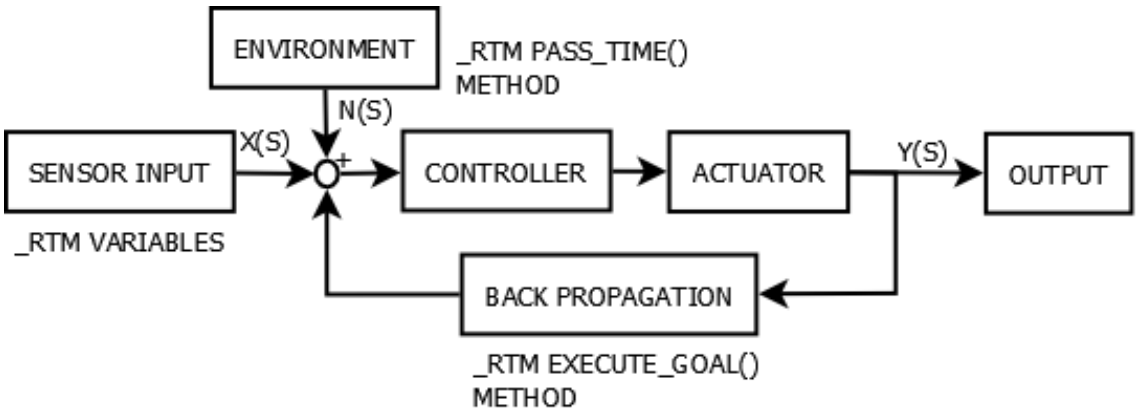

Figure 5.2: System's control loop, each one of the _RTM object components is modelling a specific control loop component

5.1.5 System's property used for verification and validation

According to [76] and [77] the verification and validation procedures performed on Self-Adaptive Systems can use the instruments belonging to the closed-loop control system's theory, since a Self-Adaptive System is a specific kind of closed-loop control system.

In this thesis will be exposed the results of the testing procedures created to verify the following properties:

- **Stability**: assuring that the system does not continue the adaptation process indefinitely and that its state does not diverge from the desired one, the system is stable if its error function is converging.
- **Accuracy**: the error margin with which the desired states of the system are reached, depends on the values assumed by the error function.

- **Settling time**: the readiness characterizing the system in satisfying the defined goals starting from an initial state, this property is determined by the number of steps necessary to minimize the error function.
- **Overshoot**: during the system's settling it must be assured that, for the system to reach the desired state, the inputs are never assuming values that can be harmful for the system. For this to be assured, the values of the input variables will be studied during the settling time.
- **Robustness**: in spite the input noise caused by the environment, the system should reach its ideal state, with a certain margin of error. The system should not diverge from its desired state because of the environmental noise. If the system's error is limited during the simulations in which the environmental noise is provided, then the system is robust.
- **Adaptability**: When the system's controller undergoes a reconfiguration, the system capability to satisfy the goals must be kept intact. To verify such property, simulations in which the controller is reconfigured at run-time will be runned.

5.2 Simulations performed over the examples

5.2.1 Results of the airplane with variable geometry wings simulation

For the RTMS implementation of this system, a Java class called **_RTM_geometry_wings** was used.

The class variables are:

- int alt;
- String wing;
- double speed;
- double wind;
- String user_id;
- static int time_counter;

The time_counter variable is used to define the time passing. As evident with its integer type, the modelled system, as all the other systems generated, is a discrete time system.

The input used for testing consists of two sinusoids, which output value is summed to a bias for the values to be always positive. The first sinusoid is the airplane altitude and the second the airplane speed.

The results for a system configured setting the higher priority over the fuel saving are the ones exposed in figure:

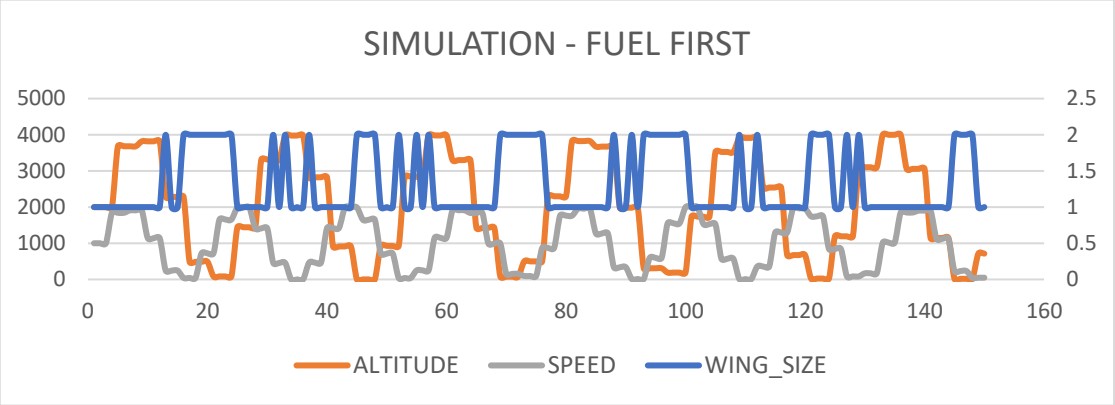

Figure 5.3: Simulation with 150 iterations of the system set with rateFuel=3 and rateManouver=2

From the figure is noticeable that, in conditions of low altitude and/or low speed, the airplane is forced to increase its wing size, such behaviour is expected and regulated by the system's logic, giving the airplane a higher manoeuvrability and lift in such critical conditions.

The results for the configured system with priority placed over manoeuvrability are the following:

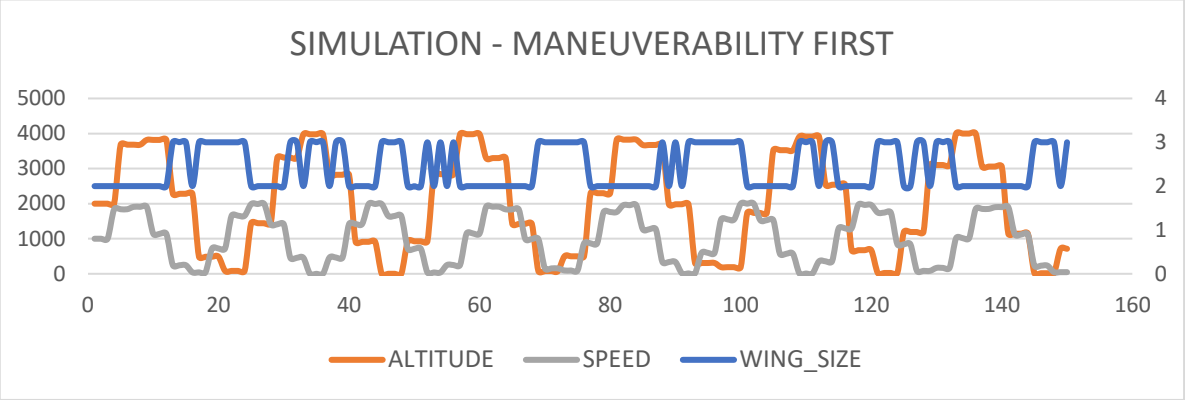

Figure 5.4: Simulation of 150 iterations of the system set with rateFuel=2 and rateManeuver=3

As shown by the two figures, when the priority placed over the fuel saving is higher, the airplane will always try to set its wing size on the lowest possible values. On the other hand, when the priority placed over the system's manoeuvrability is higher, the airplane will always try to set its wing size on the highest possible values.

5.2.2 Results of the Content Distribution Manager simulations

The goal of the control system created in this example is to allow the continuity of a web content distribution service, keeping high the available bandwidth. The available bandwidth can be kept high by increasing the number of servers allocated in the pool or decreasing the distributed content quality.

Although, the efforts done by the system to keep high the available bandwidth might undergo the environmental interference. In this case, the environment is the communication network. Since it is not possible to define the available bandwidth between client and server deterministically, the intervention of the network over the available bandwidth must be considered in the system simulation as well.

The RTMS of this system was implemented using a Java class called **_RTM_server_manager**.

The variables belonging to the class are:

- String Quality;
- int Pool;
- double Bandwidth;
- String user_id;
- static int time_counter;

In this case the input test consists in a bandwidth regulation accounting both the system operations (increasing the bandwidth proportionally to the number of servers used and if the quality of the distributed content is lower) and the environment intervention (introducing a random variable which can increase or decrease the bandwidth).

The results for the system configured placing the highest priority over the content quality are the following:

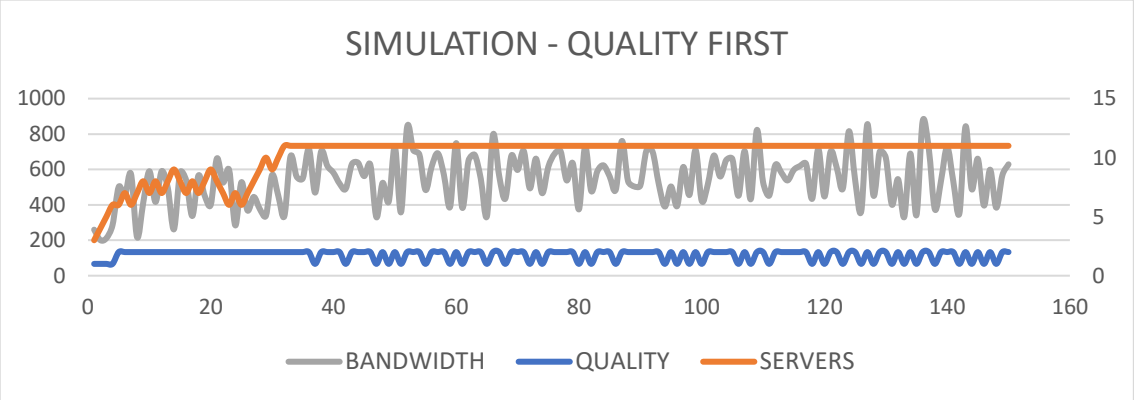

Figure 5.5: Simulation with 150 iterations with the system set with rateResources=3 and rateQuality=2

The results for the system set with higher priority placed over the resource sparing (the number of the allocated servers) are presented in the following figure:

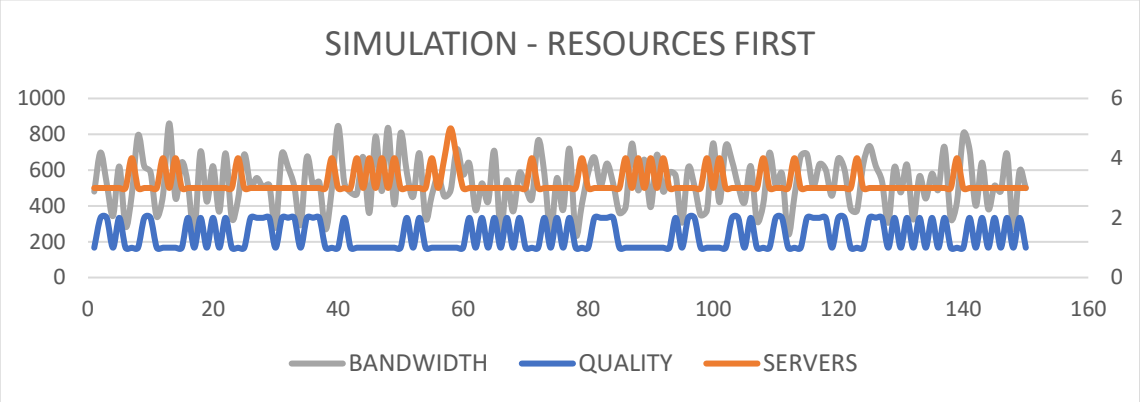

Figure 5.6: Simulation of 150 iterations and with the system set with rateResources=2 and rateQuality=3

As shown in the figures, when the priority is focused on the quality, during stress situations, the system will allocate every available server before reducing the distributed content quality to allow the service continuity. On the contrary, when the priority is placed over the resource saving, the system will resort to reduce the quality of the distributed content before allocating more servers and when the bandwidth will be restored to higher values, will deallocate as many servers as possible before raising the content quality.

5.2.3 Results of the Video Encoder simulations

For the RTMS implementation of this system a Java class named **_RTM_video_encoder** was used.

The class variables are:

- double ssim;
- String current_encoding_quality;
- double filter_radius;
- double sharpening_radius;
- double fps;
- String user_id;

- static int time_counter;

The input test consists in the regulation of the frames per second generated by the encoder, which accounts the filter sizes (reducing the fps when their size is increased) and the encoding algorithm quality (reducing the fps when the encoding quality increases). A random variable able to increase/reduce the frames per seconds is used as well to simulate the environment's intervention. Another variable affected by the environment is the SSIM (Structural Similarity Index) which is defined both by the system's adaptations (the filter sizes used) and the contribution of a random variable.

The fps has a random contribution since the computer running the encoding might have other processes running that can cause variations in the percentage of CPU cycles and generally the resources that the computer dedicates to the encoding, consequently causing a framerate variation. The SSIM random contribution is justified by the fact that some frames might react positively to the encoding, preserving the image structure and some might react negatively to the encoding, because rich of small and colourful details that are lost during the encoding procedure.

The results of the simulation performed placing higher priority over the fluidity are the following:

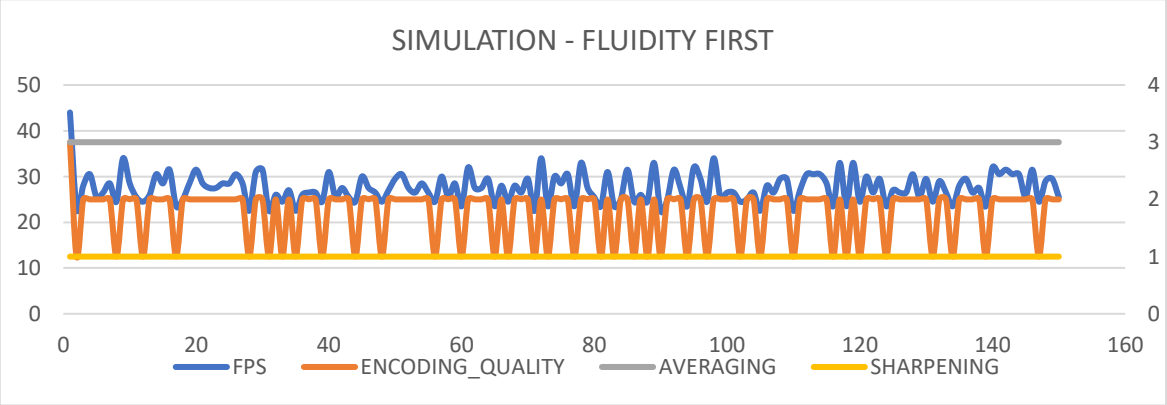

Figure 5.7: Simulation with 150 iterations configuring the system's controller with rateQuality=1, rateFluidity=3 and rateNoiseReduction=1

For the system with its priorities focused on the image quality, the results are presented in the following figure:

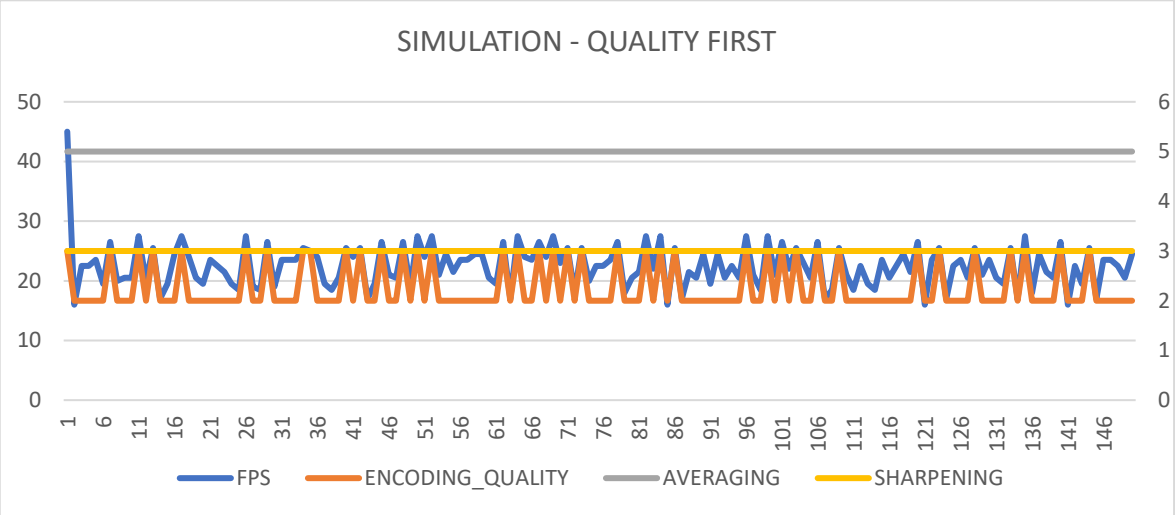

Figure 5.8: Simulation with 150 system iterations when the controller is configured with rateQuality=3, rateFluidity=1 and rateNoiseReduction=1

When the priority is focused on the noise reduction, the system behaves as follows:

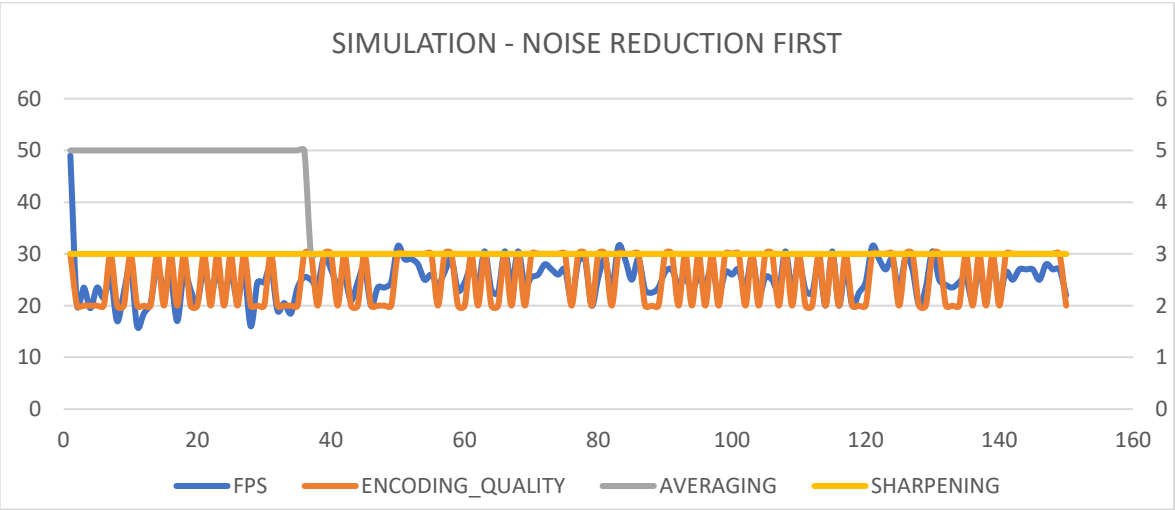

Figure 5.9: Simulations with 150 iterations of the system when the controller is set with rateQuality=3, rateFluidity=1 and rateNoiseReduction=3

As it can be noticed in the figures, when the system's priority is focused on fluidity, lower quality encoding algorithm and filter with smaller size are used when the fps are too low. In the configuration favouring the quality, the filter sizes and the encoding algorithm will be the highest allowed. In the configuration oriented to noise reduction, the system behaves in a similar manner to when is configured on the quality preservation, by using wide filters and high-quality encoding algorithms. The similarity between the noise reduction oriented and quality-oriented configurations is explained by the need to use a high-quality encoding algorithm to reduce the noise, because it reduces the information loss caused by the encoding and allows to preserve the SSIM.

5.2.4 Results of the variable frequency transmitter simulations

To implement the RTMS of this system, a Java class called **_RTM_wireless_strength** was created.

The class variables are:

- String frequency;
- double distance;
- String user_id;
- static int time_counter;

The testing input consists in the distance variable as a sinusoid wave summed with a bias for it to be always positive. The results for the system configured placing the higher priority on the signal strength are the following:

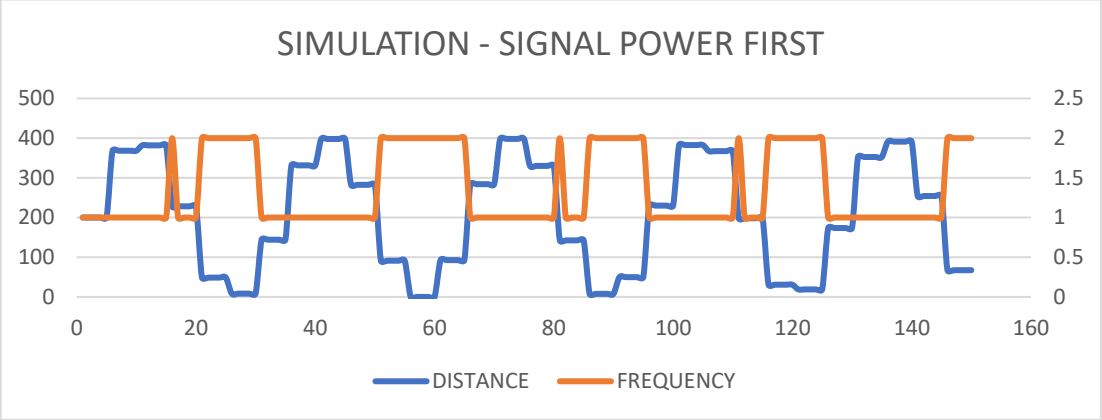

Figure 5.10: Simulation consisting of 150 iterations with the system's controller set with rateStrength=3 and rateBandwidth=2

When the system is configured with a higher priority over the bandwidth, the simulation runs as follows:

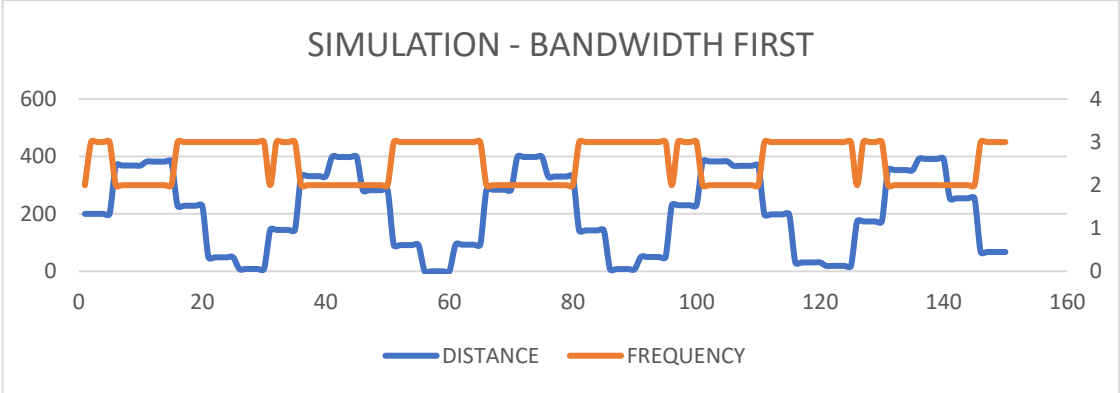

Figure 5.11: Simulation with 150 system iterations with the controller set with rateStrength=2 and rateBandwidth=3

As noticed in the figures, a higher priority over the signal power leads the system to adopt the lowest communication frequencies allowed, while a higher priority over the signal bandwidth leads the system to use the highest communication frequencies allowed.

5.3 Examples validation

5.3.1 Error quantification and noise introduction

The error function is necessary to verify the stability, accuracy, settling time and robustness properties of the generated systems. This function analyses the current values of the input variables and their difference with their ideal values, associating a cost to each variable.

The error function is calculated as the sum of the differences between the ideal value of each input and its current value, each difference is multiplied by the cost associated to the variable.

$$\boldsymbol{e(k)} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{c_i} * \ (\boldsymbol{s_i} - \ \boldsymbol{u_i(k)})$$

Where $\boldsymbol{c_i}$ is the cost associated to each input variable, $\boldsymbol{u_i(k)}$ is the current input variable value and $\boldsymbol{s_i}$ its ideal value.

To controller reconfiguration (performed by modifying the rating values), will cause the cost function to change as well, since the desired states are changing with the controller reconfiguration.

5.3.2 Airplane with variable geometry wings example validation

In this example the configuration of the cost functions are two, as many as the possible configurations: one with priority over fuel saving and the other with priority over manoeuvrability.

When the controller is configured to privilege the fuel saving, the error will be as high as the wing size, since the ideal configuration is the one saving fuel, which is the one with the lowest possible wing size.

On the contrary, when the controller will place higher priority on the manoeuvrability, the error will be as high as low the wing size is, because the ideal configuration is the one allowing the highest degree of manoeuvrability, which is the one having the largest wing size.

The following figures are describing the error function in both the configurations:

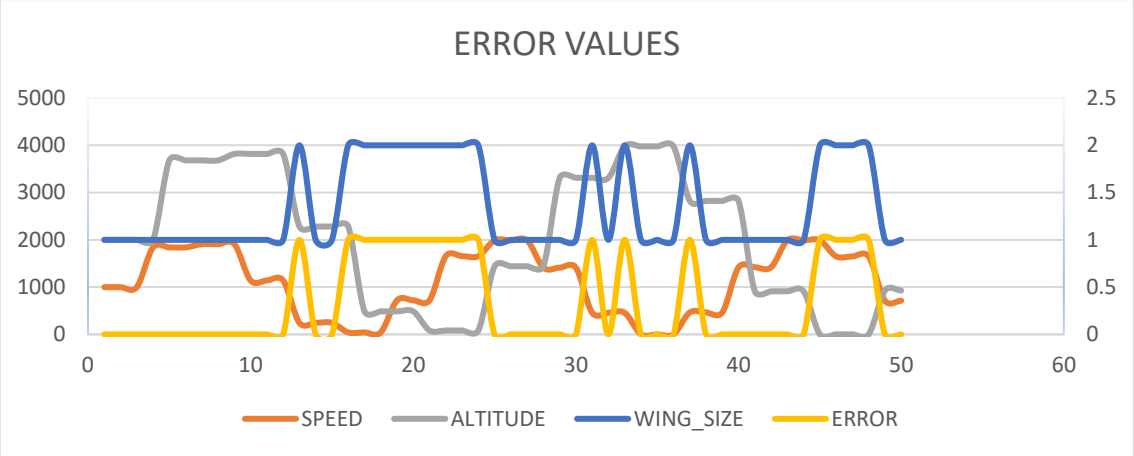

Figure 5.12: Error function for the configuration oriented to the fuel saving (rateFuel=3 and rateManouver=2)

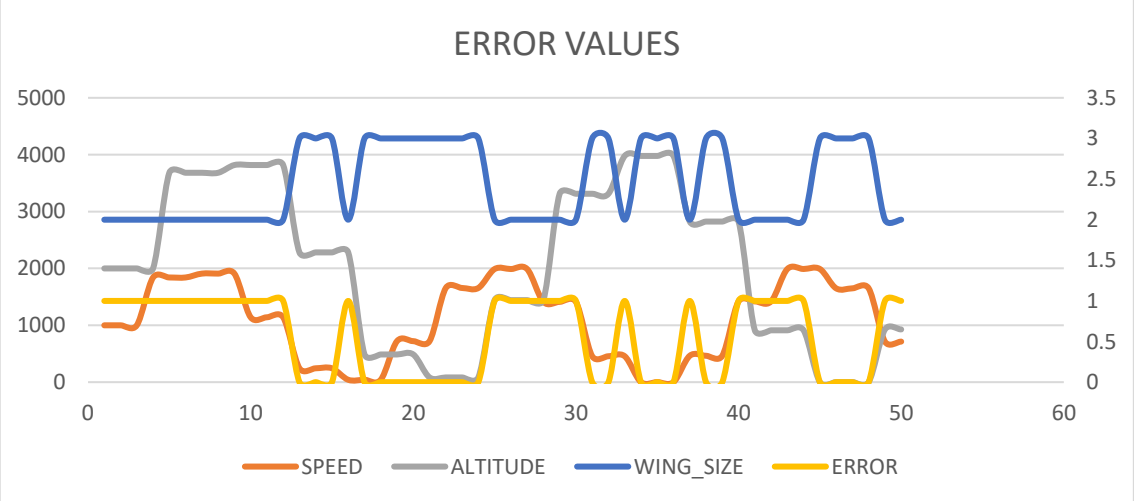

Figure 5.13: Error function for the configuration oriented to the manoeuvrability (rateFuel=2 and rateManouver=3)

5.3.3 Content Distribution Manager example validation

In this example, as in the previous one, there are two cost functions: the first for the resource sparing and the second privileging the quality of the distributed content.

When the system is configured to privilege the quality, the error function will be as high as the quality of the distributed content is low, because in the desired state for this configuration is the one in which the quality of the distributed content is the highest possible.

When the system is configured to spare the available resources, the error function is proportional to the number of servers used, because in the desired state deriving from this configuration the system is using only one server to distribute the content, which is the minimum necessary amount.

The following graphics are showing the error function behaviour without external interferences, the system consequently adapts in a small number of steps:

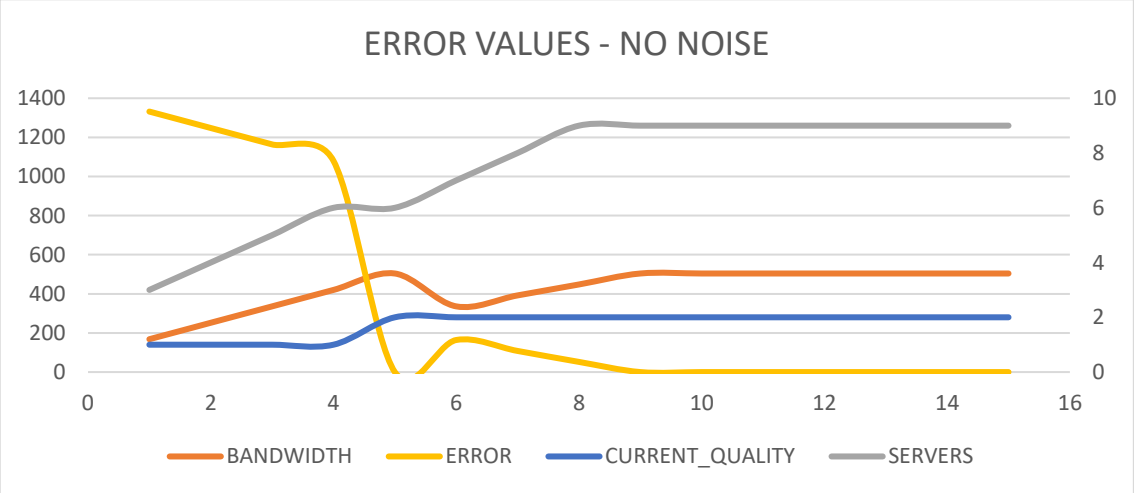

Figure 5.14: Error function in the quality-oriented configuration (rateResources=2 and rateQuality=3) it is noticeable how, for the system to set, the number of allocated server increases, the quality increases and the required bandwidth is reached

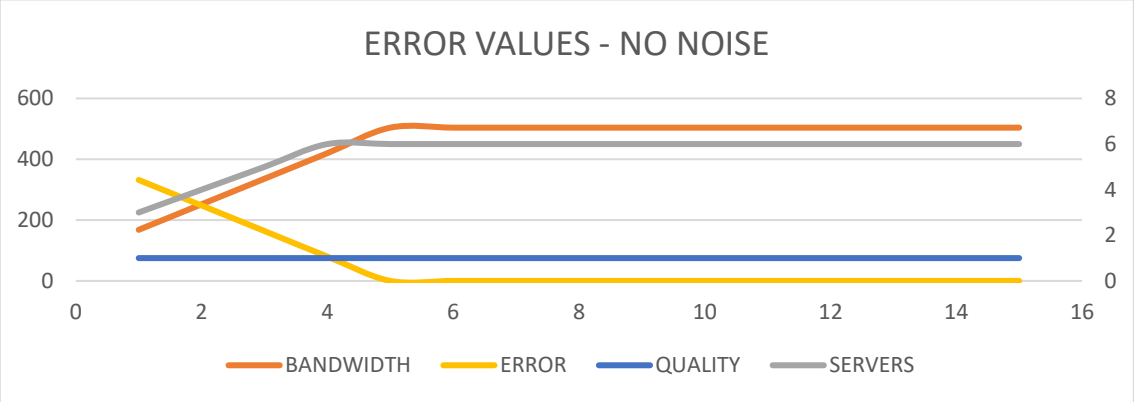

Figure 5.15: Error function in the resource saving configuration (rateResources=3 and rateQuality=2). The quality is always as low as possible and during the settling, to reach the required bandwidth the number of servers allocated is increased.

Although, the bandwidth is only partially controllable by the system, which can increase it increasing the number of allocated servers or reducing the quality of the distributed content. The system is not able in any case to set the available bandwidth autonomously, because it depends on the network availability. To account this consideration, the environment interference is accounted in the simulation, dividing the input in two parts:

$$\boldsymbol{u(k) = i(k) + r(k)}$$

Where **i(k)** represents the part of input controllable by the system (number of allocated server and content quality) and **r(k)** represents the noise, which is the undesired input coming from the network, which causes the fluctuation of the available bandwidth.

With the noise introduction, used to verify the system's robustness, the system's behaviour is the following:

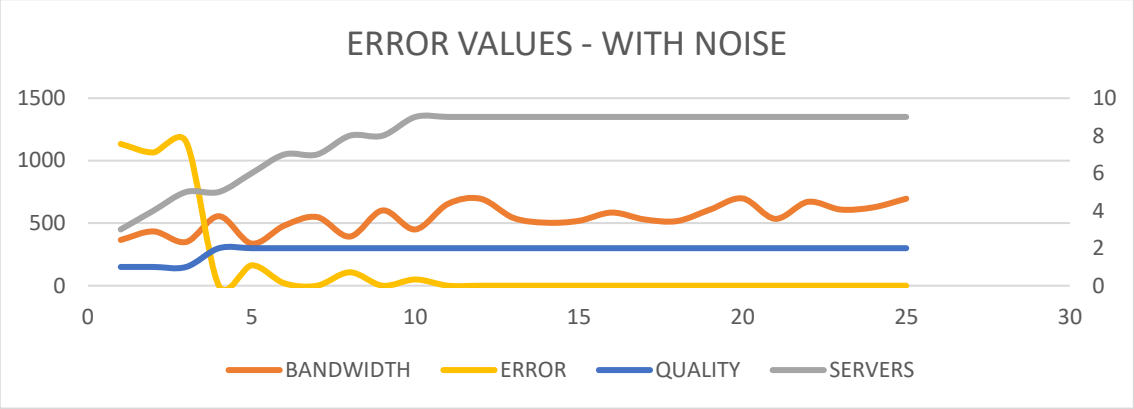

Figure 5.16: Error function where the noise is introduced, and the system is oriented on quality preservation. When the bandwidth reduces because of the environment, more servers are used.

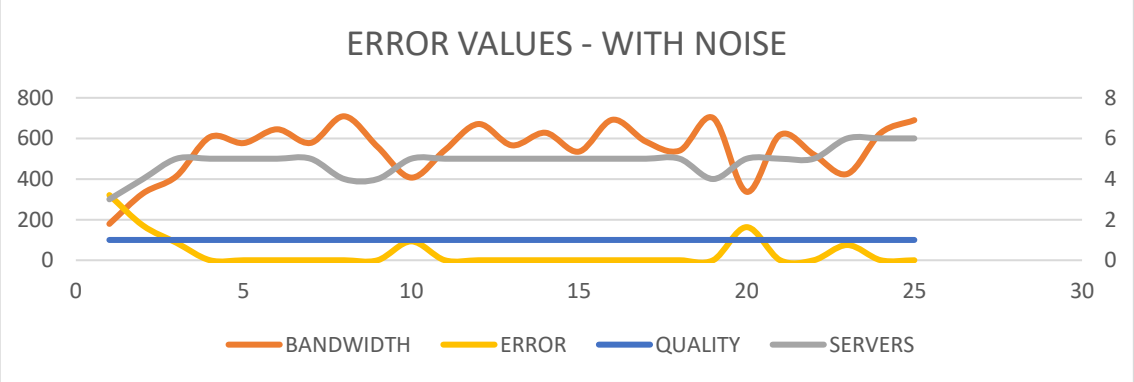

Figure 5.17: Error function when the noise is introduced, and the system is oriented on resource saving. When the available bandwidth decreases because of the environment, other servers are added and when the bandwidth returns to an acceptable level, some servers are deallocated.

5.3.4 Video encoder example validation

In this example, the cost function is calculated in three different ways, which are corresponding to the three possible controller configurations.

The first configuration will privilege the fluidity, so it will consider the difference between the current fps and the desired fps value. If the current fps value is higher than the desired one, the error is zero because the desired system state corresponding to the fluidity goal is already reached.

The second configuration is oriented to the image quality and so the error will depend on the difference between 1 and the current SSIM value.

The third configuration is oriented to the noise reduction and in this case the error will be depending on the difference between 1 and the current SSIM as well.

Without external noise, the system's behaviour is presented in the following figures, which shows that the system is already settling in the firsts iterations:

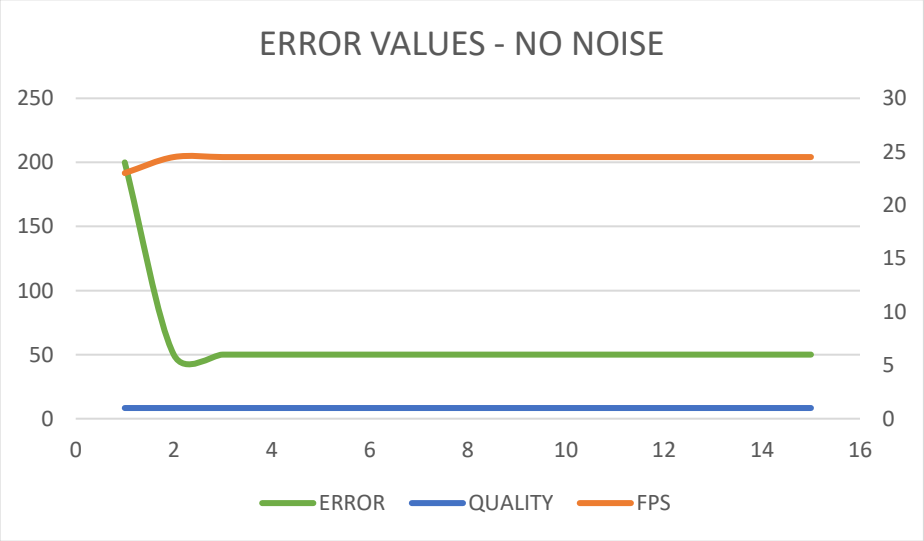

Figure 5.18: Error function values in the fluidity-oriented configuration (rateFluidity=3, rateQuality=1 and rateNoiseReduction=1), the error decreases when the fps value required by the system's desired state is reached.

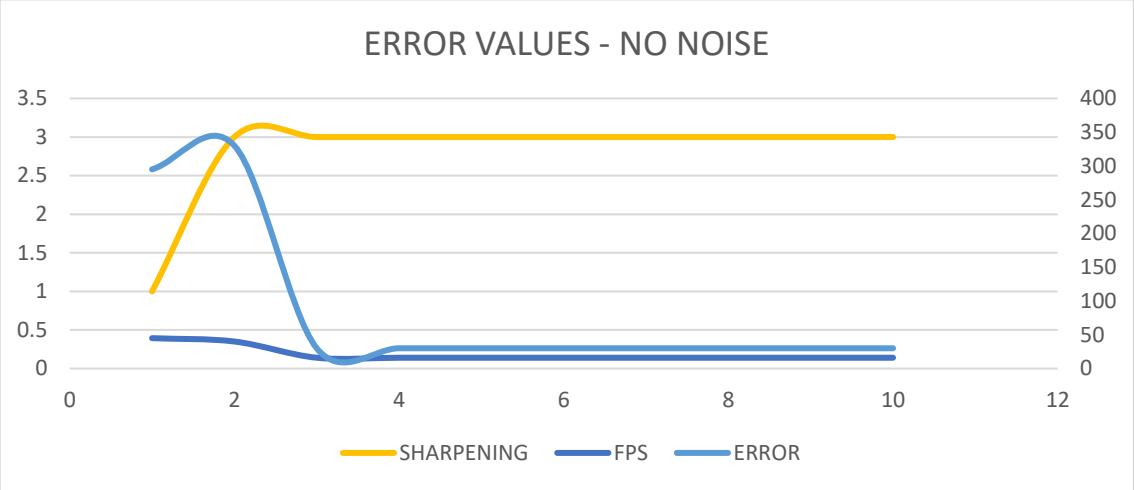

Figure 5.19: Error values in the quality-oriented configuration (rateFluidity=1, rateQuality=3, rateNoiseReduction=1). The error decreases until the desired system configuration is reached. The fps reduction consequent to the encoding quality increase is noticeable as well.

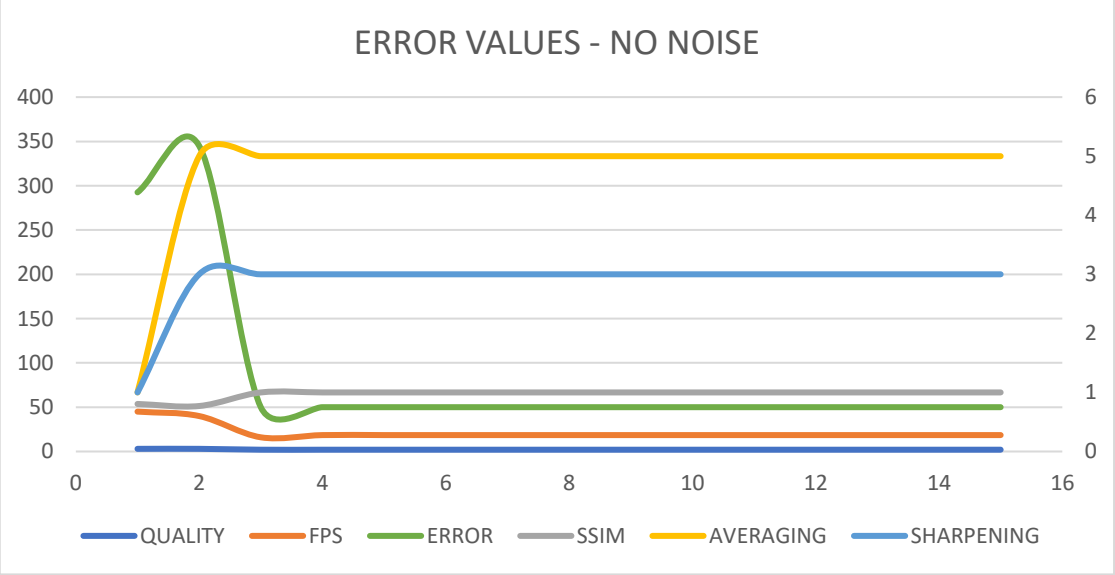

Figure 5.20: Error values in the configuration oriented to the noise reduction (rateFluidity=1, rateQuality=3, rateNoiseRedution=3), the error lowers reaching the image quality required by the configuration. In this case, as in the previous figure, the fps decreases when the filter size and the quality are increased.

As in the case of the previous example, it is necessary to consider the environment intervention in this example as well. It is not possible to predict the kind of images that are to be processed. For instance, the images with many edges and rich of colours are reacting to the encoding algorithm in a more negative way than the simpler images, therefore the SSIM is influenced by the structure of each image, which is an unpredictable element.

As seen in the previous example, the SSIM values are depending on a controllable component (the filter sizes and the encoding algorithm quality) and a non-controllable component (the structure of the processed frames). This last unpredictable contribution was modelled with a random contribution to the SSIM values.

The same thing happens with the fps, which can variate unpredictably because of the resources allocated by the computer to the encoding program. The fps value in the system will be a contribution of controllable factors (the encoding quality and the filter sizes) and uncontrollable factors (resources allocated by the computer to the encoding program).

The system's behaviour when the external noise is introduced is presented in the following figures, it is noticeable how the error function values remain bounded:

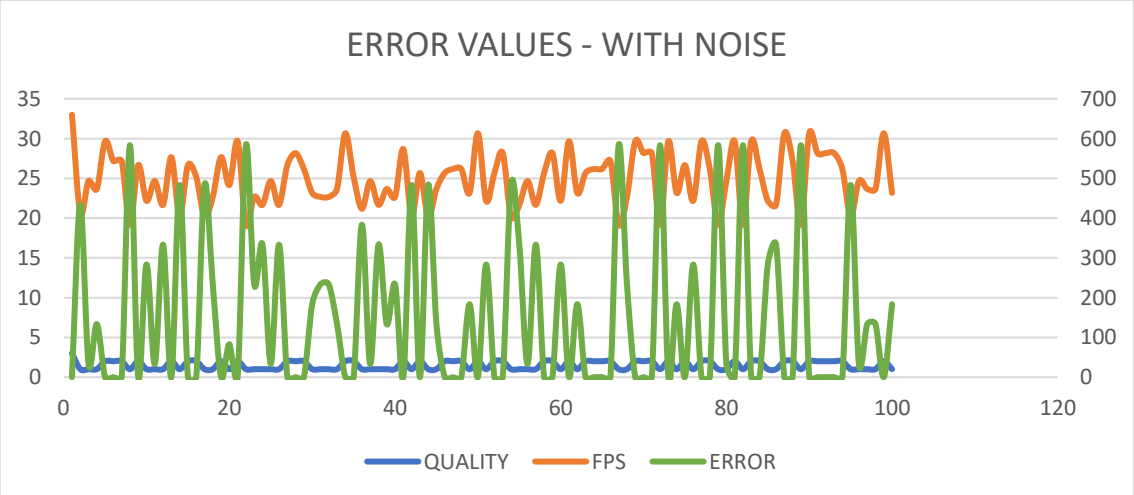

Figure 5.21: Error function in the fluidity-oriented configuration (rateFluidity=3, rateQuality=1, rateNoiseReduction=1), when the fps is lowered because of external factors, the system sacrifices the quality to preserve the fps, lowering the error value in the next steps.

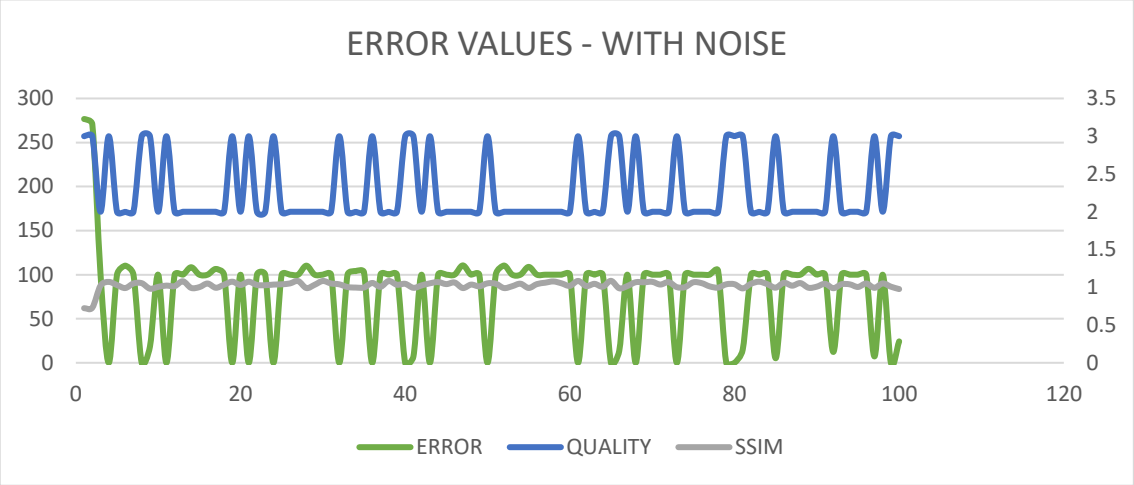

Figure 5.22: Error function values assumed in the quality-oriented configuration (rateFluidity=1, rateQuality=3, rateNoiseReduction=1), when the SSIM decreases the system compensates increasing the quality of the encoding algorithm.

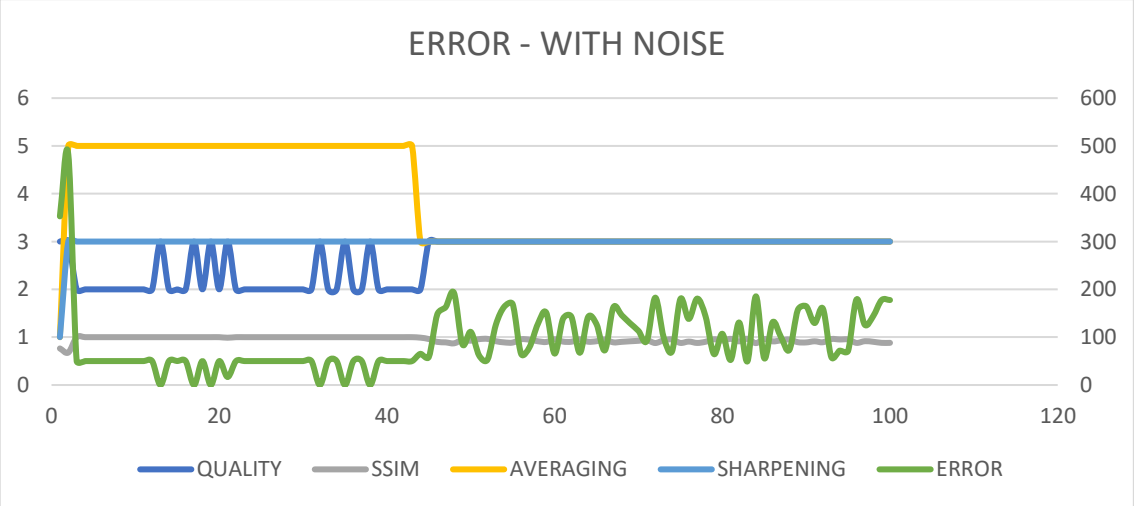

Figure 5.23: Error values assumed by the configuration oriented to noise reduction (rateFluidity=1, rateQuality=3, rateNoiseReduction=3), the SSIM reduction caused by the external factors is compensated increasing the encoding quality and the filter sizes.

5.3.5 Variable frequency wireless transmitter example validation

In this example the error function is calculated in two ways. In case the controller is privileging the signal power, the error is the difference between the used frequency and the minimal one. When the controller is configured to privilege the bandwidth, the error is the difference between the highest communication frequency and the current one.

A sinusoid is used as test input, evaluating the system's behaviour when the distance gradually increases or decreases.

The system's behaviour during the system's settling and the error function values are shown in the following figures:

Figure 5.24: Error values in the signal power-oriented configuration (rateStrength=3, rateBandwidth=2)

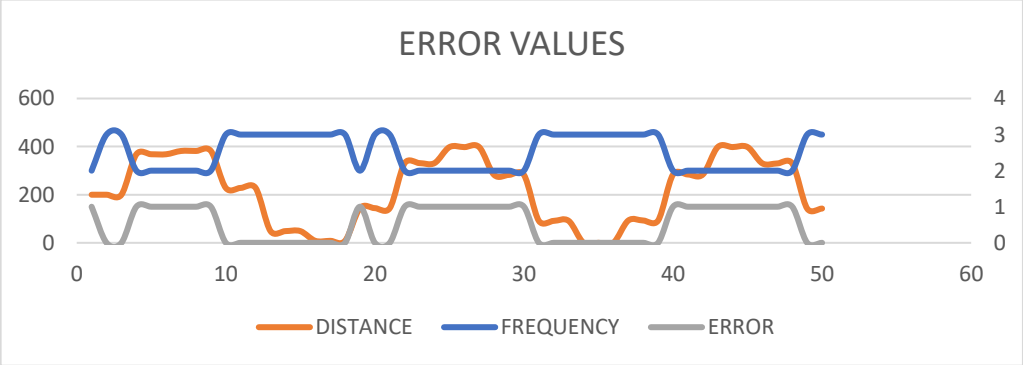

Figure 5.25: Error values in the bandwidth-oriented configuration (rateStrength=2, rateBandwidth=3)

5.4 Controller behaviour during configuration changes

5.4.1 MIRC and MRAC controllers

As [78] states, Self-Adaptive Systems can have a controller able to change its configuration at run-time, consequently changing the desired goals and the system's adaptation policies. Particularly, there are two kinds of controllers that can undergo a run-time reconfiguration: MIAC (Model Identification Adaptive Control) and MRAC (Model Reference Adaptive Control) controllers.

The MIAC controllers identify the model to use to represent the system at run-time and periodically tune the controller. The MRAC controllers on the other hand, use a set of input parameters for their configuration.

The framework designed for this thesis produces Self-Adaptive Systems with MRAC controllers, where the configuration of the parameters takes place tuning the rating values in the input JSON object, where the ratings can be modified at run-time.

To extensively test the controller's quality, it is necessary to simulate situations in which the controller is reconfigured at run-time, studying the error function behaviour and verifying whether there are divergencies or not when the system undergoes to a reconfiguration. The requirements of stability, accuracy and robustness must be preserved even when the controller configuration is changed at run-time.

The **_RTM** object, which was already used to produce the simulations previously presented, was modified to calculate the error function depending on the changing configuration assumed by the controller.

Each simulation consists in 150 iterations, each 50 iterations the system's controller configuration is changed by modifying the rating values in the input. In the examples related to the Content Distribution Manager and the Video encoder, external noise has been introduced, to check the systems robustness during the controller reconfiguration.

5.4.2 Airplane with variable geometry wings controller

In the Self-Adaptive system controlling the wings width of a variable geometry plane, the system's behaviour when the configuration is changed is analysed starting from the fuel saving configuration to go to the manoeuvrability oriented one and returning to the fuel saving configuration:

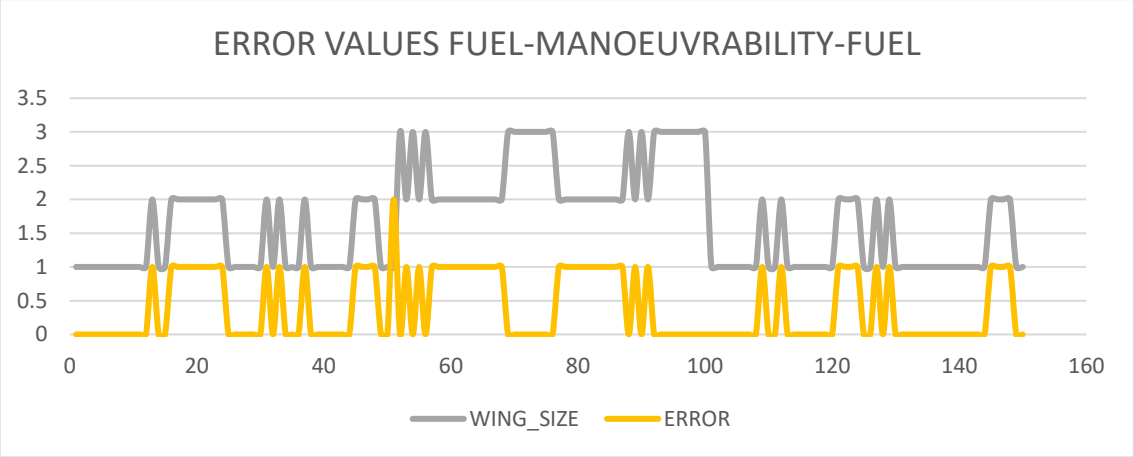

Figure 5.26: Error values at the configuration changes in the airplane with variable geometry wings example

5.4.3 Content Distribution Manager controller

In the example related to the Content Distribution Manager, the starting configuration is the one oriented to resource saving, which changes to the quality oriented one and returns to the resource saving configuration:

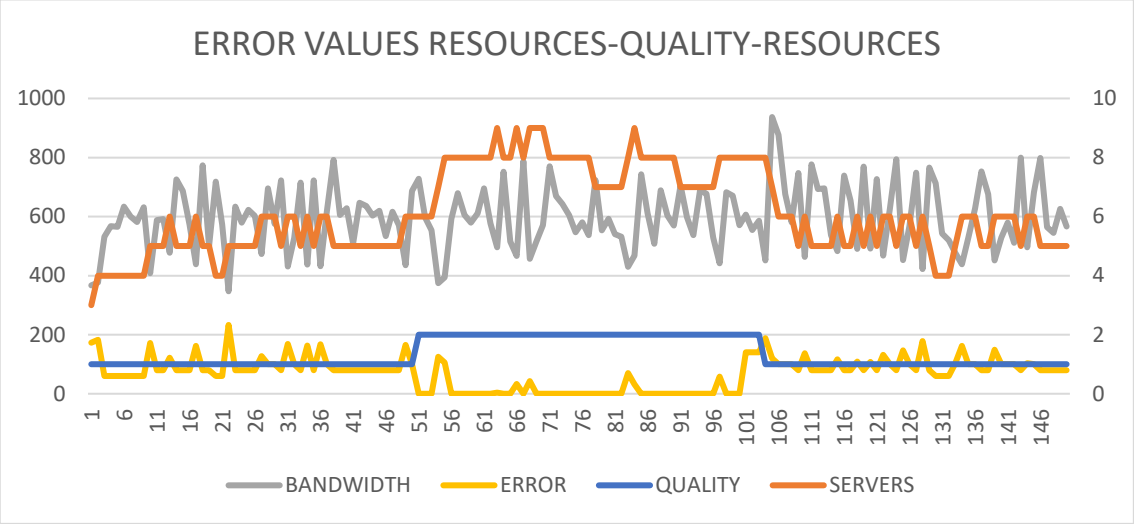

Figure 5.27: Error values in the Self-Adaptive System managing a Content Distribution service, it is visible how the quality of the content increases in the central part of the simulation, when the system is configured to preserve the distributed content quality.

5.4.4 Video encoder controller

In the Self-Adaptive System having the task to manage a Video encoder, there are three possible controller configurations: fluidity oriented, quality oriented and noise reduction oriented:

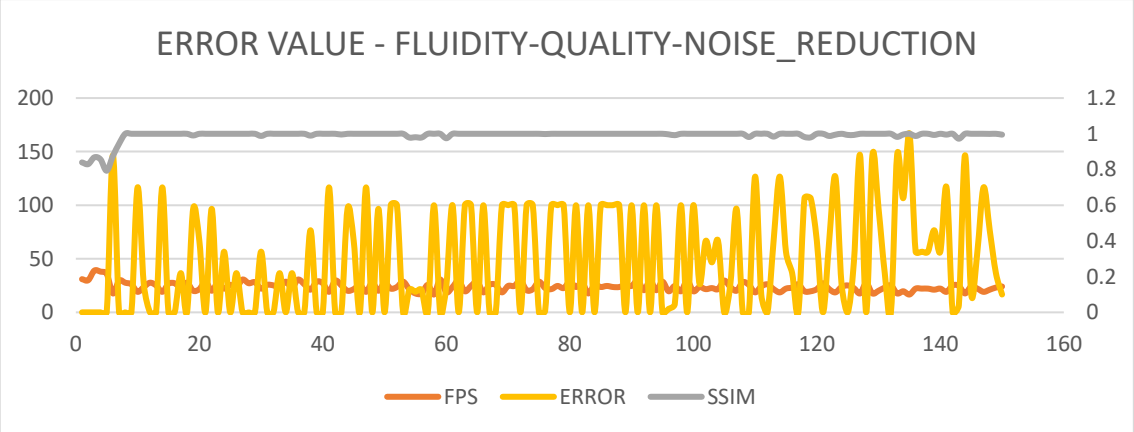

Figure 5.28: Error values in the Self-Adaptive System controlling a video encoder

5.4.5 Variable frequency wireless transmitter controller

In the last of the Self-Adaptive System generated with the framework, which task is to manage the communication frequency between a transmitter and a receiver, the simulation starts from the signal power-oriented configuration to go to the bandwidth oriented one and returning to the signal power-oriented configuration:

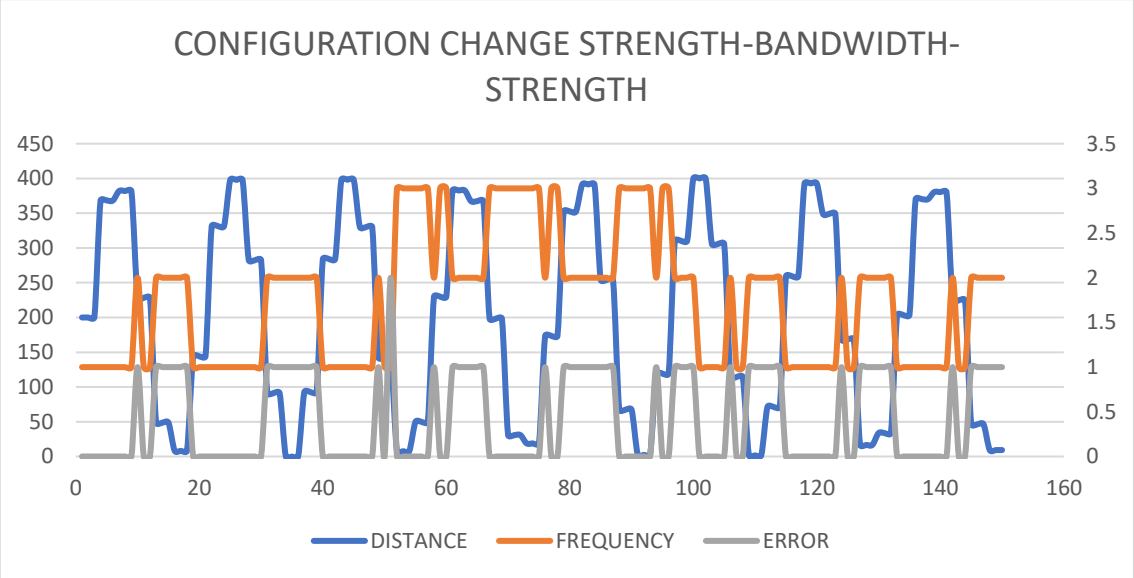

Figure 5.29: Error values in the Self-Adaptive System managing the communication frequency. The test input is displayed as well, which is a biased sinusoid representing the variable distance between receiver and transmitter.

5.5 Quantitative verifications on the implemented Self-Adaptive Systems

5.5.1 Quantitative metrics determination

In the previous paragraphs, using qualitative verifications, it has been demonstrated that the systems generated with the tool are able to achieve the designed goals even with external noise and controller reconfiguration; empirically demonstrating the properties of robustness and stability of such systems. In this paragraph is described which quantitative metrics will be used and how they will be applied to the generated systems.

As [79] states, the reliability of a Self-Adaptive System can be expressed using the reachability of a success state (or complementary of a fail one) associated with a probability index.

In particular, the quantitative metrics that will be used to evaluate the generated Self-Adaptive systems in this thesis, will aim to estimate in how much time, expressed by the number of required steps since the systems are time discrete, the system is able to reduce the value of its error function to a tolerable amount defined as **ε.**

Given the intervention of the external noise, the number of steps in which the system can reach its optimal configuration (within the error margin **ε**) is not constant, but depends on the external noise, which is an aleatory and uncontrollable variable. Since the presence of noise in the models related to the Content Distribution Manager and the Video encoder, these systems have been chosen to perform the quantitative analysis.

A huge number of simulation (a thousand) was performed to estimate the settling time probability distribution function.

Since an adaptive controller is used, each Self-Adaptive System analysed is divided in as many subsystems as the possible configurations it can have. Each subsystem composing the Content Distribution manager and the Video Encoder will be subject to the quantitative verifications.

Defined k as the number of paces necessary for the system to reach its desired configuration, it is possible to estimate the value of **α** associated to k. The variable **α** is a real number between 0 and 1 and indicates the speed of the system's convergence and is defined as the ratio between the current error and the one in the previous discrete step.

The error at the pace k, defined as e(k) is equal to:

$$\boldsymbol{e(k) = \alpha^k e(0)}$$

The system reaches the required state when **e(k) = ε**. Knowing the values of k it is possible to define the values of **α** using the following formula:

$$\boldsymbol{\alpha = \sqrt[k]{\frac{\varepsilon}{e(0)}}}$$

To determine the estimated values of **α**, the Normalized Gaussian Distribution function was used, choosing the k probability distribution function values associated to the following confidence intervals:

- **66,3% , for σ**
- **95,5%, for 2σ**
- **99,7%, for 3σ**

For each analysed subsystem, the values of **α** corresponding to the three confidence intervals are determined.

5.5.2 Content Distribution Manager performances

In the system defined by the controller configured for resource saving, the error value associated to the initial condition is 900 and the **ε** value is set to 100, the probability distribution function defining the number k of paces necessary for the system setting is defined in the following figure:

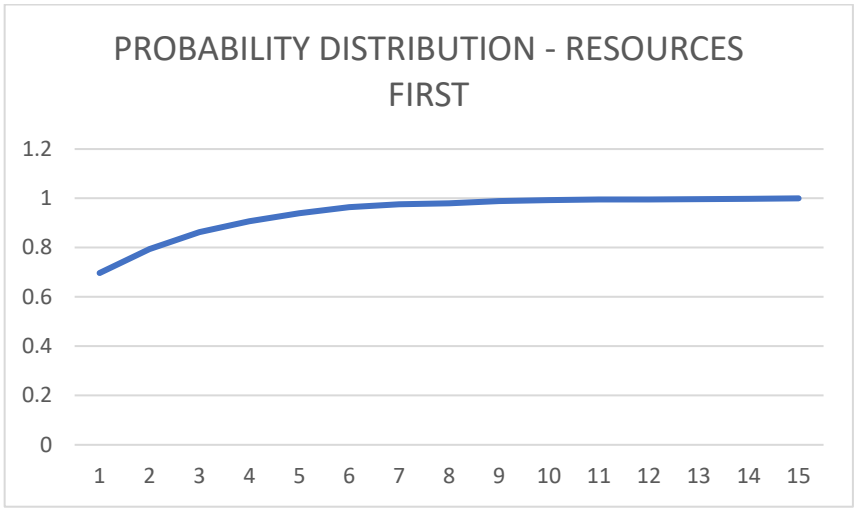

Figure 5.30: Probability distribution function of the Content Distribution manager configured for resource sparing

The values of k obtained from the 1000 iterations are shown in the following histogram:

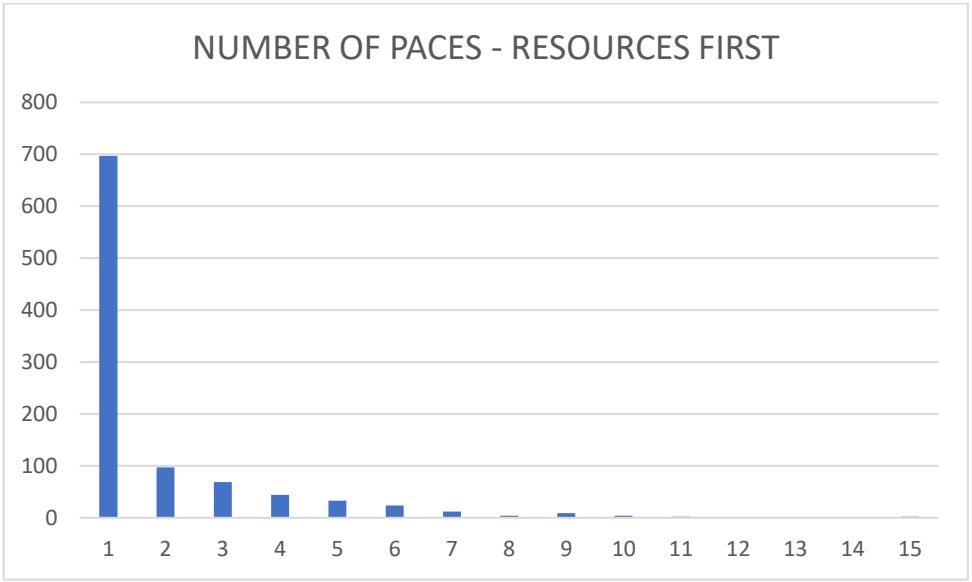

Figure 5.31: Histogram of the k values for the Content Distribution manager configured for resource sparing

The k values associated to the confidence intervals are the following:

- **k=1 for σ**
- **k=6 for 2σ**
- **k=11 for 3σ**

From which the corresponding values of **α** are calculated:

$$\boldsymbol{\alpha_1 = 0,11;\ \alpha_2 = 0,69;\ \alpha_3 = 0,81;}$$

In the subsystem configured for the preservation of the distributed content quality, the error in the initial state is 420 and **ε** is still set to 100. The probability distribution function related to the number k of paces necessary for the system setting is defined in the following figure:

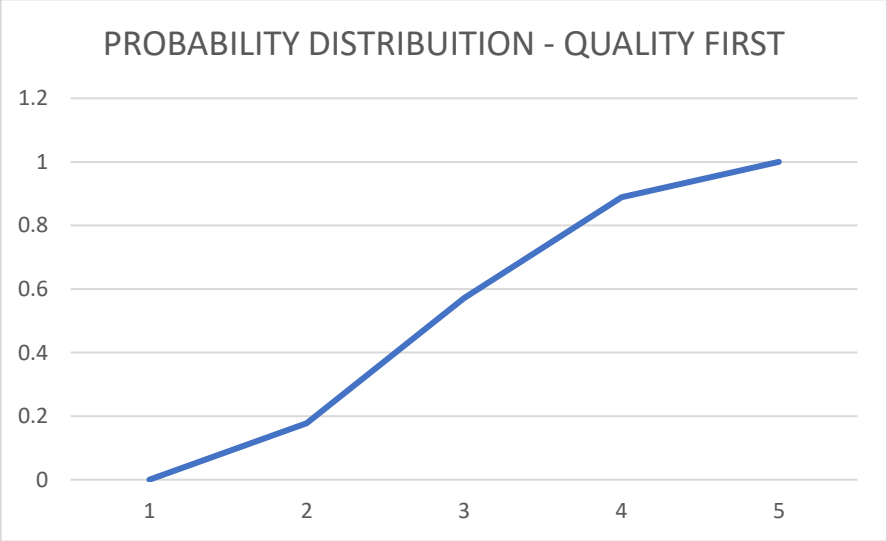

Figure 5.32: Probability distribution function of the Content Distribution manager configured for quality preservation

The values of k obtained from the 1000 iterations are shown in the following histogram:

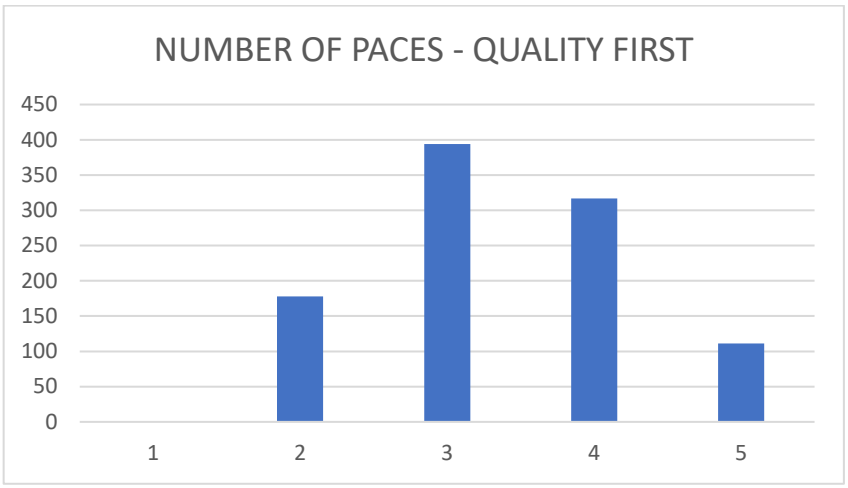

Figure 5.33: Histogram of the k values for the Content Distribution manager configured for quality preservation

The k values associated to the confidence intervals are the following:

- **k= 4 for σ**
- **k= 5 for 2σ and 3σ**

From which the corresponding values of **α** are calculated:

$$\boldsymbol{\alpha_1 = 0,70;\ \alpha_2 = \alpha_3 = 0,75;}$$

As noticed by the obtained values of **α**, the settling time variability of the resource sparing subsystem is higher than the one of the quality-preserving subsystem, since in the quality preserving subsystem the difference between the highest and lowest values of **α** is smaller.

5.5.3 Video encoder performances

This system is divided in three subsystems, the first subsystem for the fluidity-oriented configuration, the second subsystem for the quality-oriented configuration and the third subsystem for the noise reduction configuration. In all three subsystems the **ε** value is set to 100.

In the fluidity-oriented subsystem, the initial error is 300 and the distribution function related to the number of steps necessary for the subsystem to settle is shown in the following figure:

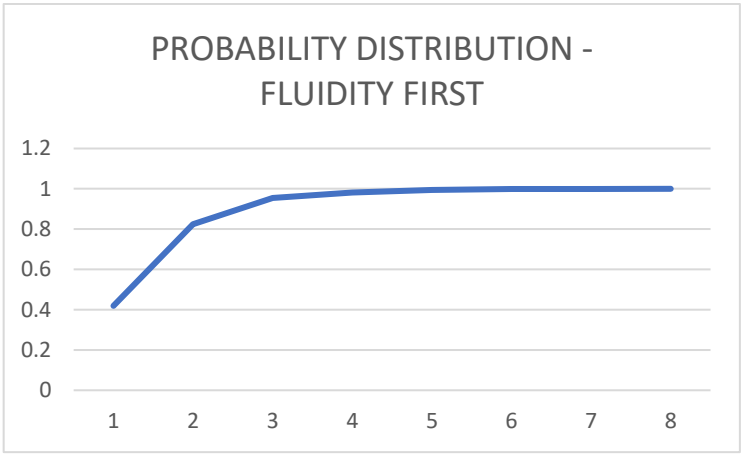

Figure 5.34: Probability distribution function of the video encoder configured for fluidity preservation

The values of k are shown in the following histogram:

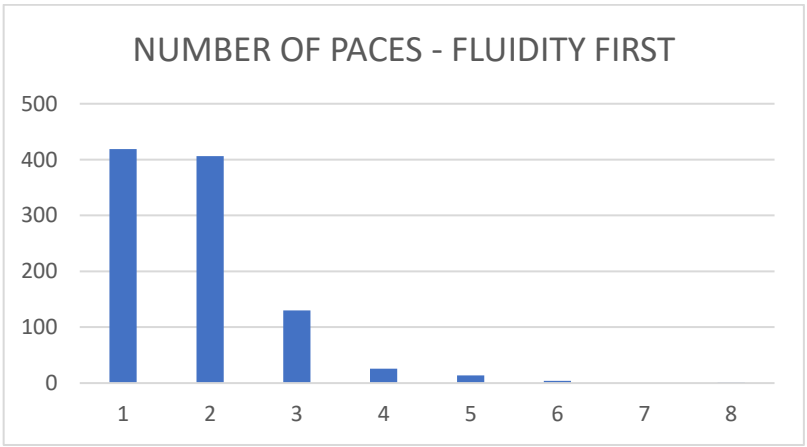

Figure 5.35: k values histogram for the video encoder configured for fluidity preservation

The k values associated to the confidence intervals are the following:

- **k= 2 for σ**
- **k= 3 for 2σ**
- **k= 6 for 3σ**

From which the corresponding values of **α** are calculated:

$$\boldsymbol{\alpha_1 = 0,58;\ \alpha_2 = 0,69;\ \alpha_3 = 0,83;}$$

In the quality-oriented subsystem, the initial error is 250 and the probability distribution function is shown in the following figure:

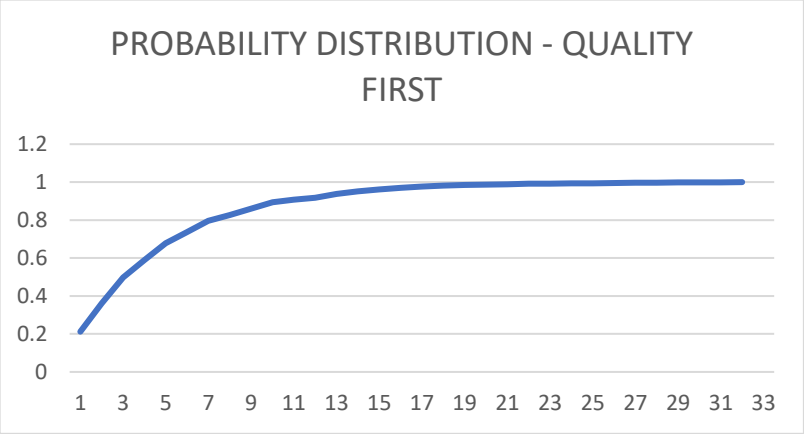

Figure 5.36: Probability distribution function of the video encoder configured for quality preservation

The values of k are shown in the following histogram:

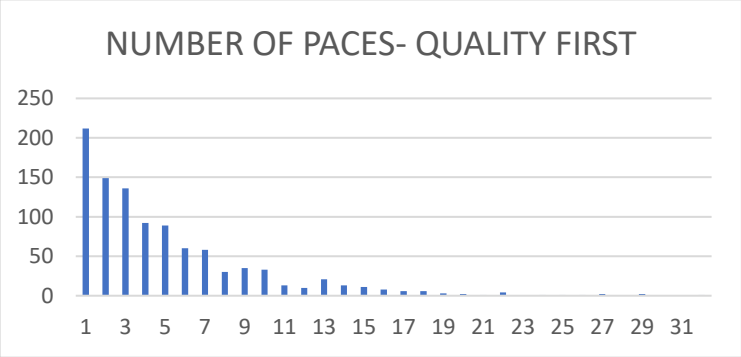

Figure 5.37: k values histogram for the video encoder configured for quality preservation

The k values associated to the confidence intervals are the following:

- **k= 5 for σ**
- **k= 15 for 2σ**
- **k= 27 for 3σ**

From which the corresponding values of **α** are calculated:

$$\boldsymbol{\alpha_1 = 0,83;\ \alpha_2 = 0,94;\ \alpha_3 = 0,97;}$$

In the subsystem oriented to noise reduction, the initial error is 250 and the probability distribution function is shown in the following figure:

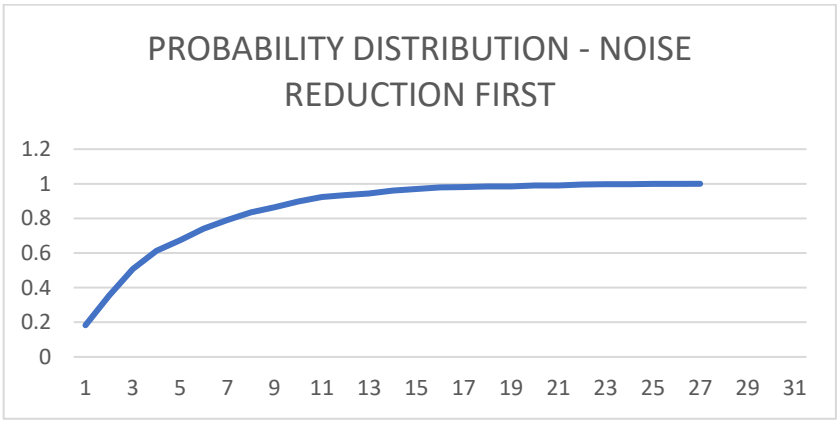

Figure 5.38: Probability distribution function of the video encoder configured for noise reduction

The values of k are shown in the following histogram:

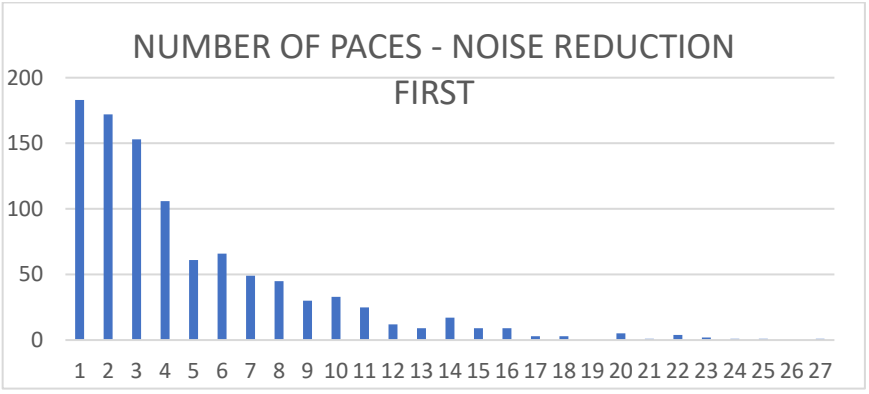

Figure 5.39: k values histogram for the video encoder configured for noise reduction

The k values associated to the confidence intervals are the following:

- **k= 5 for σ**
- **k= 16 for 2σ**
- **k= 23 for 3σ**

From which the corresponding values of **α** are calculated:

$$\boldsymbol{\alpha_1 = 0,83;\ \alpha_2 = 0,94;\ \alpha_3 = 0,96;}$$

From the **α** values that were obtained, it is evident that the fluidity-oriented subsystem is faster than the other two subsystems, which have higher values of **α.**

5.5.4 Systems performances during the controller reconfiguration

In this paragraph an estimation is done over the number of steps necessary for the controller reconfiguration of the Content Distribution Manager and video encoder Self-Adaptive systems.

Through the simulations, the system is first stabilized with a first configuration and then such configuration is modified changing the rating values at run-time.

The previous analysis gave the number of steps that are necessary for each subsystem to settle, it was chosen to let the subsystems settle with the first configuration before changing it. The number of steps that was used to let the system stabilize with the first configuration was the highest found in the k probability distribution function, to which a further step was added as caution measure, going beyond three confidence intervals.

All the possible configuration changes in the two systems have been tested, the following tables are showing the obtained results:

Server manager

Initial configuration	Final configuration	K for the first settling	K for σ (66,3%)	K for 2σ (95,5%)	K for 3σ (99,7%)
QUALITY	RESOURCES	6	1	1	8
RESOURCES	QUALITY	16	5	27	62

Video Encoder

Initial configuration	Final configuration	K for the first settling	K for σ (66,3%)	K for 2σ (95,5%)	K for 3σ (99,7%)
QUALITY	FLUIDITY	33	1	7	18
FLUIDITY	QUALITY	9	2	2	24
QUALITY	NOISE	33	2	2	2
NOISE	QUALITY	28	3	3	3
FLUIDITY	NOISE	9	2	4	5
NOISE	FLUIDITY	28	2	2	2

5.5.5 Applications of the calculated metrics

The k values calculated for the quantitative analysis of the generated systems allow to quantify the system readiness, defining a probability distribution function indicating in how much time the system and with which probability the system will settle.

Knowing the probability distribution function related to the settling time k can also help to choose the correct parameters to design the control loop's components.

The **α** values are a quantitative measure through which the different subsystems of a Self-Adaptive System can be compared, allowing to understand which adaptation procedures are more expensive in terms of time.

The comparison between the **α** values of different possible implementations of the same subsystem allows to ease the decisions made at design time, helping to choose the best implementation between different possibilities.

The three values of **α**, particularly the highest and the lowest ones, allow to quantify the settling time variability, helping to quickly understand if a system has a high variance in its response time or if the response time is mostly constant.

The knowledge of the settling time required to change the system configuration, switching from one subsystem to another, is very important in the design of a SASS system because gives the possibility to tell which changes of configurations in the systems take more time and defining the system's Self-Adaptation time consequently.

6. Conclusions and future works

6.1 Conclusions

In this thesis were described the design and implementation of a framework to assist the creation of Self-Adaptive Systems. The framework was implemented using the PHP, Java, SQL, Html, CSS and JavaScript languages. The Integrated Development Environments used were Eclipse and NetBeans.

The architectural model used to design and implement the framework follows the architectural paradigm MAPE-K. The units of Monitor and Analyser have been implemented, with the elements of the Knowledge Base necessary for their functioning.

After the framework implementation, an analysis over the created examples behaviour was performed to verify the quality of the systems generated with the framework assistance. The metrics used to evaluate the quality of the generated systems are the ones belonging to the control theory, which are stability, accuracy, settling time, overshoot, robustness and adaptability.

After the qualitative verifications, quantitative values were determined to certify the system's reliability. These quantitative values can be useful to assist in the Planner and Executor units design, allowing to evaluate the system's overall performances.

The systems generated through the framework have been analysed with simulations, in the first stage with the testing of the single elements of the system's architecture, determining if their behaviour is coherent with the designed goals (One-way testing procedures where the oracle is represented by the ideal state, having an error value of zero). In the second testing stage, consisting of the In-The-Loop testing procedure, the other components of the control ring were simulated using the Run Time Model Simulations, implemented with dedicated java classes.

The analysed systems, as shown by the testing procedures performed, responded positively in relation to the metrics used for the quality control, proving to be stable and robust and being able to satisfy the system's requirements in spite the influence of the external environment.

6.2 Future works

The future works related to the implemented project are mainly focused on the completion of the Planning and Executor components and in the knowledge base expansion, introducing the elements necessaries for the correct functioning of this two units.

The graphic interface, designed as a web-based application, can be perfected and widened in a way such as the data input is easier. Another improvement of the web interface would be the possibility to enter the system's condition propositions in the interface itself instead of using an external editor.

In the future, once all the system's components have been implemented, the systems generated by the tool can be connected to existing software, allowing to test the systems in their interaction with the real environment, as defined in the On-Line testing procedure.

Another possibility for further development of the tool is to use machine-learning techniques to automatically generate the conditions from an available data set, enriching the content of the ontologies generated by the tool with the results of different machine learning algorithms.

Since they are software systems, the Self-Adaptive systems generated through the implemented framework have other requirements than those of the classic control theory. It will be necessary to test qualitative requirements related to the software such as overall performance, reliability, security and confidentiality of the data provided to the system. Confidentiality can be a critical aspect since its implications of legal and ethical nature.

The framework here implemented can be expanded, giving it the capability to generate complex multi-agent systems, in which more than one entity is managed. This expansion can be performed moving from a model where only one entity, with its input-output variable is

monitored and controlled, to a model where many different entities, with their variables and their requirements are interacting with each other.

Since the Self-Adaptive Systems are currently a matter of study in many fields, the possible applications of the systems generated with the implemented tool are plenty, spacing from the simple management of already existing software systems to more complex fields such as robotics, e-health or the generation of decision support systems.

7. ANNEX: _RTM classes implementing the simulations

7.1 _RTM_geometry_wings.java

```
public class _RTM_geometry_wings {
 int alt;
 String wing;
 double speed;
 double wind;
 String user_id;
 static int time_counter;

 _RTM_geometry_wings(Map<String, Object> userData_input) {
       this.alt=Integer.parseInt((String) userData_input.get("alt"));
     this.wing=(String) userData_input.get("wing");
     this.speed=Double.parseDouble((String) userData_input.get("speed"));
     this.wind=Double.parseDouble((String) userData_input.get("wind"));
     this.user_id=(String) userData_input.get("user_id");
 }

 public void show_system_state() {
       System.out.println("ALTITUDE: " + this.alt);
       System.out.println("WING: " + this.wing);
       System.out.println("SPEED: " + this.speed);
       System.out.println("WIND: " + this.wind);
       System.out.println("TIME: " + time_counter);
 }

 public Map<String,Object> get_system_state() {
       Map<String,Object> current_state=new HashMap<String, Object>();
       current_state.put("alt",String.valueOf(this.alt));
       current_state.put("wing",String.valueOf(this.wing));
       current_state.put("speed",String.valueOf(this.speed));
       current_state.put("wind",String.valueOf(this.wind));
       current_state.put("user_id",String.valueOf(this.user_id));
       return current_state;
 }

 public void execute_goal(String goal) {
       switch(goal) {
       case "set_small_size":
     this.wing="small_wing";
     break;
       case "set_medium_size":
```

```
        this.wing="medium_wing";
        break;
        case "set_large_size":
        this.wing="large_wing";
        break;
        }
 }

 public void time_pass() {
        this.alt=2000+(int) (2000*Math.sin((double)(time_counter/4)));
        this.speed=1000+(int) (1000*Math.sin((double)(time_counter/3)));
        time_counter++;
 }

 public double calculate_error_1(Map<String,Object> new_state) {
        double error=0;
        double wing_error=0;
        String current_quality=(String) new_state.get("wing");
        if(current_quality.equals("medium_wing")) {wing_error=1;}
        if(current_quality.equals("large_wing")) {wing_error=2;}
        error=wing_error;
        return error;
 }

 public double calculate_error_2(Map<String,Object> new_state) {
        double error=0;
        double wing_error=0;
        String current_quality=(String) new_state.get("wing");
        if(current_quality.equals("medium_wing")) {wing_error=1;}
        if(current_quality.equals("small_wing")) {wing_error=2;}
        error=wing_error;
        return error;
 }
}
```

7.2 _RTM_server_manager.java

```
public class _RTM_server_manager {
	String Quality;
	int Pool;
	double Bandwith;
	String user_id;
	static int time_counter;

_RTM_server_manager(Map<String, Object> userData_input) {
		this.Quality=(String) userData_input.get("Quality");
		this.Pool=Integer.parseInt((String) userData_input.get("Pool"));
		this.Bandwith=Double.parseDouble((String)
userData_input.get("Bandwith"));
		this.user_id=(String) userData_input.get("user_id");
	}

	public void show_system_state() {
		System.out.println("BANDWIDTH: " + this.Bandwith);
		System.out.println("QUALITY: " + this.Quality);
		System.out.println("POOL: " + this.Pool);
		System.out.println("TIME: " + time_counter);
	}

	public Map<String,Object> get_system_state() {
		Map<String,Object> current_state=new HashMap<String, Object>();
		current_state.put("Quality",String.valueOf(this.Quality));
		current_state.put("Pool",String.valueOf(this.Pool));
		current_state.put("Bandwith",String.valueOf(this.Bandwith));
		current_state.put("user_id",String.valueOf(this.user_id));
		return current_state;
	}

	public void execute_goal(String goal) {
		switch(goal) {
		case "set_multimedia":
	    this.Quality="High";
	    break;
		case "set_text":
		this.Quality="Low";
		break;
		case "raise_pool":
		this.Pool++;
		break;
		case "lower_pool":
	    if(this.Bandwith>700) {this.Pool--;}
		break;
		}
	}

	public void time_pass() {
```

```
                //bandidth depending on number of servers used AND external
factor (noise)
                Random rand=new Random();
                this.Bandwith=(560*(double) this.Pool/10);
                if(this.Quality.equals("Low")) {this.Bandwith*=1.5;}
                this.Bandwith+= rand.nextInt(300);
                time_counter++;
        }

        public double calculate_error_1(Map<String,Object> new_state) {
                double error=0;
                double bandwidth_error=500-Double.valueOf((String)
new_state.get("Bandwith"));
                if(bandwidth_error<0) {bandwidth_error=0;}
                double server_error=20*(Double.valueOf((String)
new_state.get("Pool"))-1);
                error=bandwidth_error+server_error;
                return error;
        }

        public double calculate_error_2(Map<String,Object> new_state) {
                double error=0;
                double bandwidth_error=500-Double.valueOf((String)
new_state.get("Bandwith"));
                if(bandwidth_error<0) {bandwidth_error=0;}
                double quality_error=0;
                String current_quality=(String) new_state.get("Quality");
                if(current_quality.equals("Low")) {quality_error=500;}
                error=bandwidth_error+quality_error;
                return error;
        }
}
```

7.3 _RTM_video_encoder.java

```
public class _RTM_video_encoder {
 double ssim;
 String current_encoding_quality;
 double filter_radius;
 double sharpening_radius;
 double fps;
 String user_id;
 static int time_counter;

 _RTM_video_encoder(Map<String, Object> userData_input) {
     this.current_encoding_quality=(String)
userData_input.get("current_encoding_quality");
     this.filter_radius=Double.parseDouble((String)
userData_input.get("filter_radius"));
     this.sharpening_radius=Double.parseDouble((String)
userData_input.get("sharpening_radius"));
     this.fps=Double.parseDouble((String) userData_input.get("fps"));
     this.ssim=Double.parseDouble((String) userData_input.get("ssim"));
     this.user_id=(String) userData_input.get("user_id");
 }

 public void show_system_state() {
        System.out.println("CURRENT ENCODING QUALITY: " +
this.current_encoding_quality);
        System.out.println("FILTER RADIUS: " + this.filter_radius);
        System.out.println("SHARPENING RADIUS: " + this.sharpening_radius);
        System.out.println("SSIM: " + this.ssim);
        System.out.println("FPS: " + this.fps);
        System.out.println("TIME: " + time_counter);
 }

 public Map<String,Object> get_system_state() {
        Map<String,Object> current_state=new HashMap<String, Object>();

current_state.put("current_encoding_quality",String.valueOf(this.current_enco
ding_quality));

current_state.put("filter_radius",String.valueOf(this.filter_radius));

current_state.put("sharpening_radius",String.valueOf(this.sharpening_radius))
;
        current_state.put("ssim",String.valueOf(this.ssim));
        current_state.put("fps",String.valueOf(this.fps));
        current_state.put("user_id",String.valueOf(this.user_id));
        return current_state;
 }

 public void execute_goal(String goal) {
        switch(goal) {
```

```
        case "set_average_encoding":
      this.current_encoding_quality="med";
      break;
        case "set_low_encoding":
        this.current_encoding_quality="low";
        break;
        case "set_high_encoding":
        this.current_encoding_quality="high";
        break;
        case "set_average_radius":
        this.filter_radius=3;
        break;
        case "set_low_radius":
      this.filter_radius=2;
      break;
      case "set_high_radius":
      this.filter_radius=5;
      case "set_average_sharpening":
      this.sharpening_radius=3;
      break;
      case "set_low_sharpening":
      this.sharpening_radius=2;
      break;
      case "set_high_sharpening":
      this.sharpening_radius=5;
      break;
        }
 }

 public void time_pass() {
        Random rand=new Random();
        double bonus=0;
        if(this.current_encoding_quality.equals("low")) {bonus=1.5;}
        else if(this.current_encoding_quality.equals("med")) {bonus=1.25;}
        else if(this.current_encoding_quality.equals("high")) {bonus=1;}
        this.fps=2*(5/this.filter_radius)+2*(5/this.filter_radius)+10*bonus;
        this.fps+=rand.nextInt(10);

this.ssim=0.6+0.07*bonus+0.05*this.sharpening_radius+0.05*this.filter_radius;
        this.ssim+=0.1*rand.nextDouble();
        if(this.ssim>1.0) {this.ssim=1;}
        time_counter++;
 }

 public double calculate_error_1(Map<String,Object> new_state) {
        double error=0;
        double ssim_error=1500*(1.0-Double.valueOf((String)
new_state.get("ssim")));
        double fps_error=20*(25.0-Double.valueOf((String)
new_state.get("fps")));
        if(ssim_error<0) {ssim_error=0;}
        if(fps_error<0) {fps_error=0;}
```

```
        double quality_error=0;
        String current_quality=(String)
new_state.get("current_encoding_quality");
        if(current_quality.equals("low")) {quality_error=2;}
        if(current_quality.equals("med")) {quality_error=1;}
        quality_error*=50;
        error=fps_error;
        return error;
 }

 public double calculate_error_2(Map<String,Object> new_state) {
        double error=0;
        double ssim_error=1000*(1.0-Double.valueOf((String)
new_state.get("ssim")));
        double fps_error=100*(25.0-Double.valueOf((String)
new_state.get("fps")));
        if(ssim_error<0) {ssim_error=0;}
        if(fps_error<0) {fps_error=0;}
        double quality_error=0;
        String current_quality=(String)
new_state.get("current_encoding_quality");
        if(current_quality.equals("low")) {quality_error=2;}
        if(current_quality.equals("med")) {quality_error=1;}
        quality_error*=50;
        error=quality_error+ssim_error;
        return error;
 }

}
```

7.4 _RTM_wireless_strength.java

```
public class _RTM_wireless_strength {
	String frequency;
	double distance;
	String user_id;
	static int time_counter;

	_RTM_wireless_strength(Map<String, Object> userData_input) {
		this.frequency=(String) userData_input.get("frequency");
		this.distance=Double.parseDouble((String)
userData_input.get("distance"));
		this.user_id=(String) userData_input.get("user_id");
	}

	public void show_system_state() {
		System.out.println("DISTANCE: " + this.distance);
		System.out.println("FREQUENCY: " + this.frequency);
		System.out.println("TIME: " + time_counter);
	}

	public Map<String,Object> get_system_state() {
		Map<String,Object> current_state=new HashMap<String, Object>();
		current_state.put("distance",String.valueOf(this.distance));
		current_state.put("frequency",String.valueOf(this.frequency));
		current_state.put("user_id",String.valueOf(this.user_id));
		return current_state;
	}

	public void execute_goal(String goal) {
		switch(goal) {
		case "set_high":
	    this.frequency="high";
	    break;
		case "set_med":
		this.frequency="med";
		break;
		case "set_low":
		this.frequency="low";
		break;
		}
	}

	public void time_pass() {
		this.distance=200+200*Math.sin(time_counter/3);
		time_counter++;
	}

	public double calculate_error_1(Map<String,Object> new_state) {
		double error=0;
		double frequency_error=0;
```

```
		String current_frequency=(String) new_state.get("frequency");
		if(current_frequency.equals("med")) {frequency_error=1;}
		if(current_frequency.equals("high")) {frequency_error=2;}
		error=frequency_error;
		return error;
	}

	public double calculate_error_2(Map<String,Object> new_state) {
		double error=0;
		double frequency_error=0;
		String current_frequency=(String) new_state.get("frequency");
		if(current_frequency.equals("med")) {frequency_error=1;}
		if(current_frequency.equals("low")) {frequency_error=2;}
		error=frequency_error;
		return error;
	}
}
```

8. Bibliography and sitography

[1] De Lemos R. et al. (2013) Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, https://software.imdea.org/~alessandra.gorla/papers/deLemos-Roadmap-SEfSAS213.pdf

[2] Cheng B.H.C. et al. (2009) Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng B.H.C., de Lemos R., Giese H., Inverardi P., Magee J. (eds) Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol 5525. Springer, Berlin, Heidelberg

[3] Robert Hirschfeld, Pascal Costanza, Oscar Nierstrasz: "Context-oriented Programming", in Journal of Object Technology, *vol. 7, no. 3, March-April 2008, pp. 125-151, http://www.jot.fm/issues/issue_2008_03/article4/*

*[4] P*eyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.4060&rep=rep1&type=pdf

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming, https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[6] Irwin J., Loingtier J.-M., et al., Aspect-Oriented Programming of Sparse Matrix Code, Xerox PARC, Palo Alto, CA. Technical report SPL97-007 P9710045, February, 1997

[7] Mendhekar A., Kiczales G., et al., RG: A Case-Study for Aspect-Oriented Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February, 1997.

[8] Lopes C. V. and Kiczales G., D: A Language Framework for Distributed Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, February, 1997.

[9] G. Salvaneschi, C. Ghezzi and M. Pradella, Context-Oriented Programming: A Programming Paradigm for Autonomic Systems, https://arxiv.org/ftp/arxiv/papers/1105/1105.0069.pdf

[10] L. A. Zadeh. Fuzzy sets. Information and control, vol. 8 (1965), pp. 338–353.

[11] Mohammad Hassan Khooban, Taher Niknam, A new intelligent online fuzzy tuning approach for multi-area load frequency control: Self-Adaptive Modified Bat Algorithm, https://pdfs.semanticscholar.org/c64a/2613f54bec278af1d08750284addd8670a47.pdf

[12] Tommaso Di Noia , Eugenio Di Sciascio , Francesco Maria Donini , Marina Mongiello , Francesco Nocera, Formal model for user-centred adaptive mobile devices, http://sisinflab.poliba.it/publications/2017/DDDMN17/SEN-SI-2016-0169-FINAL.pdf

[13] Maxime Cordy , Andreas Classen , Patrick Heymans , Axel Legay , and Pierre-Yves Schobbens1, Model Checking Adaptive Software with Featured Transition Systems, https://pdfs.semanticscholar.org/cffc/91d82e4956cb50ec682836afdef721348802.pdf

[14] Aimee Borda and Vasileios Koutavas, Self-Adaptive Automata, https://www.scss.tcd.ie/~bordaa/files/formalise/Formalise.pdf

[15] Russel Nzekwa, Romain Rouvoy and Lionel Seinturier, Modelling Feedback Control Loops for Self-Adaptive Systems, https://pdfs.semanticscholar.org/4a29/4cdbc6a12e3c580c281b8056277c54b7834a.pdf

[16] Pieter Vromant, Danny Weyns, Sam Malek and Jesper Andersson, On Interacting Control Loops in Self-Adaptive Systems, https://cs.gmu.edu/~smalek/papers/SEAMS2011.pdf

[17] Regina Hebig, Holger Giese and Basil Becker, Making Control Loops Explicit when Architecting Self-Adaptive Systems, https://www.hpi.uni-potsdam.de/giese/misc/publications/pdf/Hebig_et_al:2010.pdf

[18] Nelly Bencomo, Amel Belaggoun, Valerie Issarny, Bayesian Artificial Intelligence for Tackling Uncertainty in Self-Adaptive Systems: The Case of Dynamic Decision Networks, https://www.researchgate.net/profile/Amel_BELAGGOUN/publication/235984746_Bayesian_Artificial_Intelligence_for_Tackling_Uncertainty_in_Self-Adaptive_Systems_the_Case_of_Dynamic_Decision_Networks/

[19] K.Korb and A.Nicholson , Bayesian artificial Intelligence, 2nd ed. Chapman and Hall, 2010

[20] N. E. Fenton and M.neil, Making decisions: using Bayesian nets and mcda, Knowl-Based Syst. Vol. 14, no.7, pp 307-325, 2001

[21] S.Kullback, Information Theory and Statistics, New York: Wiley 1959

[22] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen, Dynamic and selective combination of extensions in component-based applications.

[23] Andrei Popovici , Thomas Gross, and Gustavo Alonso, Dynamic Weaving for Aspect-Oriented Programming, http://www.lst.ethz.ch/research/publications/AOSD_2002/AOSD_2002.pdf

[24] Miroslaw Malek, Felix Salfner and Günther A. Hoffmann , Prediction-Based Software Availability Enhancement, https://www.researchgate.net/publication/220964149_Prediction-Based_Software_Availability_Enhancement

[25] GitHub, Rainbow Self-Adaptive Framework, https://github.com/cmu-able/rainbow

[26] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl and Peter Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure, http://acme.able.cs.cmu.edu/pubs/uploads/pdf/computer04.pdf

[27] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Making Self-Adaptation an Engineering Reality, https://pdfs.semanticscholar.org/4187/984f18ba76c7f9c8b36c1b8d5b2cfba28b49.pdf

[28] Alexander L. Wolf, Dennis Heimbigner, Antonio Carzaniga, Kenneth M. Anderson, and Nathan Ryan. Achieving survivability of complex and dynamic systems with the Willow framework.

[29] Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis.

[30] T.Quiroz, O. M. Salazar and D. A. Ovalle, Adaptable and Adaptive Human-Computer Interface to recommend Learning Object Repositories, https://pdfs.semanticscholar.org/5e61/5d822487af69e6fb8bc374c4508f539f774d.pdf

[31] Evangelia Kavakli, Goal Oriented Requirements Engineering: A Unifying Framework, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.3596&rep=rep1&type=pdf

[32] Mizbah Fatima, KAOS: A Goal Oriented Requirement Engineering Approach, http://www.ijirst.org/articles/IJIRSTV1I10035.pdf

[33] Eric S. K. Yu, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, http://www.cs.toronto.edu/pub/eric/RE97.pdf

[34] Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein and Emmanuel Letier, Requirements Reflection: Requirements as Run-time Entities, https://www.researchgate.net/publication/221553964_Requirements_reflection_Requirements_as_run-time_entities

[35] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng and Jean-Michel Bruel, RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems, http://www.cse.msu.edu/~mckinley/Pubs/files/Whittle.RELAX.2009.pdf

[36] M. Morandini, L. Penserini, A. Perini, Towards Goal-Oriented Development of Self-Adaptive Systems, http://selab.fbk.eu/morandini/publications/seams010-morandini.pdf

[37] Carlos E. Cuesta and M. Pilar Romay, Elements of Self-Adaptive Architectures, https://distrinet.cs.kuleuven.be/events/soar/2009/contents/papers/cuesta-soar09.pdf

[38] A. G. Ganek, and T. A. Corbi, The dawning of the autonomic computing era, https://pdfs.semanticscholar.org/9e58/7266bfb13e39a9f722ae240a7a78ae7380ea.pdf

[39] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr. and Jason E. Robbins, A Component- and Message-Based Architectural Style for GUI Software, https://users.soe.ucsc.edu/~ejw/papers/c2-icse17.pdf

[40] Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio, Architectural Reflection Realising Software Architectures via Reflective Activities, http://cazzola.di.unimi.it/pubs/edo00-www.pdf

[41] Jim Downling and Vinny Cahill, The K-Component Architecture Meta-Model for Self-Adaptive software, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.2970&rep=rep1&type=pdf

[42] Gordon S. Blair, Geoff Coulson, Anders Andersen and Katia B. Saikoski, The Design and Implementation of Open ORB 2., https://www.researchgate.net/publication/220062871_The_Design_and_Implementation_of_Open_ORB_2

[43] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi and T. Vogel, Software Engineering Processes for Self-Adaptive Systems, https://www.researchgate.net/profile/Thomas_Vogel3/publication/229130556_Software_Engineering_Processes_for_Self-Adaptive_Systems

[44] Object Management Group, Software Process Engineering Metamodel Specification, ftp://ftp.omg.org/pub/spem-rtf/SPEM-CD-20040308.pdf

[45] S. Hillemacher, S. Kriebel, E. Kusmenko, M. Lorang, B. Rumpe, A. Sema, G. Strogl and M. von Wenkstern, Model based Development of Self-Adaptive Autonomous Vehicles using the SMARDT methodology, http://www.se-rwth.de/publications/Model-Based-Development-of-Self-Adaptive-Autonomous-Vehicles-using-the-SMARDT-Methodology.pdf

[46] S.Balaji and M.Sundararajan Murugaiyan, WATEERFALL Vs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC, http://jitbm.com/Volume2No1/waterfall.pdf

[47] D. Garlan, S. Cheng, A. Huang, B. Schmerl and P. Steenkiste, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, http://acme.able.cs.cmu.edu/pubs/uploads/pdf/computer04.pdf

[48] Joachim Hänsel, Thomas Vogel and Holger Giese , A Testing Scheme for Self-Adaptive Software Systems with Architectural Run-time Models, https://arxiv.org/pdf/1805.07354.pdf

[49] C. Xu, S.C. Cheung, X. Ma, C. Cao, J. Lu, Dynamic Fault Detection in Context-aware Adaptation, https://www.researchgate.net/publication/262246582_Dynamic_fault_detection_in_context-aware_adaptation

[50] V. Venkatesh and D. Morris, "User Acceptance of Information Technology: Toward a Unified View"

[51] T. Mens, Introduction and Roadmap: History and Challenges of Software Evolution, https://www.researchgate.net/publication/249607006_Introduction_and_Roadmap_History_and_Challenges_of_Software_Evolution

[52] E. Burton Swanson, The dimensions of maintenance, http://www.mit.jyu.fi/OPE/kurssit/TIES462/Materiaalit/Swanson.pdf

[53] W. Shi, J. Cao, Q. Zhang, Y. Li and Lanyu Xu, Edge Computing: Vision and challenges.

*[54] F.Bonomi, R. Milito, P. Natarajan and J. Zhu, Fog Computing: A Platform for Internet of Things and Analytics,phttps://www.researchgate.net/profile/Rodolfo_Milito/publication/260753114_Fog_*

Computing_A_Platform_for_Internet_of_Things_and_Analytics_Flavio_Bonomi_Rodolfo_Milito_Preethi_Natarajan_and_Jiang_Zhu_N_Bessis_and_C_Dobre_eds_Big_Data_and_Internet_of_Things_169_A_Roadmap_for_Sm

[55] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao. V. Stankovski, Monitoring Self-Adaptive applications within Edge Computing frameworks: a state-of-the-art review, https://www.sciencedirect.com/science/article/pii/S016412121730256X

[56] M. Borkowski, S. Schulte and C. Hochreiner, Predicting Cloud Resource Utilization, http://www.infosys.tuwien.ac.at/staff/mborkowski/pub/UCC2016.pdf

[57] D. Weyns and R. Calinescu, Tele Assistance: A Self-Adaptive Service-Based System Exemplar, https://pdfs.semanticscholar.org/9258/03593ef3335a4eb04921f263aa32af32289d.pdf

[58] Dimitrios Al. Alexandrou, Ioannis E. Skitsas, and Gregoris N. Mentzas, A Holistic Environment for the Design and Execution of Self-Adaptive Clinical Pathways, http://imu.ntua.gr/sites/default/files/biblio/Papers/a-holistic-environment-for-the-design-and-execution-of-Self-Adaptive-clinical-pathways.pdf

[59] Y. Zhag, C. Qian, J. Lv and Y. Liu, Agent and cyber-physical system based self-organizing and Self-Adaptive intelligent shopfloor, http://eprints.gla.ac.uk/145043/1/145043.pdf

[60] Raspberry Pi, https://www.raspberrypi.org

[61] Instructables, READING JSON WITH RASPBERRY PI, https://www.instructables.com/id/Reading-JSON-With-Raspberry-Pi/

[62] University of Oxford, Information Systems Group, HermiT OWL Reasoner, http://www.hermit-reasoner.com

[63] G Hall, Pearson's correlation coefficient, http://www.hep.ph.ic.ac.uk/~hallg/UG_2015/Pearsons.pdf

[64] ECLIPSE FOUNDATION, The Platform for Open Innovation and Collaboration, https://www.eclipse.org

[65] University Of Stanford, A free open-source ontology editor and framework for building intelligent systems, https://protege.stanford.edu

[66] NETBEANS, NetBeans IDE, https://netbeans.org

[67] T. BERNERS-LEE; J. HENDLER; O. LASSILA, The semantic web, Scientific american, 2001, 284.5: 34-43.

[68] Springer – The world Wide Web, https://www.springer.com/cda/content/document/cda_downloaddocument/9783319456973-c1.pdf

[69] J. Bosak, T. Bray, XML and the Second-Generation Web, http://www.floppybunny.org/robin/web/virtualclassroom/xml/scientific_american_xml_web_may_1999.pdf

[70] O. Lassila, R. R. Swick, Resource description framework (RDF) model and syntax specification. 1999.

[71] DBpedia, https://wiki.dbpedia.org

[72] J, Wusteman. Document Type Definition (DTD). In: *Encyclopedia of Library and Information Sciences*. CRC Press, 2017. p. 1381-1389.

[73] D. L. MCGUINNESS, et al. OWL web ontology language overview. W3C recommendation, 2004, 10.10: 2004.

[74] I. Horrocks, S. Bechhofer, Semantic Web, http://www.cs.man.ac.uk/~horrocks/Publications/download/2008/HoBe08.pdf

[75] F. Baader et al. (ed.). *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[76] B. Radunovic, Dinan Gunawardena, P. Key , A. Proutiere , N. Singh , V. Balan×, G. Dejean, Rethinking Indoor Wireless: Low Power, Low Frequency, Full-duplex, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.564.9373&rep=rep1&type=pdf

[77] N. Villegas, H.Muller, G. Tamura, L. Duchien, R. Casallas, A framework for Evaluating Quality-Driven Self-Adaptive Software Systems

[78] M. Litoiu, M. Shaw, G. Tamura, N. M. Villegas, H. Müller, H. Giese, R. Rouvoy, E. Rutten, What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems? , https://hal.inria.fr/hal-01281063/document

[79] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements, https://core.ac.uk/download/pdf/77013454.pdf