POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING

ADAPTABILITY ANALYZER SOFTWARE

Author:
Paolo Paterna

Student ID (Matricola):
852548

Supervisor (Relatore):
Prof. Raffaela Mirandola

Co-Supervisor (Correlatore):
Ph.D. Diego Perez-Palacin

A.Y. 2017/2018

Contents

List of Figures

List of Tables

Acknowledgments

Abstract (Italian version)

Abstract

CHAPTER *1*

Introduction

CHAPTER 2

State of the Art

2.1 Self-Adaptive Software Systems

In modern-day applications, software complexity has extremely increased thanks to the spread of highly available and faster wireless connection such as in the Internet of Things (IoT) ambit. Since software is often deployed in dynamic contexts, where requirements, environment assumptions and usage profiles varies continuously, software complexity increased over time to the point where it is often composed by a number of sub-components and/or sub-services that work together in order to offer a service to the users. This is the case of service-oriented applications – also called Service Based Systems (SBS) – that are composed by multiple *services* and *components*. In these systems, services offered by third-party providers are dynamically composed into workflows to deliver complex functionalities, so SBSs rely on self adaptation to cope with the uncertainties associated with third-party services as the loose coupling of services makes a reconfiguration feasible. Without adaptation, the application is prone to degraded performance because of faulty components, messages lost between services or delays due to an increasing number of users.

During the past decade a lot of research has been made in this scope but the engineering of adaptive systems remains a incredible challenge.[1] In order to solve the problem, **Self-Adapting Software Systems (SASS)** are born. These are flexible systems that can adapt themselves to their contextual needs and can do so with the highest performance and availability. General discussion concerning the issue and the state of the art in the design and implementation have been presented.[1][2][3][4][5][6][7]

These kind of systems have some fundamental properties called auto-managing that are:

- Auto-configuration
- Auto-recovery in case of failure
- Auto-optimization
- Auto-protection

All these properties can be grouped in two more abstract concepts which are self-awareness and context-awareness.

Self-Awareness is the ability of the system to be able to monitor itself in terms of available resources and behavior.

Context-Awareness is the ability of the system to understand the environment where it is working, using the information provided by its components, and adapt itself to all the changes that can occur during its normal operational status. To better understand how a SASS works we need to answer some simple questions:

- Who is adapting?
- Which adaptation is required?
- When is necessary to adapt?
- Where is needed to change something?
- Why is needed an adaptation?
- How we achieve this goal?

During the past years have been developed some dimensions that help to answer all this simple questions: *Time*, *Reason*, *Level*, *Technique* and *Adaptation Control* shown in Figure 2.1.

Who is adapting? As the name suggests, it's the system itself that changes something in order to preserve some given constraint.

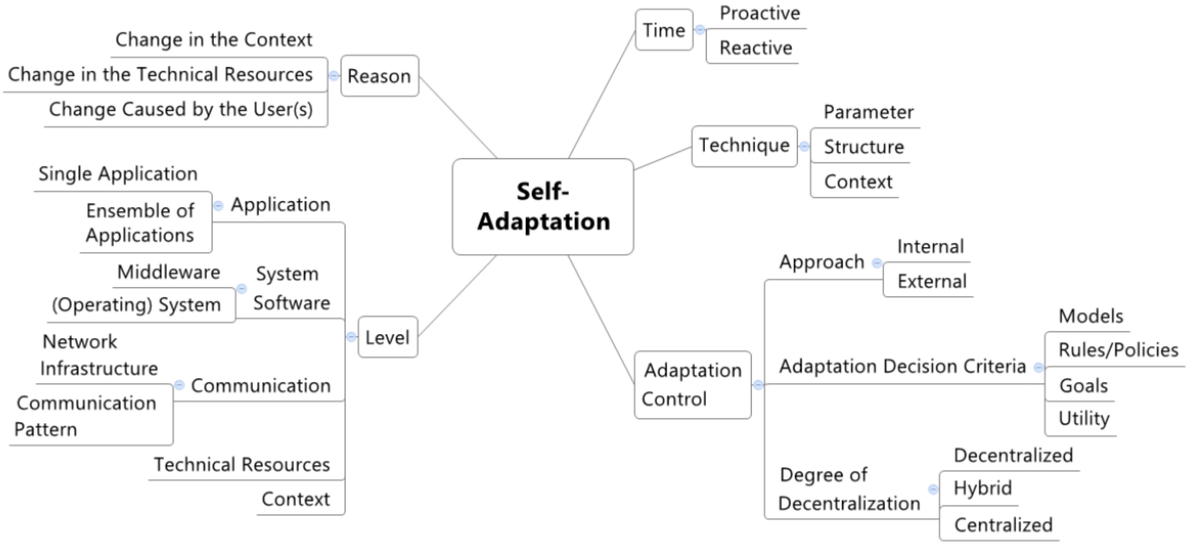

Figure 2.1: *The Dimensions to analyze adaptation.[8]*

Which adaptation is required? The *Technique* dimension is the one that answers this question in fact the software engineer can change either the parameters or the system can be considered as a set of components. The former case allows to fine tuning the system at the expense of an higher complexity, the latter is called composite vision and permits the systems to cooperate exchanging algorithms and much more important, reusing components which improve performance because failed or defected components can be replaced.

When is necessary to adapt? The *Time* dimension is crucial in this situation. There are three typical approaches: the reactive one is the more traditional one which states that an adaptation is needed only after a causative event. The other two approaches are more interesting and they are predictive and proactive. The former studies the system before any event and calculate the need of an adaptation, the latter applies and adaptation despite an event and improves the performance. From the user perspective the proactive approach is the best because it doesn't interrupt the operation of the system in any load but it is the more complicated to implement. Monitoring continuously the system is a costly task to do, on the other side an adaptive monitoring is simple that analyze only specific aspect and/or resources and intervenes only if needed.

Where is needed to change something? In general a SASS is composed by two main part: the the Adaptability Logic (AL) and the Managed Resource (MR). The former in general doesn't change, the latter is composed at the base of the hardware and of the software such as the operating system or, in case of distributed systems, the middleware that control the hardware; at a higher level of the application. These are the parts that require adaptation. To answer this question is needed to decide at which level the operation has to be applied without neglecting the relationship between the MR and the AL which is composed by the network that connects them and/or the view of the communication patterns. Thus *Level* is the considered dimension.

Why is needed an adaptation? In this case, *Reason* is the right dimension. There can be one or more reason because a system needs adaptation such as a change in the available resources, a change in the environment or a change in the user base of the system.

How we achieve this goal? The answer to this question is more complicated than the others because it needs a new topic called *Adaptation Control*.

2.1.1 Adaptation Control

In the literature can be found 2 approaches: the *internal approach* that intertwine the adaptation logic with the system resources, which has problems with the maintainability and scalability of the system, and the *external approach* that splits the system into adaptation logic and managed resources, which increases maintainability and scalability through modularization.

The control unit needs a metric in order to decide how to adapt and in literature are present different metrics: models, rules and policies, goal or utility functions [9].

Another aspect of the adaptation logic is the degree of decentralization. If we have a system with limited resources then a centralized adaptation logic has to be preferred but with greater systems a decentralized AL can improve performance and every sub-system can communicate with another with different patterns of communication. Of course hybrid technique can be made mixing the previous approaches.

Adaptation Logic Issues

As said before a SASS is composed of managed resources and the adaptation logic; it can be represented by the tuple `SASS = (AL, MR)`. *AL* = $a_1, \ldots, a_n$, with a_i representing a logic element, monitors the environment (M), analyzes the data for change (A), plans adaptation (P) and control the execution of the adaptation (E): these are known as *MAPE cycle* or *MAPE functionality*[10]. *MR* = $mr_1, \ldots, mr_n$, with mr_i representing a resource, is the set of resources such as hardware with software, smart-phones, robotics or unmanned vehicles. Figure 2.2 shows a SASS where the dashed line represent the system border. The di-

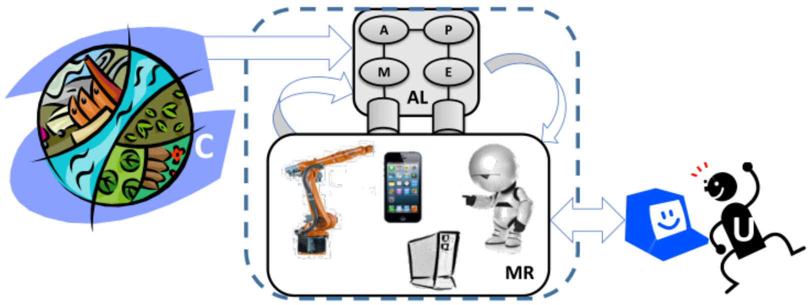

Figure 2.2: *A SASS (AL = Adaptation Logic, MR = Managed Resources, U = User(s), C = Context, M,A,P,E = MAPE functionality).[8]*

mension can therefore be mapped to the MAPE functionalities as shown in table 2.1.

	Time	Reason	Level	Technique
Monitoring	Continuos	What to monitor	Identification of the levels	—
Analyzing	Algorithms depend on reactive or proactive dimension	Where to analyze	—	—
Planning	—	What should be influenced by planning	Adaptation plans address these levels	Plans for performing the techniques
Executing	—	—	Execution of the change on the levels	Execution of the change on the levels

Table 2.1: *Relation of the MAPE Activities and the Dimensions*

2.2 The SOLAR Framework

However working on the adaptability of a system can impact other quality attributes such as performance, reliability or maintainability and in the worst case improving adaptability can decrease part, if not all, of these attributes as stated in [11]: *quality attributes can never be achieved in isolation, the achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others*.

Find a balance between these quality attributes is often a challenging task because sometimes they're conflicting each other, e.g. lower cost and higher availability, so find an adaptability value that can meet all the requisites is, as a consequence, a challenging task too.

The SOLAR (SOftware quaLities and Adaptability Relationships) framework [12] helps the software architect to select the best set of components in order to fulfill the requirements trying to achieve a minimum level for some quality attribute such as availability and/or cost. This tool helps the software architect to build a suitable architecture for his needs but is not a *"solution for every situation"*.

2.2.1 The SOLAR Metrics

All the metrics in SOLAR are greatly inspired by [13] and are all defined at an architecture level and static perspective.

To define them, the software architecture relies on a component-and-connector view (C&C view). In this C&C view *components* are principal computational elements present at runtime. The representation uses the UML diagram. In Figure 2.3 is shown an example of component and their respective connections.

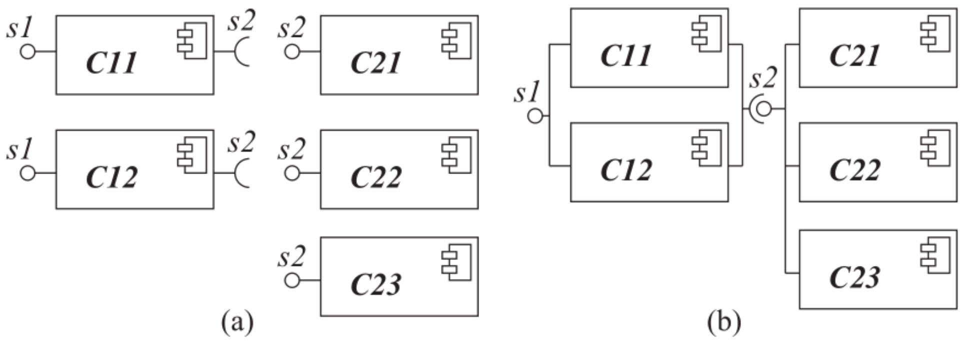

Figure 2.3: *(a) A set of components and their interfaces and (b) the C&C view of the components in (a).[12]*

Components have interfaces attached to ports. *Connectors* are pathways of interaction between components and also have interfaces or roles. In Figure 2.4 is shown an example of an architecture, the used components are highlighted in gray; this example will be used to explain the metrics later in this chapter.

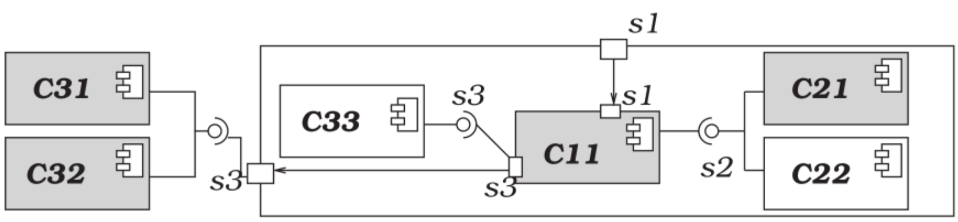

Figure 2.4: *C&C view: discovered components and used components (in gray).[12]*

In the architecture of Figure 2.4:

- the service S1 is provided by component C11 that is unique and must be in use. S1 is the provided service of this architecture.
- component C11 requires S2 and S3 services.
- service S2 is provided by C21 and C22 where only C21 is in use.
- service S3 is provided by C31, C32 and C33 but only C31 and C32 are in use.

Absolute adaptability of a service (AAS)

This metric measures the number of used components for providing a given service.

$$AAS \in \mathbb{N}^n | AAS_i = |UC_i|$$

Quantifies how much adaptable a service is by counting the different alternatives to execute the service (1 no adaptable, >1 adaptable), where the service adaptability grows according to the number of components able to provide it.
Referring to the example in Figure 2.4, we observe that $AAS = [1, 1, 2]$.

Relative adaptability of a service (RAS)

This metric measures the number of used components that provide a given service with respect to the number of components actually offering such service.

$$RAS \in \mathbb{Q}^n | RAS_i = \frac{|UC_i|}{|C_i|}$$

It describes how each service stresses its adaptability choices and it informs how much more adaptable the service could be. RAS vector values near to one mean that the service is using almost all the adaptability potentially reachable.
Referring to the example in Figure 2.4, we observe that $RAS = [1, 0.5, 0.6]$.

Mean of absolute adaptability of services (MAAS)

This metric measures the mean number of used components per service.

$$MAAS \in \mathbb{Q} | MAAS = \frac{\sum_{i=1}^{n} AAS_i}{n}$$

It offers insights into the mean size and effort needed to manage each service.
Referring to the example in Figure 2.4, $MAAS = 4/3 = 1.3$.

Architectures with more adaptable services have higher values of MAAS. Besides, a $MAAS > 1$ means that the architecture includes adaptable services (at least one of the components $AAS_i > 1$). For $MAAS \leq 1$, there may be adaptable services or not (AAS should be checked in this case).

Mean of relative adaptability of services (MRAS)

This metric represents the mean of RAS.

$$MRAS \in \mathbb{Q} | MRAS = \frac{\sum_{i=1}^{n} RAS_i}{n}$$

It informs about the mean utilization of the potential components for each service. Values of this metric range between zero and one.
Referring to the example, $MRAS = (1 + 0.5 + 0.6)/3 = 0.72$.

The higher the MRAS of an architecture, the more adaptable its services are, on average. The maximum value of this metric is obtained when $RAS_i = 1 \ \ \forall i \in [1, \ldots, n]$, which is in turn obtained when all services are as much adaptable as possible because they use all the available components. Therefore, a value close to one for MRAS means that, on average, services are as much adaptable as possible. A value close to zero means that:

a) services can be much more adaptable (adding components not yet used)

b) different architecture alternatives with the same quantity of adaptability can be created

Level of system adaptability (LSA)

This metric measures the number of components used to make up the system with respect to the number of components that the most adaptable architecture would use.

$$LSA \in \mathbb{Q}0..1 | LSA = \frac{\sum_{i=1}^{n} AAS_i}{\sum_{i=1}^{n} |C_i|}$$

The value of this metric ranges between zero and one. For LSA, a value of one means that the system is using all existing components for each service, i.e., $AAS_i = |C_i| \forall i \in 1, \ldots, n$, and then its adaptability is already to the maximum. A value close to one means that the market offers few choices to increase the system architectural adaptability. When a new component is bounded to the architecture, LSA increases in a constant value $(1/\sum i = 1^n |C_i|)$ irrespective of the number of components already considered for the same service.
Referring to the example in Figure 2.4, $LSA = 4/(1 + 2 + 3) = 0.6$.

Table 2.2 summarizes the five metrics and their values for the example in Figure 2.4.

Name	Range	Value	Example in Fig. 2.4
AAS	$\mathbb{N}^n$	$\|UC_i\|$	$[1, 1, 2]$
RAS	$\mathbb{Q}^n \in 0, \ldots, 1$	$\frac{\|UC_i\|}{\|C_i\|}$	$[1, 0.5, 0.6]$
MAAS	$\mathbb{Q}_+$	$\frac{\sum_{i=1}^{n} AAS_i}{n}$	1.3
MRAS	$\mathbb{Q}^n \in 0, \ldots, 1$	$\frac{\sum_{i=1}^{n} RAS_i}{n}$	0.72
LSA	$\mathbb{Q}^n \in 0, \ldots, 1$	$\frac{\sum_{i=1}^{n} AAS_i}{\sum_{i=1}^{n} \|C_i\|}$	0.6

Table 2.2: *Summary of the metrics.[12]*

2.2.2 Relating adaptability to a system quality attribute

The analysis of the relation between system adaptability and quality attributes can give three different results as shown in Table 2.3. In the rows we read that, when the adaptability increases then some quality attributes:

- tend to increase their measured values
- tend to decrease their measured values
- are not affected. We are not interested in this group since we are focussed on the influence of adaptability on the requirement.

The columns in the table consider how the quality requirement is formulated:

- as higher than, e.g., "system availability shall be higher than..."
- as lower than, e.g., "system mean response time shall be lower than..."

Each region of interest in Table 2.3 has been labeled as Helps or Hurts to indicate the effect of the adaptability upon the quality requirement.
The best cases are when the quality attribute completely depends on the adaptability such as:

1. The higher the adaptability, the higher the quality attribute
2. The lower the adaptability, the lower the quality attribute

In Figure 2.5 are shown all the intermediate cases that can result mixing the two extreme cases above. The *X* axis represents the adaptability value, The *Y* axis represent the quality attribute. For each value of adaptability there are two extreme values:

- Q_{A_iU} is the maximum value of the quality attribute with respect to A_i

- Q_{A_iL} is the minimum value of the quality attribute with respect to A_i

Between these extreme values there are all the architecture that have the same adaptability and intermediate quality attribute value. Among all the Q_{A_iU} and Q_{A_iL} in the graph, two of them have a particular meaning: $Adapt^+$ and $Adapt^-$.

To describe the meaning of $Adapt^-$ and $Adapt^+$ we focus on parts (a) and (d) in Figure 2.5. $Adapt^-$ is the lowest A_i for which we can find an architecture satisfying the requirement. $Adapt^+$ is the lowest A_i whose bounds, Q_{A_iU} and Q_{A_iL}, satisfy the requirement. These values indicate that to fulfill the requirement, the architecture must have at least adaptability $Adapt^-$, and, any architecture with at least $Adapt^+$ will also satisfy it. For adaptabilities between them, there will be architectures satisfying the requirement (those highlighted in the figure) and others that will not.

In parts (b) and (c) in Figure 2.5 (regions where the adaptability Hurts), $Adapt^-$ is the threshold adaptability value for which any architecture with adaptability $A_i \leq Adapt^-$ fulfills the requirement; and $Adapt^+$ is the maximum A_i for which we know that exists some architecture that satisfies the requirement.

When adaptability increases	Requirement formulated as	
	Higher than	*Lower than*
The quality attribute value increases	Helps	Hurts
The quality attribute value decreases	Hurts	Helps
The quality attribute is not affected	No effect	

Table 2.3: *Effect of adaptability on a measured quality requirement.[12]*

2.2.3 Analysis of the approach and its limits

Both of these tools want to help the software architect in choosing the right set of components in order to satisfy the adaptability requirements and if possible other quality attributes. With the SOLAR framework all the possible architectures that satisfy the requisites are generated and the choice is left to the architect. This approach is for sure slower but is a valid tool to have an idea of the possible outcome and build an architecture from scratch. On the other side it presents some limitations presented in no particular order:

- It analyzes all the architecture only with a static analysis using the component diagram.
- All components are given equal importance thus the time a component is used and the number of usages per call is completely ignored.
- It does not considers the probability of failure of a component at runtime.

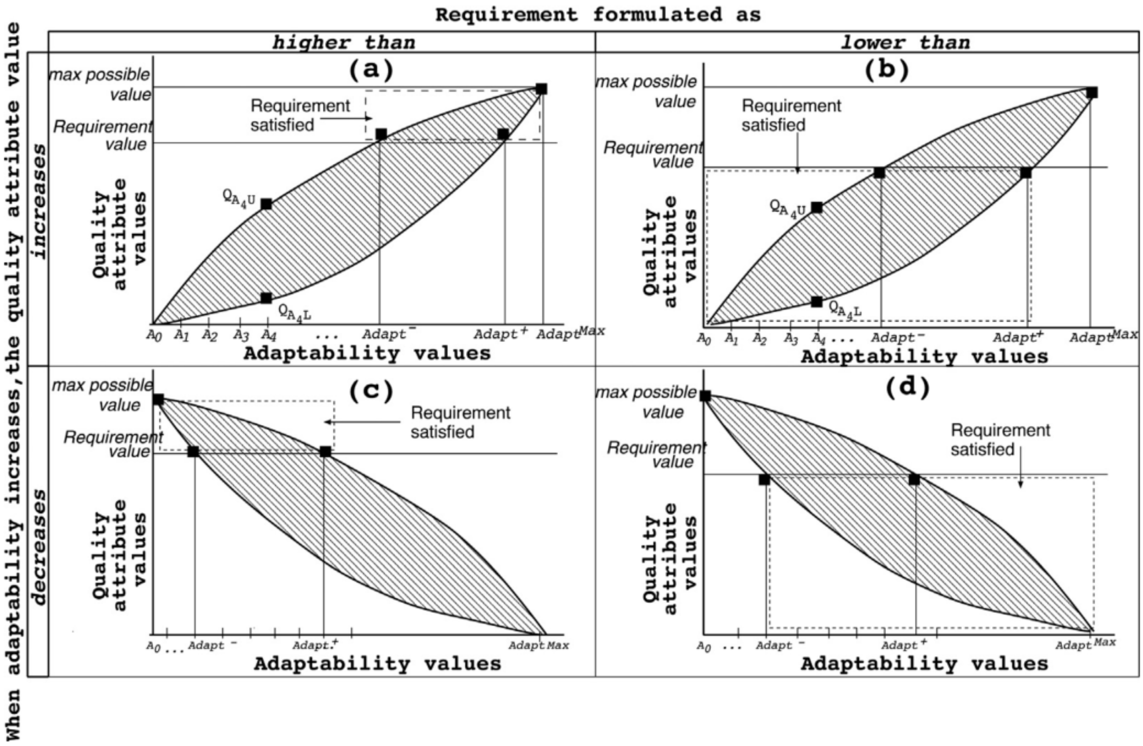

Figure 2.5: *Relations among adaptability and other quality attributes.[12]*

- Requires a lot of time to produce complete results.

In the next chapter is shown how to analyze the quality of a software and in particular how to evaluate the adaptability of a software with respect to some attributes such as cost and availability.

CHAPTER 3

The new quality metrics

In this chapter it is explained how to improve the SOLAR framework in order to integrate the existing metrics with another set of metrics in order to provide the software engineer with more information regarding the architecture he's building.

All modern applications are born to meet some functional requirements that often are subject to changes due to the fact that the environment where they operate is dynamic and can change in a unpredictable way. In the academic and industrial reality rose up the need to standardize some quality metrics in order to evaluate a software, this is the case of the **ISO/IEC 25010** standard called *Systems and software Quality Requirements and Evaluation (SQuaRE)*[14]. However in the self-adaptive context these metrics are not of much help since they do not account the ability of these systems to auto adapt whenever they need it.

To overcome this problem some more new metrics have been defined that analyze the architecture in two different ways:

- using a component diagram as defined in the UML standard[15] to analyze the static behavior
- using a sequence diagram as defined in the UML standard[15] to analyze the runtime behavior

The final objective is to define some adaptability metrics for the architecture as a whole and for every service that the architecture uses. In a more specific way what is shown is how important is every service in a architecture and as a consequence, how important is every component.

3.1 Systems and software Quality Requirements and Evaluation

ISO/IEC 25010, *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE)*[14] is an international standard for the evaluation of software quality that replaced the previous **ISO/IEC 9126** *Software engineering — Product quality standard*[16].

It presents eight product quality characteristics (in contrast to ISO 9126's six):

- Functional suitability - degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- Reliability - degree to which a system, product or component performs specified functions under specified conditions for a specified period of time
- Usability - degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use
- Performance efficiency - performance relative to the amount of resources used under stated conditions
- Maintainability - degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- Portability - degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another
- Compatibility - degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment
- Security - degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization

3.2 The New Metrics

All the metrics shown in this section are focused on two primary quality attributes for the software engineer which are *availability* and *cost*. As a consequence two parameters must be given to perform some of the calculations:

- $availability_{target}$ which represent the minimum availability the system must have
- $cost_{target}$ which represents the maximum cost the system must have

Both this parameters are given by the software engineer and come from the previous analysis of the requisites.

In table 3.1 are summarized all the terms used in the equations.

Name	Meaning
$availability_{target}$	represent the minimum availability the system must have
$cost_{target}$	represents the maximum cost the system must have
C_i	Component *i*
T_{C_i}	Execution time for component *i*
PS_i	Provided service *i*
$N_{exec_{PS_i}}$	Number of executions of provided service *i*
T_{PS_i}	Execution Time of provided service *i*
$TotTime$	Total time of execution
S_i	Service *i*
T_{S_i}	Execution time of service *i*
$N_{exec_{S_i}}$	Number of executions of service *i*
$P_{exec_{S_i}}$	Probability of execution service *i*
$ExecPerCall_{S_i}$	Number of execution per call of service *i*
$N_{exec_{PS_i}}$	Number of executions of provided service *i*
$TimeAction_{S_i}$	The time a service *i* is working
AV_{C_i}	Intrinsic availability of component *i*

Table 3.1: *Summary of the metrics.*

3.2.1 Components

Fitness ratio w.r.t. availability

This metric defines the ratio between a component availability and the target system availability. This result is from the hypothesis that a component with a high availability can provide, at first glance, more guarantees of functioning.

$$FRA_{C_i} \in \mathbb{R}_0^+ \mid FRA_{C_i} = \frac{1 - availability_{target}}{1 - availability_{C_i}}$$

This means that if the result is ≥ 1 the component satisfies the target requisite and as a consequence it can work for a longer time improving the availability of the service it offers.

A value ≥ 0 and < 1 indicates that this component is prone to failure more frequently than how is requested in the architecture.

Fitness ratio w.r.t. cost

Defines the ratio between a component cost and the system target cost.

$$FRC_{C_i} \in \mathbb{R}_0^+ \mid FRC_{C_i} = \frac{cost_{target}}{cost_{C_i}}$$

With the same component cost, if the system target cost grows higher the Fitness Ratio w.r.t. Cost becomes bigger. This implies that the bigger the component and the system target cost gap is, the more can be saved from buying this component w.r.t. the maximum budget invested in buying all the components.

Weight of residence time

Calculates which fraction of time a component is running w.r.t total running time of the architecture.

$$T_{C_i} \in \mathbb{R}^+ \mid T_{C_i} = \sum_{i=0}^{n} N_{exec_{PS_i}} * T_{PS_i}$$

$$TotTime \in \mathbb{R}^+ \mid TotTime = \sum_{i=0}^{n} N_{exec_{S_i}} * T_{S_i}$$

$$WRT \in \mathbb{R}^+ \mid WRT = \frac{T_{C_i}}{TotTime}$$

Higher results means that the component runs more than others, thus is a important piece of the architecture. A failure in this component can compromise the functioning of the architecture in a bigger way than a failure of other components.

3.2.2 Services

Number of executions

Defines the number of times a service is executed.

$$N_{exec_{S_i}} \in \mathbb{N} \mid \forall PS_i \; N_{exec_{S_i}} = \sum_{i=0}^{n} P_{exec_{S_i}} * ExecPerCall_{S_i} * N_{exec_{PS_i}}$$

Probability to be running

Defines the probability that a given service is running in a given moment.

$$PTBR_{S_i} \in [0..1] \mid PTBR_{S_i} = \frac{N_{exec_{S_i}}}{\sum_{i=0}^{n} N_{exec_{S_i}}}$$

In Action

This metrics calculates the probability to find a given service active considering the dynamic analysis of the architecture.

$$TimeAction_{S_i} \in \mathbb{R}^+ \mid \sum_{j=0}^{n} T_{execS_iPath_j} * P_{exec_{S_i}}$$

$$InAction_{S_i} \in [0..1] \mid InAction_{S_i} = \frac{TimeAction_{S_i}}{\sum_{i=0}^{n} TimeAction_{S_i}}$$

It considers all the possible paths available in the selected workflow; in this way a workflow with an `Alt` and/or `Opt` block in the sequence diagram can be represented in a correct way.

3.2.3 Architecture

Global availability of system

Defines the availability of the components that are in the architecture as a probability that are all active in a given instant.

$$GAS \in \mathbb{R}_0^+ \mid \forall C_i \; GAS = \prod_{i=0}^{n} FRA_{C_i}$$

A better availability of a component in the architecture implies a better Fitness Ratio w.r.t. Availability that is reflected in a better global availability. Higher numbers mean better availability.

Global cost of system

Defines the total cost of the components in an architecture w.r.t. the cost of each individual component.

$$GCS \in \mathbb{R}_0^+ \mid \forall C_i GCS = \sum_{i=0}^{n} FRC_{C_i}$$

Total Static Availability

This methods calculate the Availability of an architecture without considering actual workflows of the architecture, so it uses the component diagram. It considers all components as used and considers a call to the main service to use always all the components.

Given that S_x is the main offered service we can calculate the availability of such service as total availability of the system.

- If a component is terminal, so doesn't require any service:

$$Av(C_i) \in \mathbb{R}_0^+ \mid Av(C_i) = AV_{C_i}$$

- If a component is not terminal and requires some service S_k:

$$Av(C_i) \in \mathbb{R}_0^+ \mid \forall S_k \; Av(C_i) = AV_{C_i} * \prod_{k=0}^{n}((1 - p_{S_k}^{C_i}) + p_{S_k}^{C_i} * (AV_{S_k})^{N_{S_k}^{C_i}})$$

With these we can calculate the availability of S_x thus the availability of the architecture with C_i being any component that offers S_x.

$$Av(S_x) \in \mathbb{R}_0^+ \mid \exists C_i \; Av(S_x) = 1 - \prod_{i=0}^{n}(1 - Av_{C_i})$$

CHAPTER 4

The Adaptability Analyzer Software

To provide the software engineer a tool that can ease his work a new software has been developed ad hoc, called Adaptability Analyzer Tool.

This tool has been developed from scratch with some goals in mind and puts together the previous SOLAR[12] metrics with the new metrics presented in Chapter 3. All the goals are explained in the following section.

A new feature was also implemented: the possibility to generate random architectures.

4.1 Adaptability Analyzer Software Goals

4.1.1 User Point of View

From a usability point of view the first goal of this software was to be a tool that can be used by every software engineer without knowing the underlying programming language or how the code is structured; to achieve this I've chosen to provide the software with a Graphical User Interface (GUI) that doesn't require any programming skill to be used.

Another goal was that creating an architecture and then perform calculations should be made easy for everyone, not only for the software engineer, so every result is clearly visible in the GUI and, where possible, numeric results are supported by Cartesian graphs and/or visual representations.

4.1.2 Programmer Point of View

From a programming point of view, instead, the software has to be portable so it can be run on most computers, without strict limitations on operating systems and/or hardware. This is achieved by using the Java programming language[17].

With this programming language the choices on how to develop the GUI were only two framework (discarding the old AWT[18]): Swing[19] or JavaFX[20]. I've chosen to use the newer JavaFX because it should become the new standard for developing Java graphical applications and it provides a set of useful API to the scope of this software.

Others programming goals were to provide a software that can have a small memory footprint and low CPU usage but that can provide results in a meaningful time frame; this is achieved by using common design patterns with efficient algorithms in term of time and memory complexity.

The last goal was to make the user be able to export and import his architecture, thus interrupting and resuming his work on another machine is made easy.

More details on programming choices can be found in the Appendix A

4.2 Adaptability Analyzer Software Features

When launched, the tool presents a welcome page with the main menu accessible in the top left corner. From this menu it is possible to create a new architecture, either from scratch or generating one from the provided generator, or to import a previously created one as shown in Figure 4.1

When creating a new architecture from scratch the architecture's name is required as shown in Figure 4.2 and then the main tool window, with the components tab open, is shown as in Figure 4.4.

When generating an architecture instead some more parameters are required and are summarized in Table 4.1, the window is shown in Figure 4.3

Parameter name	Meaning
Architecture name	Name of the architecture
# of Components	The number of components that the architecture should have
# of Required Functions	The number of functions every component should require
Adaptability Degree	How many copies of the same component should be
Seed	Random number to regenerate the same architecture multiple times

Table 4.1: *The parameters required by the architecture generator.*

Figure 4.1: *The main window of the tool that welcomes the user at launch with menu open.*

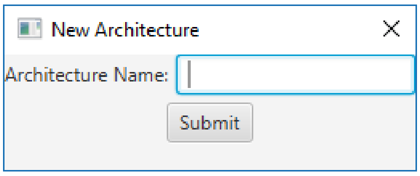

Figure 4.2: *The new architecture window requesting only a name.*

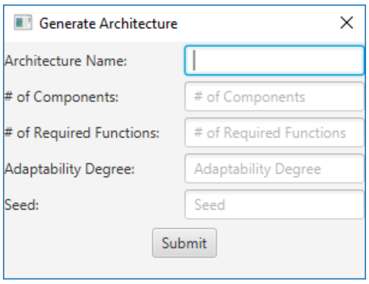

Figure 4.3: *The window that allow to generate an architecture.*

In this case the architecture is named *Arch 1* and the component *C3-1* is selected from the list of available components on the right, visible because it is highlighted. Every component in the list can be deleted by right clicking on it

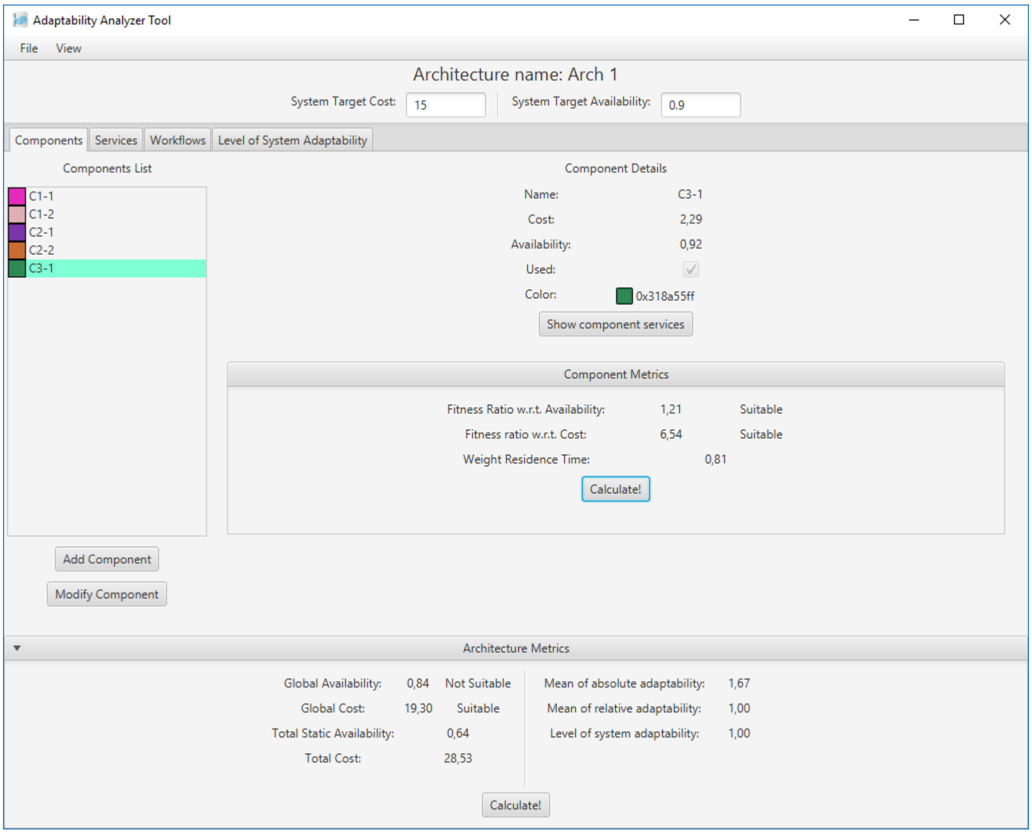

Figure 4.4: *The main tool window showing all the component details.*

and selecting delete from the context menu. On the left is possible to see all the component details. All the fields are explained in Table 4.2.

Parameter name	Meaning
Component name	Name of the selected component
Cost	Cost of the component
Availability	Availability of the component as specified in its technical features
Used	If the component is active in the architecture
Color	The color used in the graphical representation

Table 4.2: *The parameters required by the architecture generator.*

If the top fields, *System Target Cost* and *System Target Availability* are filled as described in Chapter 3 Section 3.2 and at least one component exists, it is already possible to calculate all the component metrics, shown on the right just below the component details, and architecture metrics shown on the bottom of the tool pressing the respective buttons. The meaning of this metrics are explained in Chapter 3.

It is also possible to add a new component or modify an existing one by pressing the respective button under the components list; in this case the window that is opened is the same but in case of modifying a component the fields are already filled as shown in Figure 4.5

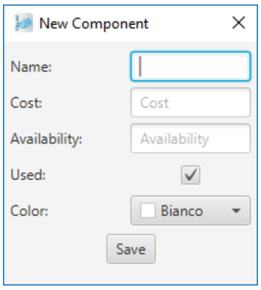

Figure 4.5: *The window that is shown when adding or modifying a component.*

Now is possible to navigate all the other features of the tool.

From the top menu, in the *View* section, it is possible to view a graphical representation of the architecture as shown in Figure 4.6 and from there it is possible to export an image of such representation.

The nodes can be rearranged inside the window by dragging and dropping them.

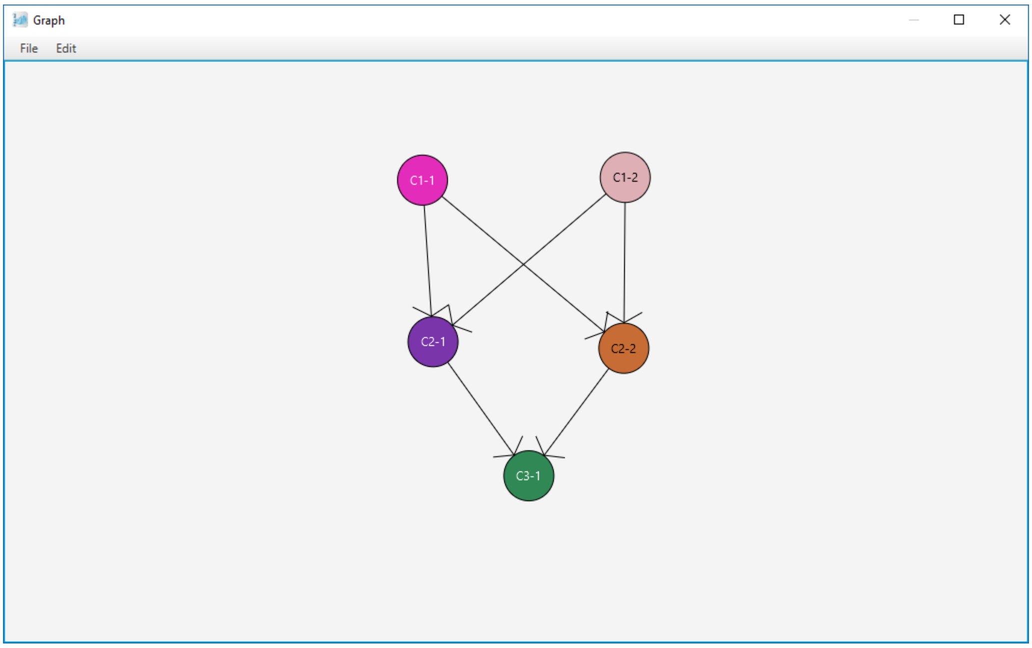

Figure 4.6: *The graphical representation of the architecture in example.*

The service tab shown in Figure 4.7 allows to choose a component from the drop-down menu on the left and a service it provides or requires. It is also possible here to add a service to the selected component by pressing the corresponding button on the bottom of the list as shown in Figure 4.8. Every service can be deleted by right clicking on it and selecting delete from the context menu.

Once selected a service its details are displayed on the right; if some detail does not apply for the selected service it is grayed out automatically. The meaning of the details are explained in Table 4.3

Parameter name	Meaning
Name	Name of the selected service
Type	If the service is *Required* or *Provided* by the component
Execution Time	Time to execute if this service is called (only provided services)
Used Probability	The probability that this service is needed by a call (only required services)
Number of Executions per Call	The number of executions the service must do before giving a result (only required services)

Table 4.3: *The parameters required by the architecture generator.*

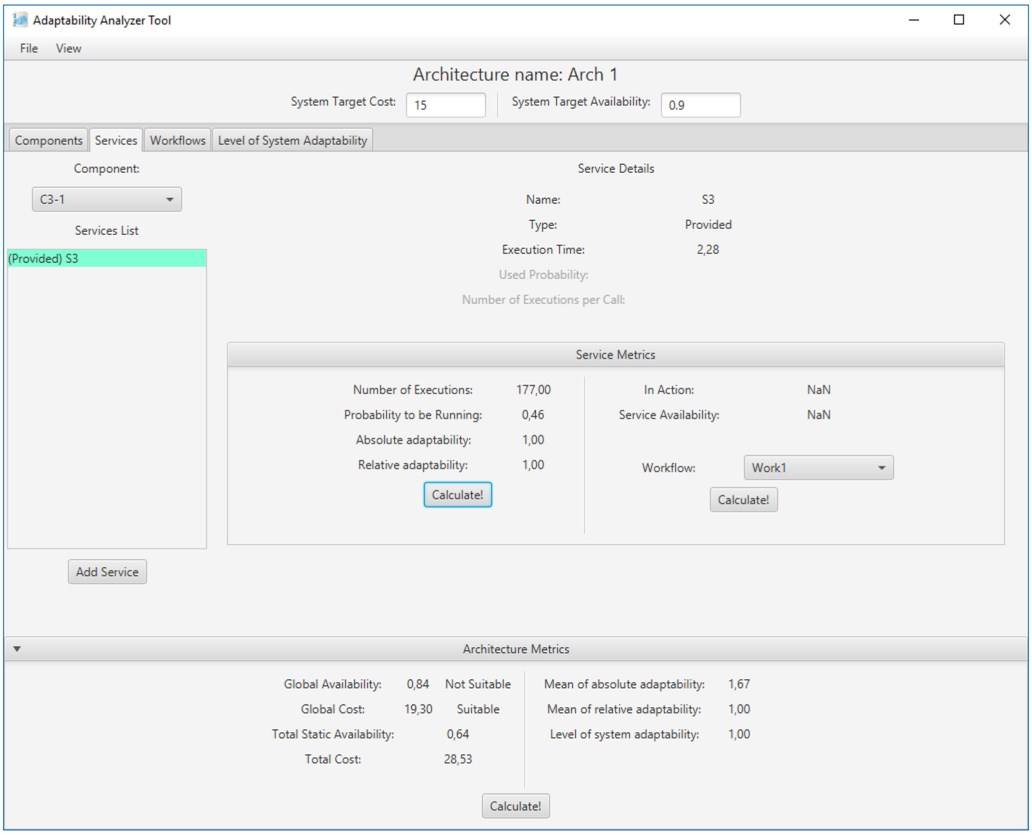

Figure 4.7: *The main tool window showing all the service details.*

The Workflow tab allows to create different workflows for the architecture; it is shown in Figure 4.9. Every workflow is represented as a Sequence Diagram from the UML Standard [15]. Each workflow has one or more paths that can be created using the corresponding button below the path list. Every workflow or path can be deleted by right clicking on it and selecting delete from the context

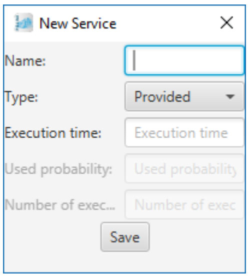

Figure 4.8: *The window that is shown when adding a service.*

menu.

If an `Alt` and/or `Opt` block are needed that can be specified by additional paths which on creation require an execution probability.

The execution probability is shown on top of the left side of the tab and below that is possible to add a message with the corresponding button. The window that allows to create a message is shown in figure 4.10

To delete a message and all its successor just click on the arrow.

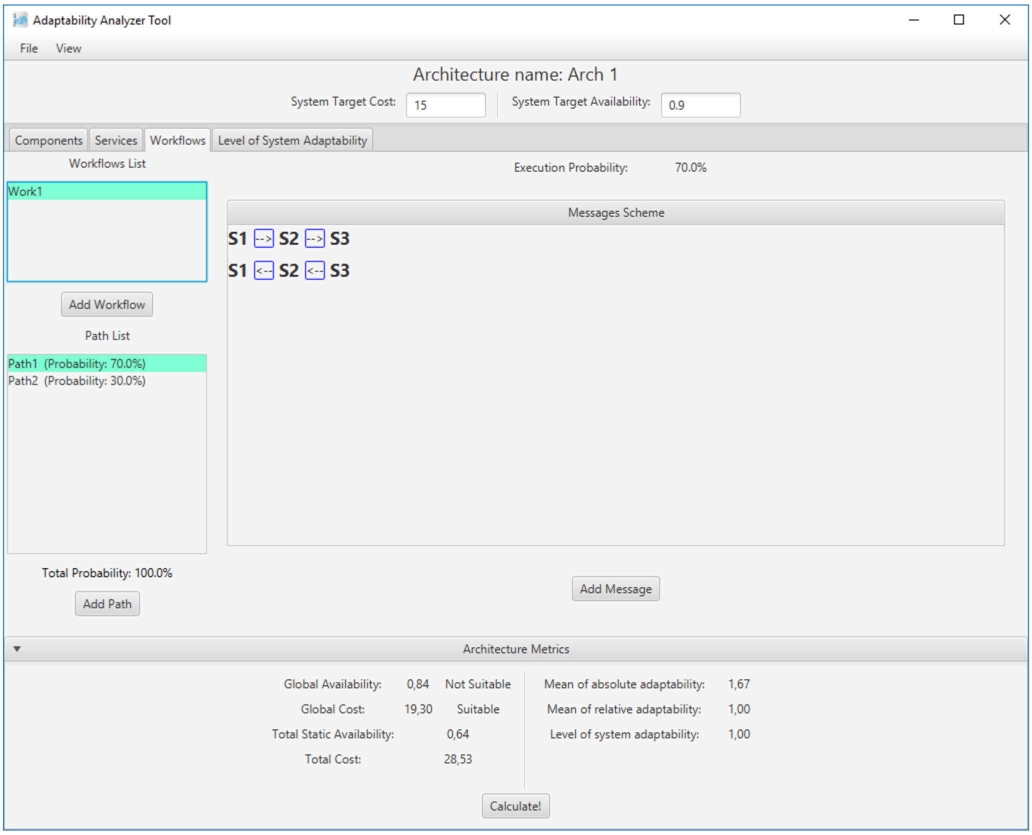

Figure 4.9: *The main tool window showing workflows, paths and the sequence diagram.*

The last tab is the Level of System Adaptability Tab. In this tab it is possible to calculate all possible architectures that can exists with the given components

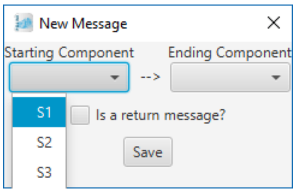

Figure 4.10: *The window that is shown when adding path.*

and calculate how the cost varies with respect to the adaptability as shown in Figure 4.11.

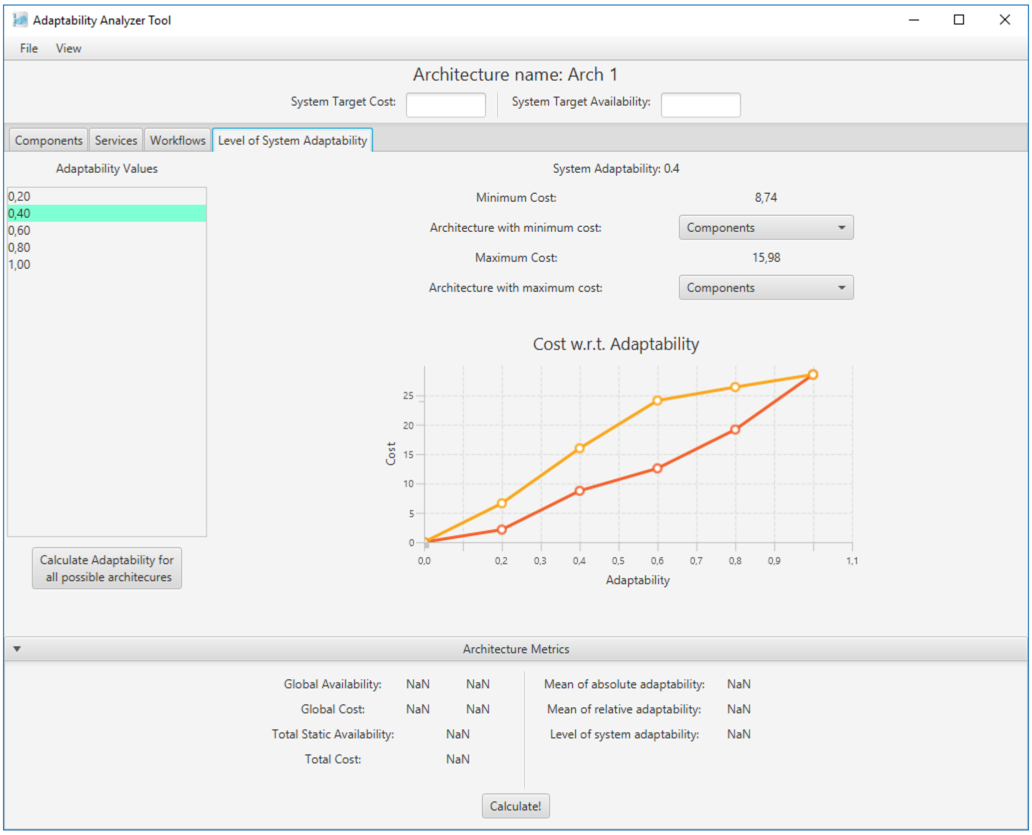

Figure 4.11: *The window that is shown when adding path.*

Clicking an adaptability value in the list on the left or a dot in the graph selects a possible adaptability value and displays on the right the minimum and maximum cost for that adaptability and the list of components that are in the architecture with the selected adaptability as shown in Figure 4.12.

Selecting a component the component tab is shown.

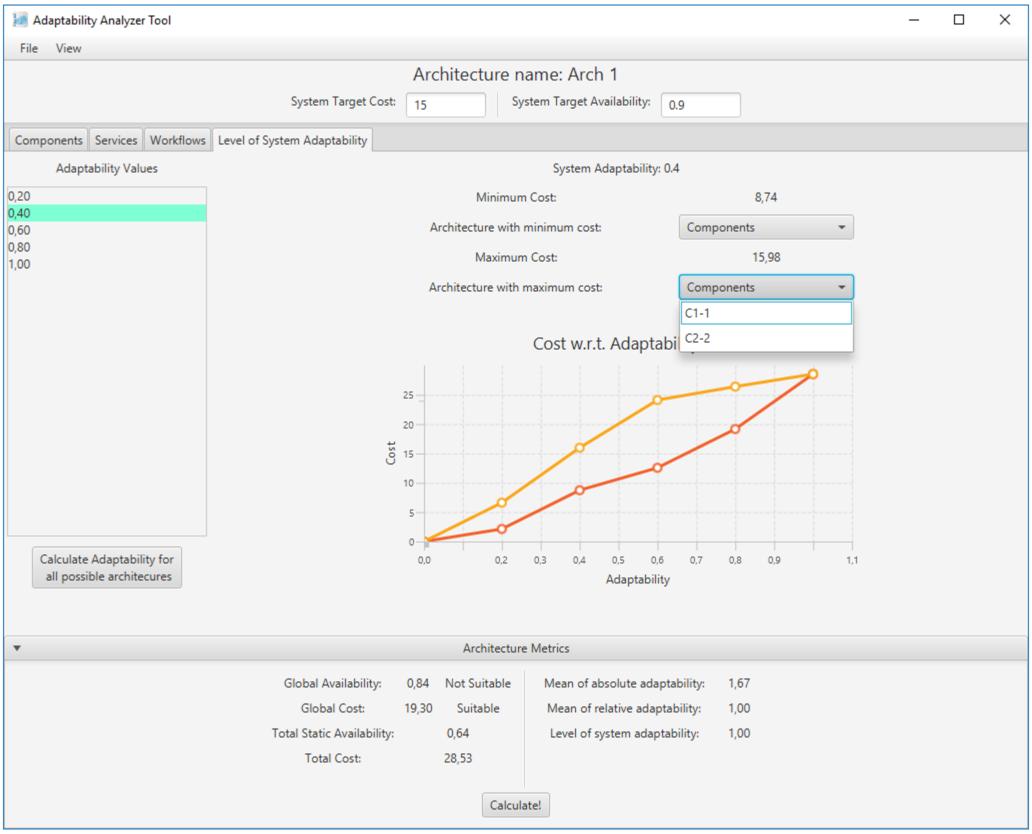

Figure 4.12: *The window that is shown when adding path.*

CHAPTER 5

Experimental Evaluation

5.1 Tele Assistance: A Self-Adaptive Service-Based System

Tele Assistance System, also known as TAS, is a research, done by the University of York [21], on self-adaptation in the domain of service-based systems. Originally introduced in [22] has already been used in the evaluation of several self-adaptation solutions [22], [7], [23], [24], [25], albeit based on ad-hoc implementations, scenarios and evaluation metrics that make the comparison of these solutions and its use to evaluate other solution very difficult.

To address these limitation the University of York implemented TAS on their Research Service Platform (ReSeP) in conjunction with concrete scenarios in order to have an immediate use in evaluation of the self-adaptation solutions.

The system provides health support to chronic condition sufferers within the comfort of their homes. TAS uses a combination of sensors embedded in a wearable device and remote services from healthcare, pharmacy and emergency service providers. As shown in Figure 5.1, the TAS workflow takes periodical measurements of the vital parameters of a patient and employs a third-party medical service for their analysis. The analysis result may trigger the invocation of a pharmacy service to deliver new medication to the patient or to change his/her dose of medication, or the invocation of an alarm service leading, e.g., to an ambulance being dispatched to the patient. The same alarm service can be invoked directly by the patient, by using a panic button on the wearable device.

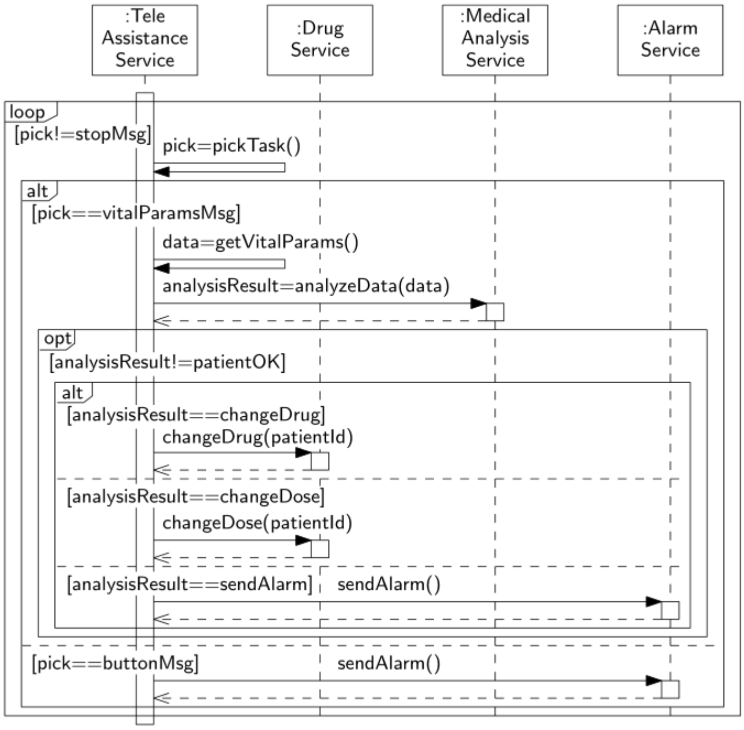

Figure 5.1: *TAS workflow*

They also device some generic adaptation scenarios shown in Table 5.1 and some metrics shown in table 5.2

In conclusion TAS is a reference implementation of a service based system and generic adaptation scenarios associated with different types of uncertainty. First, it aims to promote research and understanding among multiple researchers and research groups, through enabling the comparison of different self-adaptation approaches, without favouring any particular approach. Second, TAS aims to serve the advance of single research efforts by reducing the time required to evaluate self-adaptation solutions. Finally, it aims to contribute to advancing the practice of engineering self-adaptive systems, by being a realistic example of a widely used type of software system.

Scenario	Type of uncertainty	Type of adaptation	Type of requirements
S1	Unpredictable environment: service failure	Switch to equivalent service; Simultaneous invocation of several services for idempotent operation	QoS: Reliability, cost
S2	Unpredictable environment: variation of service response time	Switch to equivalent service; Simultaneous invocation of several services for idempotent operation	QoS: Performance, cost
S3	Incomplete information: new service	Use new service	QoS: Reliability, performance, cost
S4	Changing requirements: new goal	Change workflow architecture; Select new service of the change on the levels	Functional: new operation
S5	Inadequate design: wrong operation sequence	Change workflow architecture	Functional: operation sequence compliance

Table 5.1: *Generic adaptation scenarios for service-based systems*

Quality Attribute	Metrics
Reliability	Number of failed service invocations Number of specific operation sequence failures Mean time to recovery
Performance	Number of specific operation sequences exceeding allowed execution time
Cost	Cumulative service invocation cost over given time period
Functionalities	Number of faulty process executions

Table 5.2: *Quality attributes and metrics for the evaluation and comparison of SBS self-adaptation solutions*

CHAPTER 6

Future Works and Conclusions

APPENDIX *A*

User Manual

Bibliography

[1] R. de Lemos et al. *Software engineering for self-adaptive systems: A second research roadmap*, volume 7475 of *Lecture Notes in Computer Science*. Springer, 2013.

[2] M.C. Huebscher and J.A. McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv. 40*, 3, 2008.

[3] M. Salehie and L. Tahvildari. Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst. 4*, 2:1–42, 2009.

[4] B.H Cheng et al. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, volume 5525. Springer, 2009.

[5] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software engineering for self-adaptive systems*, volume 5525, pages 27–47. Springer, 2009.

[6] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst. 14*, 3:54–62, 1999.

[7] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.

[8] Felix Maximilian Roth, Christian Krupitzer, et al. A survey on engineering approaches for self-adaptive systems. In *Pervasive and Mobile Computing*, volume 17, pages 184–206. Elsevier, 2015.

[9] P. Lalanda, J. A. McCann, and A. Diaconescu. Autonomic computing, 2013.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer 36*, 1:41–50, 2003.

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd edn. SEI Series in software engineering*. Addison-Wesley Pearson Education, Boston, 2003.

[12] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. On the relationships between qos and software adaptability at the architectural level. *The Journal of Systems and Software*, 87, 2014.

[13] N. Subramanian and L. Chung. Process-oriented metrics for software architecture adaptability. *IEEE Computer Society*, page 311, 2011.

[14] International Organization for Standardization (ISO). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, ISO/IEC 25010. International Organization for Standardization Catalogue (`https://www.iso.org/standard/35733.html`), March 2011.

[15] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 2.5.1. OMG Document Number formal/December 2017 (`https://www.omg.org/spec/UML/2.5.1/`), December 2017.

[16] International Organization for Standardization (ISO). Software engineering - Product quality, ISO/IEC 9126. International Organization for Standardization Catalogue (`https://www.iso.org/standard/22749.html`), 2001.

[17] Oracle. Java platform, standard edition 8. `http://www.oracle.com/technetwork/java/javase/documentation/index.html`.

[18] Oracle. Awt framework. `http://www.oracle.com/technetwork/java/javase/documentation/index.html`.

[19] Oracle. Swing framework. `https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html`.

[20] Oracle. Javafx framework. `https://docs.oracle.com/javase/8/javase-clienttechnologies.htm`.

[21] Danny Weyns and Radu Calinescu. Tele assistance: A self-adaptive service-based system examplar.

[22] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET*, 1(6):219–232, 2007.

[23] R. Calinescu, Lars Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, 37(3):387–409, 2011.

[24] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *International Conference on Software Engineering*, 2009.

[25] A. Filieri, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Conquering complexity via seamless integration of design-time and run-time verification. In *Conquering Complexity*. Springer, 2012.