NExt ApplicationS of Quantum Computing

<NE|AS|QC>

Quantum Rule-Based System (QRBS) Requirement Analysis (D6.5)

Document Properties

Contract Number	951821
Contractual Deadline	M16 (31/12/2021)
Dissemination Level	Public
Nature	Report
Edited by :	Vicente Moret-Bonillo (UDC)
Authors	Vicente Moret-Bonillo (UDC), Eduardo Mosqueira-Rey (UDC), Samuel Magaz-Romero (UDC)
Reviewers	Mohamed Hibti (EDF), Alfons Laarman (ULEI), Andrés Gómez [Internal Review] (CESGA)
Date	13/12/2021
Keywords	Requirements Analysis
Status	Final
Release	1.1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 951821

History of Changes

Release	Date	Author, Organization	Description of Changes
0.1	02/09/2021	Samuel Magaz-Romero (UDC)	First draft: structure of the document
0.2	22/09/2021	Vicente Moret-Bonillo (UDC), Eduardo Mosqueira-Rey (UDC), Samuel Magaz-Romero (UDC)	Second draft: definition of needs, features, use cases and test cases.
0.3	18/10/2021	Samuel Magaz-Romero (UDC)	Corrections of second draft after internal revision
1.0	06/12/2021	Samuel Magaz Romero (UDC), Eduardo Mosqueira Rey (UDC)	For review (first)
1.1	13/12/2021	Vicente Moret-Bonillo (UDC), Eduardo Mosqueira-Rey (UDC), Samuel Magaz-Romero (UDC	After reviewer comments

Table of Contents

1. Executive Summary

This report is the second deliverable of Task 6.2 - Quantum Rule-Based Systems (QRBS) for breast cancer detection of the NEASQC project. The document presents the work carried out so far, and is complementary to the other deliverables of this task.

The report begins with an introduction into the requirement analysis process, explaining the necessary concepts to understand the development of the work. Along with said concepts, the applied procedure for the analysis of requirements used in this work is also presented, to put the reader in context and to justify the following sections.

Once the necessary concepts have been presented, the report illustrates the different phases of the work carried out, starting with the needs and features (vision). This phase serves to detect several factors that are crucial to the requirement analysis, such as the actors related to the system or the needs that the system must cover.

The document continues with next step of the process, related to the use cases, which are decompositions of the previous needs according to their functionality and logical structuring. Each use case is studied, explained and detailed in depth. This decomposition allows for a more robust and traceable development across the next phases of the work.

Finally, following the previous decomposition, test cases are defined. This test cases conform a test plan that will help on later stages of the development, when it arises the need to check whether the system meets the requirements specified.

2. Context

2.1. Project

In the context of this project, this document describes the analysis of requirements regarding the development of the framework for Quantum Rule-Based Systems (QRBSs). This analysis has been approached from a software engineering perspective, specifically following the strategy of the Unified Process of software development (Kruchten, 2004).

The requirement analysis is a well-known and established procedure in the field of software engineering, as it is of critical importance to the success or failure of the project. The requirements should be documented, measurable, testable, traceable, related to needs or opportunities, and, in general, defined to a level of detail sufficient for system design (Kotonya & Sommerville, 1998).

However, this document in particular does not delve into quantum computing specifics, since this analysis works on a higher level than implementation. The software specificiations, which include the part related to quantum computing, will be addresed in the next deliverable.

Although there are several ways to classify and categorize requirements, we have decided to not make such kind of distinction between them in this document (besides the possible functional/non-functional classification). This is due to the fact that we have evaluated the project in question and have decided that such a level of decomposition is not necessary in this case, since the complexity of the software product we are developing here does not require such classification.

With this analysis of requirements we identify and describe the needs and problems of the task in question (the ones of the development of the QRBS framework), so we can define the features that our system must provide in order to address and resolve said needs and problems.

2.2. Work package

In the context of the Work Package 6 – “Symbolic AI and graph algorithmics”, this document illustrates one of the steps (the requirement analysis) that must be followed in order to achieve the final result of our task, the development of the framework to represent Quantum Rule-Based Systems, which is one of the methodologies that conforms this work package.

In our previous deliverable (D6.2), we defined the models, architecture and formal specification of Quantum Rule-Based Systems. That first step allowed us to establish and define several critical concepts and definitions for our use case. Now, the theoretical framework explained in that deliverable supports the work carried out on this document, since it will serve as the basis on which we base the features that the final software will implement.

The requirement analysis is a fundamental step in the process of software development since, as we explain later on, it connects the concepts and ideas of the real world with a tangible description that can be studied and improved. This description, for which there are several methodologies to obtain it (ur Rehman, Khan, & Riaz, 2013), provides material that can be discussed in depth more precisely than those concepts from the real world, therefore helping to achieve a better result. It also serves as a blueprint for the work that will come after (design and implementation), and will be of use in case there is a need to get a better understanding of how the software works in the future.

To achieve a requirement analysis of the quality required for the previous characteristics to be present, this work has followed the next stages (Ambler, 2002):

- **Needs and features**: the concepts and ideas of the real world must be declared, including actors and their needs. Although this stage can sometimes be one of the largest in the requirement analysis, we are carrying this work for a software library, so it is briefer than usual, since a software library does not have the complexity of a full software product.

- **Use cases**: the features that have been extrapolated from the actors' needs are decomposed in smaller tasks, called use cases (this is the terminology used by the Unified Process of software development, not to be mistaken with the use case concept of the NEASQC project). Said use cases are related to the actors and to each other, as is illustrated in the use case diagram of section 0.
- **Use cases description**: each use case is taken individually and analyzed in depth, providing its conforming characteristics, among which some of the most important are its name, description and execution flows, since they provide essential information for the comprehension of the use case.
- **Test cases**: in order to ascertain that the carried out work has been correctly developed, a test plan must be designed (ISO/IEC/IEEE, 2010). This plan will be composed of several test cases that check different scenarios and conditions of the system, so we can later confirm whether the software implementation follows the requirement analysis. This will provide a reference that helps to trace the issues in case the test plan returns negative results.

Future deliverables are complementary to this one, since they specifies how the requirements defined in this document must be implemented in order to develop the framework of Quantum Rule-Based Systems, by the means of the software specifications, and how the final software product will be employed for the use case of breast cancer detection.

3. Requirement analysis

In this chapter we explain some of the critical concepts of the requirement analysis so that the reader can understand the following work.

3.1. Definitions

In this section we will briefly detail concepts related to requirements analysis that will be useful to understand the rest of the document. Table 1 shows a summary of all the definitions included in this chapter.

Software requirement	A condition or capability to which the software being built must conform.
Need	Operational problem (opportunity) that must be fulfilled to satisfy an user (also known as a goal or an objective).
Feature	A capability or characteristic of a system that directly fulfills a Need.
Use case	A sequence of actions, performed by the system, that establishes an observable result of value for a particular actor.

Table 1: Definitions

First of all we must define what is a **software requirement**, this can be understood as a condition or capability to which the software being built must conform. There are different ways of understanding and relating requirements to other artifacts of the discipline, such as features and use cases. In our case, we will follow the strategy followed by the Unified Process (UP) of software development (Kruchten, 2004) and in particular the agile version of the Unified Process (Ambler, 2002) which focuses on use cases as the basic artifact that makes up software requirements.

In the unified process scheme, the requirements analysis begins by identifying the user's **needs**, understanding a need as a operational problem (opportunity) that must be fulfilled to satisfy an user (also known as a goal or an objective). From the needs we can identify product **features**, which are high-level capabilities of the system that are required to meet the needs of the users. Both needs and features are detailed in the Vision document, which is responsible for detailing the high-level objectives of the project. In this case, as we are describing a system with simple requirements, the vision document is structured as another section of this document (section 4).

Once we've mapped the need-feature relationships and have determined that the needs and features are correctly accounted for and understood, it's time to consider the next level of the hierarchy: the relationships between the features and the functional and non-functional requirements. Broadly speaking, functional requirements define what a system is supposed to do and non-functional requirements define how a system is supposed to be.

Functional requirements are specified in the UP through the use case model. This model allows us to capture the functionality of the system and serve as a contract between the client and the developers. The use case model is composed of the following elements:

- **Actors**: Set of roles that interact with a system.
- **Use cases**: A sequence of actions, performed by the system, that establishes an observable result of value for a particular actor.
- **Use cases diagram**: Diagram relating actors and use cases through associations and use cases to each other through dependencies and inheritance relationships.

On the other hand, non-functional requirements are those that are not related to any specific functionality, but rather to aspects such as usability, reliability, performance, supportability, and so on. These non-functional requirements are included in a supplementary specification. This supplementary specification has not been included in this first version of the deliverable, as we prefer to focus on the functional requirements, but we do not rule out including it in future iterations of the document.

3.2. Traceability

Experience has shown that the ability to trace requirements artifacts through the different stages of software development is a significant factor in assuring a quality software implementation. The ability to track these relationships and analyze the impact when change occurs is common to many modern, high-assurance software processes.

For a starting point, Institute of Electrical and Electronics Engineers (IEEE) provides a compound definition of traceability: "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match." (ISO/IEC/IEEE, ISO/IEC/IEEE International Standard - Systems and software engineering - Vocabulary, 2017)

Different projects drive different types of requirements artifacts and different ways of organizing them. These decisions, in turn, drive differing needs as to the number and nature of the artifacts to be traced. But all of them have in common that the traceability strategy goes from higher level requirements through more detailed requirements, and then on to implementation and testing.

In our case the traceability hierarchy (Spence & Probasco, 2001) is shown in Figure 1 in which we see that the different needs traces to features, and features traces to use cases and a supplementary specification. The former two (needs and features) form the vision document, the latter two (use cases and supplementary specification) form the software requirements.

Finally, the software requirements lead us to the definition of test cases, the design of these test cases will allow us, in the future, to make a reverse traceability and check that, by successfully fulfilling these tests, we are fulfilling use cases that can be traced to user features and needs.

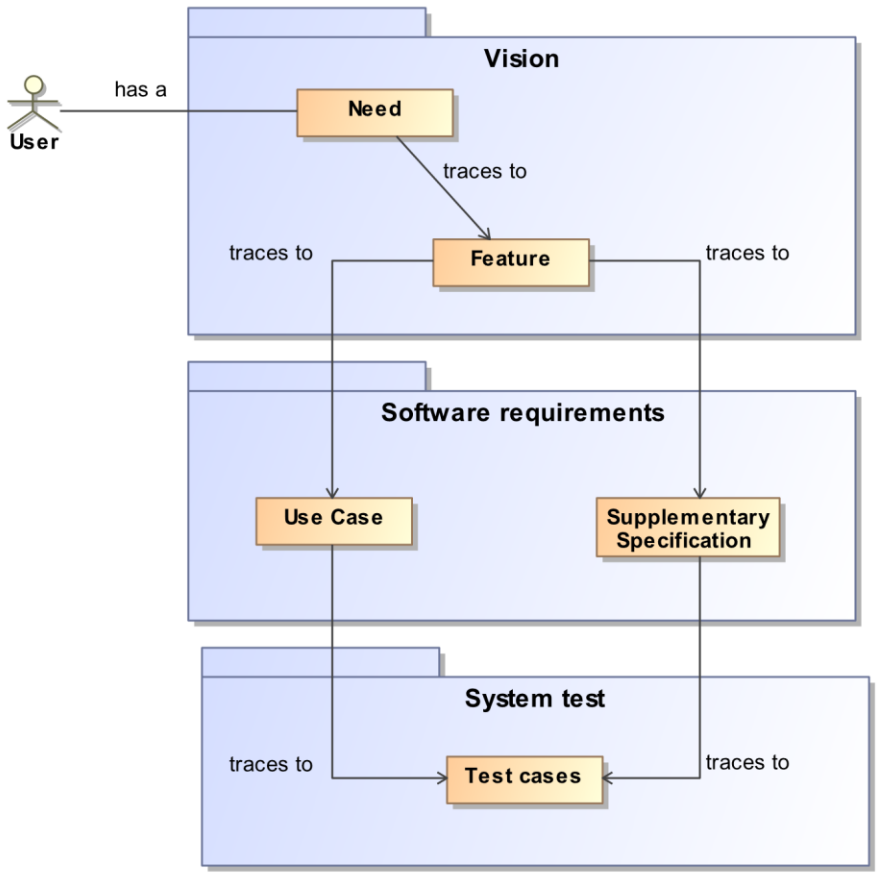

Figure 1: Traceability of the different requirement analysis artifacts.

4. Needs and features (Vision)

In this chapter we will detail the high-level requirements that form the Vision document of the Unified Process. First of all, we will cite the actors or stakeholders (Table 2). In our case there is a single actor, which would be a person who is interested in developing Rule-Based Systems (RBS) to solve certain types of problems. This person will be referred to with the generic name *User* from now on.

Actor ID	A-01
Name	User
Description	Person that uses Rule-Based Systems (RBS) to solve certain types of problems.

Table 2: Specification of actor A-01

From this actor arose the need to develop the Quantum Rule-Based Systems (QRBS) that we detailed in Table 3.

Need ID	N-01
Name	Quantum Rule-Based Systems (QRBS)
Description	Rule-Based Systems (RBS) that use the formalism of Quantum Computing (QC) for representing knowledge and for making inferences and that addresses imprecision and uncertainty.

Table 3: Specification of need N-01

This need is the basis of the following features that we have identified for our system and that are specified in Table 4, Table 5 and Table 6. This features form the basic functioning of a QRBS:

- **Knowledge management**: Knowledge Management (KM) represents the management of the elements that form the knowledge in a QRBS, i.e., facts and rules (Prusak, 2001). These can be grouped into knowledge islands (Perot, 1988) (reduced grouping of inferential knowledge that leads to a hypothesis) to facilitate their management (Hasan, Ramsay, & Moyes, 1994).
- **Uncertainty management**: The real world does not work with absolute positions: a fact may not be completely true or completely false, an implication may not be fulfilled every time. A QRBS that claims to be realistic has to accommodate this uncertainty by allowing for the addition of imprecision to facts and uncertainty to rules (Krause & Clark, 2012).
- **QRBS management**: The user must have the ability to initialize the QRBS, run it, view the results, etc.

Feature ID	F-01
Name	Knowledge management
Description	Declarative KM, procedural KM, knowledge islands management

Table 4: Specification of feature F-01

Feature ID	F-02
Name	Uncertainty management
Description	Allows for the addition of imprecision to facts and uncertainty to rules.

Table 5: Specification of feature F-02

Feature ID	F-03
Name	QRBS management
Description	A QRBS can be initialized, run and its results can be analyzed.

Table 6: Specification of feature F-03

5. Use cases

In this chapter we briefly illustrate how the defined features in chapter 4 are related to the use cases, as well as how these are related to each other with a use case diagram.

5.1. Definition of use cases

In order to follow the methodology of UP, we must now define the use cases that will implement the necessary utilities to address the features that the system must provide. Table 7 establishes the correlation between the features defined previously with the use cases that include behaviour or logic related to said features. Chapter 6 will delve into the use cases, with a more precise, in depth description.

	F-01 Knowledge management	F-02 Uncertainty management	F-03 QRBS management
UC-01 Declarative knowledge management	X		
UC-02 Procedural knowledge management	X		
UC-03 Knowledge island management	X		
UC-04 Uncertainty management		X	
UC-05 QRBS management			X
UC-06 QRBS evaluation			X
UC-07 QRBS execution			X

Table 7: Correspondence between features and use cases

5.2. Use cases diagram

Use cases are grouped and represented in a use case diagram. This type of diagram relates actors and use cases through communication associations (Rumbaugh, Jacobson, & Booch, 2004) and relates use cases to each other through dependencies and inclusion/extensions relationships, that enable reuse of use case described functionality.

In our particular case we have decided to keep the graph simple, so that it is easily understandable, for this reason we have included only inclusion dependencies between use cases. An include dependency is a dependency relationship from a base use case to an include use case, which specifies that the behavior defined by the include use case is inserted into the behavior defined by the base use case.

As it was expected, all the use cases related to knowledge management are included in the QRBS management use case, since it will serve as the main way for the user to interact with the system. Use cases of QRBS execution and evaluation are not included in the QRBS management use case since we have chosen to differentiate those two parts of the user interaction with the QRBSs.

We have decided to make such a differentiation in order to facilitate the use of the framework to the end user. With this approach, he or she will be able to focus firstly on the part of design and implementation of the systems the user wants, and once said systems are defined, he or she can proceed to choose a backend to evaluate and, if possible, execute the system.

The other important type of use case dependency that is usually present in these diagrams is the extend dependency, where a use case adds new steps (extends) to an already defined use case. Like previously exposed, this type of dependency is not present on this diagram since our use cases are already self-contained and there is no need to extend any of them with the analysis carried out until now.

However, we do not discard that in the future, if there appears any need that must be modeled through a new use case, it may happen to have dependencies of inclusion, extension or both with the already defined use cases, following the open-closed principle of the SOLID principles (Martin, 2000).

In Figure 2 we can see the resulting use case diagram. In any case, it is important to note that the use case diagram, in the words of Craig Larman (Larman, 2005) is “a bidimensional mnemonic mechanism that serves a cognitive purpose, i.e., to reveal relationships, use it for that purpose, not to replace the text”.

In other words, a use case diagram summarizes the information in the use case model to facilitate its understanding, but the real software requirements reside in the textual explanation of the use cases, which we include in the section 6.

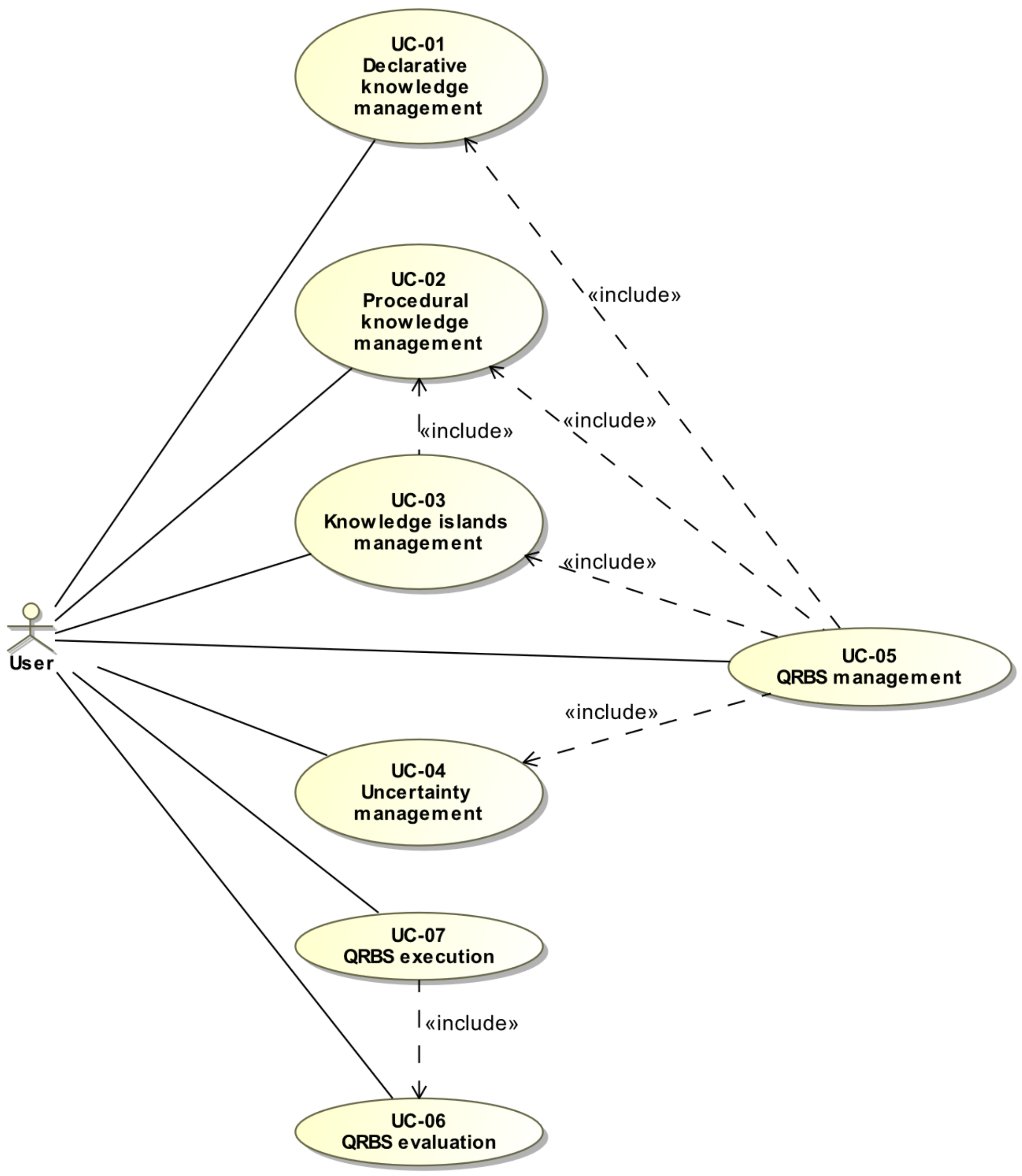

Figure 2: Use cases diagram.

6. Use cases description

6.1. Structure of an use case

In this section, we use the structure of a use case suggested by Craig Larman (Larman, 2005). We also follow the advice and recommendations contained in the books by Cockburn (Cockburn, 2001) and Adolph et al. (Adolph, Bramble, Cockburn, & Pols, 2003) in order to write effective use cases.

In this way, an use case is composed of:

- **Use case ID**: identifier of the use case so it can be easily traced and localized.
- **Name**: short name that synthesizes what the use case is about.
- **Description**: paragraph that explains what does the use case do and/or what functionality does it provide.
- **Actor**: set of roles that involved in the use case (in our case it will always be *User*).
- **Basic flow**: flow that the use case follows by default.
- **Alternative flows**: flows that the use case may follow if the actor takes an option or an error occurs.
- **Scenarios**: possible and/or frequent combinations of flows.
- **Preconditions and Postconditions**: conditions that must be met prior to and following the use case flow.
- **Extension points**: points from where the use case can be extended using additional use cases.
- **Special requirements**: requirements that the use case must comply yet are not critical to the functionality.
- **Additional information**: any extra information considered necessary to understand the use case.

Not all information needs to be specified in all use cases. In addition, following an agile methodology not all use cases have to be fully developed before implementation. This specification will be supplemented as development progresses.

Anyway, as we have few use cases in this work and they are relatively simple, we have decided to specify most of their characteristics in this document. Although these may undergo changes in subsequent iterations.

6.2. CRUD use cases

Before describing the different use cases, it is necessary to make some clarification about the use cases called CRUD. CRUD is an acronym that stands for “Create, Retrieve, Update, Delete”. This uses cases refers to situations in which you create, retrieve, update or delete some information from a given repository or data base.

We have three options for defining CRUD use cases:

- **Separate use cases**: Write a use case for each operation, one for create, one for retrieve, one for update and one for delete. This option is ideal if the use cases of each operation are complex or performed by different actors.
- **One CRUD use case**: A CRUD use case called "manage" or "maintain" the given information is created. This option is ideal if all the use cases are carried out by the same actor and are simple use cases where each one can be solved in a few lines. In this approach, one of the use cases is chosen as the main flow (usually the creation one) and the rest of the use cases are left as alternative flows.
- **Intermediate option**: Use a general CRUD use case that includes “include” relationships with the four separate use cases (creation, retrieve, update and delete). This is generally not a good idea since the main use case usually ends up being a list of include relationships without any

additional noteworthy behavior. It would only be suitable for complicated use cases where the decision of which operation to perform is complex and depends on several factors.

We have decided to follow in this document the “one CRUD use case” strategy using the word “management” to identify them. We believe this is the best way to focus our attention on the author's high-level goals, reducing the number of use cases and not including use cases that are, at their core, mere functions. For example, we have a "Declarative knowledge management" use case which is a CRUD use case to manage the facts of a rule system.

6.3. UC-01: Declarative knowledge management

Use case ID	UC-01
Name	Declarative knowledge management
Description	User must be able to create, initialize and handle facts.
Actor	User
Basic flow	1. User selects to create a fact 2. User provides the attribute and the value of the fact. 3. User adds the fact (asserts) to the working memory.
Alternative flows	1a. User selects to delete a fact 1. User selects the fact to be deleted. 2. The fact is deleted (retracted) from the working memory. 1b. User selects to modify a fact 1. User selects the fact to be modified 2. User modifies the attribute or the value of the fact. 3. The fact is updated in the working memory.
Scenarios	1. Creation of a fact: Basic flow 2. Deletion of a fact: Basic flow, alternative 1a 3. Modification of a fact: Basic flow, alternative 1b
Additional information	

Table 8. Specification of use case UC-01

6.4. UC-02: Procedural knowledge management

Use case ID	UC-02
Name	Procedural knowledge management
Description	User must be able to create, initialize and handle rules.
Actor	User
Basic flow	1. User selects to create a rule. 2. User provides the elements of the left-hand side of the rule (a fact, or a logical combination of facts using logical operators —and, or, not—). 3. User provides the elements of the right-hand side of the rule (that represents the assertion of new facts in the working memory). 4. User adds the rule to the inference engine.
Alternative flows	1a. User selects to delete a rule. 1. User selects the rule to be deleted. 2. The rule is deleted from the inference engine. 1b. User selects to modify a rule. 1. User selects the rule to be modified. 2. User modifies the left-hand side or the left-hand side of the rule. 3. The rule is updated in the inference engine.
Scenarios	1. Creation of a rule: Basic flow 2. Deletion of a rule: Basic flow, alternative 1a 3. Modification of a rule: Basic flow, alternative 1b
Additional information	

Table 9. Specification of use case UC-02

6.5. UC-03: Knowledge islands management

Use case ID	UC-03
Name	Knowledge islands management
Description	User must be able to create, initialize and handle knowledge islands.
Actor	User
Basic flow	1. User selects to create a knowledge island. 2. User provides the rules from the inference engine that conform the knowledge island. 3. User adds the knowledge island to the inference engine.
Alternative flows	1a. User selects to delete a knowledge island 1. User selects the knowledge island to be deleted. 2. The knowledge island is deleted from the inference engine. 1b. User selects to modify a knowledge island 1. User selects the knowledge island to be modified. 2. User modifies the rules of the knowledge island. 1. Includes *UC-02: Procedural knowledge management*. 3. The knowledge island is updated in the inference engine. 3a. The knowledge island has disjoint sets of rules (more than one output) 1. The system will display an error message, “Knowledge island has more than one output”.
Scenarios	1. Creation of a knowledge island: Basic flow 2. Deletion of a knowledge island: Basic flow, alternative 1a 3. Modification of a knowledge island: Basic flow, alternative 1b 4. Erroneous creation of a knowledge island: Basic flow, alternative 3a
Additional information	

Table 10. Specification of use case UC-03

6.6. UC-04: Uncertainty management

Use case ID	UC-04
Name	Uncertainty management
Description	User must be able to add imprecision to facts and uncertainty to rules.
Actor	User
Basic flow	1. The user selects to add imprecision or uncertainty to the system
Alternative flows	1a. Add imprecision to a fact. 1. A value of 1 represents that the fact is true. 2. A value of 0 represents that the fact is false. 3. A value between 0 and 1 represents the degree of imprecision in the truth value of the fact. 1b. Add uncertainty to a rule. 1. A value of 1 represents that the causal relationship of the rule is true. 2. A value of 0 represents that the causal relationship of the rule is false. 3. A value between 0 and 1 represents the degree of uncertainty in the causal relationship of the rule.
Scenarios	1. Only imprecision: Basic flow, alternative 1a 2. Only uncertainty: Basic flow, alternative 1b 3. Both imprecision and uncertainty: Basic flow, alternative 1a, alternative 1b
Additional information	• Imprecision is a property associated with declarative knowledge of the problem, therefore facts can be affected with imprecision. • Uncertainty is a property related to the evidential power of causal relationships, therefore rules can be affected with uncertainty.

Table 11. Specification of use case UC-04

6.7. UC-05: QRBS management

Use case ID	UC-05
Name	QRBS management
Description	User must be able to create, initialize and handle QRBS.
Actor	User
Basic flow	1. User selects to create a QRBS. 2. User assigns knowledge islands to the QRBS.
Alternative flows	1a. User selects to delete a QRBS 1. User selects the QRBS to be deleted. 2. The QRBS is deleted. 1b. User selects to modify a QRBS 1. User selects the QRBS to be modified. 2. User modifies the elements of the QRBS. a. The knowledge islands are modified. Includes *UC-03: Knowledge islands management*. b. The procedural knowledge is modified. Includes *UC-02: Procedural knowledge management*. c. The declarative knowledge is modified. Includes *UC-01: Declarative knowledge management*. d. The uncertainty information is modified. Includes *UC-04: Uncertainty management*. 3. The QRBS is updated.
Scenarios	1. Creation of a QRBS: Basic flow 2. Deletion of a QRBS: Basic flow, alternative 1a 3. Modification of a QRBS: Basic flow, alternative 1b
Additional information	

Table 12. Specification of use case UC-05

6.8. UC-06: QRBS evaluation

Use case ID	UC-06
Name	QRBS evaluation
Description	User must be able to evaluate the feasibility of the QRBS. This means to evaluate whether, given a quantum backend for its execution, the system can or cannot run on it.
Actor	User
Basic flow	1. User selects the QRBS to evaluate 2. User indicates the quantum backend to evaluate with 3. The QRBS is evaluated regarding said quantum backend
Alternative flows	3a. The evaluation returns a positive result 1. The system displays information of each knowledge island (number of qubits and gates required) 3b. The evaluation returns a negative result (the QRBS cannot run in the indicated backend) 1. The system displays a message with the causes why it cannot run (e.g.: a knowledge island is too large, the given backend does not support a certain operation).
Scenarios	1. Positive evaluation: Basic flow, alternative 3a 2. Negative evaluation: Basic flow, alternative 3b
Additional information	

Table 13. Specification of use case UC-06

6.9. UC-07: QRBS execution

Use case ID	UC-07
Name	QRBS execution
Description	User must be able to execute the system (by default, running all of its knowledge islands) and obtain the results.
Actor	User
Basic flow	1. User selects the QRBS to execute 2. User indicates the quantum backend to execute with 3. The QRBS is evaluated: Includes *UC-06: QRBS Evaluation* 4. The QRBS is executed regarding said quantum backend
Alternative flows	1a. User does not specifiy which knowledge islands to execute 1. All knowledge islands of the QRBS will be executed 1b. User specifies which knowledge islands wants to execute 1. The specified knowledge islands of the QRBS will be executed 4a. The execution returns a positive result, with information of said execution (results for each of the knowledge island, time and duration of execution) 4b. The execution returns a negative result (the QRBS cannot run in the indicated backend) 1. The system will display a message with the causes why it cannot run (e.g.: a left-hand side of the rule is not initialized, a runtime error ocurred).
Scenarios	1. Successful default execution: Basic flow, alternative 1a, alternative 3a (*UC-06*), alternative 4a 2. Failed default execution (because of evaluation): Basic flow, alternative 1a, alternative 3b (*UC-06*) 3. Failed default execution (because of execution): Basic flow, alternative 1a, alternative 3a (*UC-06*), alternative 4b 4. Successful specified execution: Basic flow, alternative 1b, alternative 3a (*UC-06*), alternative 4a 5. Failed specified execution (because of evaluation): Basic flow, alternative 1b, alternative 3b (*UC-06*) 6. Failed specified execution (because of execution): Basic flow, alternative 1b, alternative 3a (*UC-06*), alternative 4b
Additional information	See in Figure 3 an activity diagram that shows graphically the execution of this use case.

Table 14. Specification of use case UC-07

Since the QRBS execution is the central part of the functioning of the system we wanted to clarify a little how it works by adding an activity diagram (Figure 3) that shows the different processes that are carried out.

The execution starts with the user designing a rule-based system (RBS) for a given domain problem. The "QRBS Management" activity allows us to express this RBS in a model that can be processed by a quantum computer. This QRBS can be executed in a quantum backend if the user adds a specification of it to the "QRBS execution" activity.

Before executing a QRBS in a quantum backend, the system evaluates whether this is possible. If it is not, the reasons why it is not executable on that backend will be specified. If it is executable, the QRBS will be adapted to run on the backend and the results will be displayed.

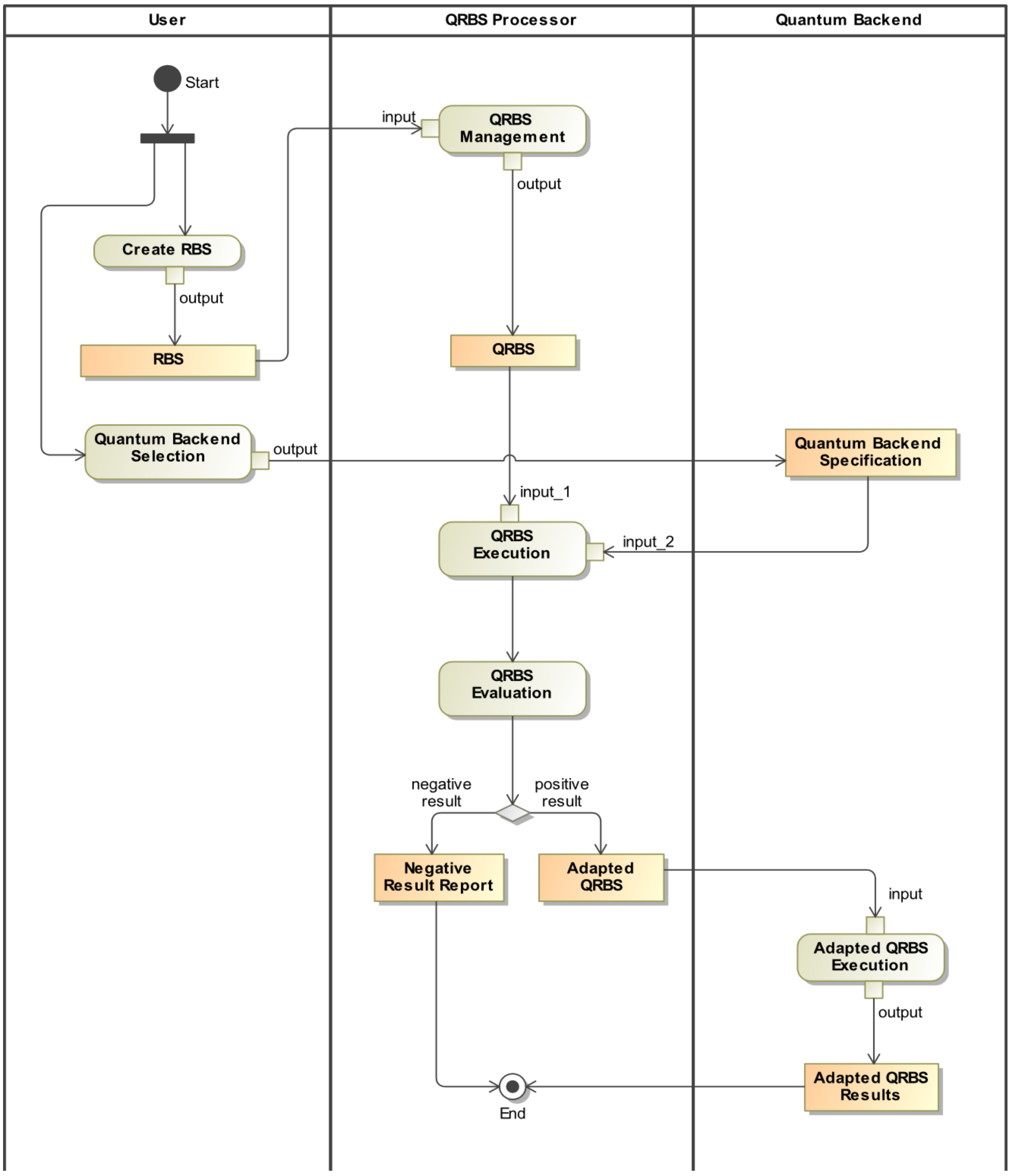

Figure 3: Activity diagram showing the QRBS execution process.

7. Test cases

The last system boundary that we must bridge to implement a complete traceability strategy is the bridge from the requirements domain to the testing domain. One specific approach to comprehensive testing is to ensure that every use case is “tested by” one or more test cases. This was already reflected in Figure 1 that shows our traceability model.

However, this simple diagram understates the complexity of the case somewhat, for it is not a trivial 1-on-1 relation that is described. First, we have to identify all the scenarios described in the use case itself. This is a one-to-many relationship, as an elaborated use case will typically have a variety of possible scenarios that can be tested. From a traceability viewpoint, each use case “traces to” each scenario of the use case.

Then each scenario can drive one or more specific test cases. In Table 15 we can see a traceability matrix of one-to-many (use case to scenario) and an additional one-to-one or one-to-many (scenario to test case) relationship that fully describe the relationship among these elements. This will allow us in the future to verify that we have developed the system according to the specifications provided in this document.

Use Case	Scenario number	Test case ID	Test case name
UC-01	1	T-01-1	Creation of a fact
	2	T-01-2	Deletion of a fact
	3	T-01-3	Modification of a fact
UC-02	1	T-02-1	Creation of a rule
	2	T-02-2	Deletion of a rule
	3	T-02-3	Modification of a rule
UC-03	1	T-03-1	Creation of a knowledge island
	2	T-03-2	Deletion of a knowledge island
	3	T-03-3	Modification of a knowledge island
	4	T-03-4	Erroneous creation of a knowledge island
UC-04	1	T-04-1	Uncertainty management with only imprecision
	2	T-04-2	Uncertainty management with only uncertainty
	3	T-04-3	Uncertainty management with imprecision and uncertainty
UC-05	1	T-05-1	Creation of a QRBS
	2	T-05-2	Deletion of a QRBS
	3	T-05-3	Modification of a QRBS
UC-06	1	T-06-1	Positive evaluation
	2	T-06-2	Negative evaluation
UC-07	1	T-07-1	Successful default execution
	2	T-07-2	Failed default execution (because of evaluation)
	3	T-07-3	Failed default execution (because of execution)
	4	T-07-4	Successful specified execution
	5	T-07-5	Failed specified execution (because of evaluation)
	6	T-07-6	Failed specified execution (because of execution)

Table 15. Traceability from use cases to test cases.

8. Acronyms and Abbreviations

Term	Definition
AI	Artificial Intelligence
IEEE	Institute of Electrical and Electronics Engineers
KM	Knowledge Management
QC	Quantum Computing
RBS	Rule-Based System
QRBS	Quantum Rule-Based System
RBS	Rule-Based System
UC	Use Case
UP	Unified Process

Table 16: Acronyms and Abbreviations

9. List of Figures

10. List of Tables

11. Bibliography

Adolph, S., Bramble, P., Cockburn, A., & Pols, A. (2003). *Patterns for Effective Use Cases.* Pearson Education, Inc.

Ambler, S. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process.* John Wiley & Sons.

Cockburn, A. (2001). *Writing Effective Use Cases.* Addison-Wesley.

Hasan, K., Ramsay, B., & Moyes, I. (1994). Object oriented expert systems for real-time power system alarm processing: Part I. Selection of a toolkit. *Electric Power Systems Research, 30*(1), 69-75.

ISO/IEC/IEEE. (2010). ISO/IEC/IEEE International Standard - Systems and software engineering -- Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, 1-418.

ISO/IEC/IEEE. (2017). *ISO/IEC/IEEE International Standard - Systems and software engineering - Vocabulary.* ISO/IEC/IEEE. doi:10.1109/IEEESTD.2017.8016712

Kotonya, G., & Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* John Wiley & Sons.

Krause, P., & Clark, D. (2012). *Representing uncertain knowledge: an artificial intelligence approach.* Springer Science & Business Media.

Kruchten, P. (2004). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.

Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Pearson Education, Inc.

Martin, R. C. (2000). Design principles and design patterns. *Object Mentor, 1*(34), 597.

Perot, F. (1988). A knowledge acquisition assistant for the expert system shell Nexpert-Object.

Prusak, L. (2001). Where did knowledge management come from? *IBM Systems Journal, 40*(4), 1002-1007.

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education.

Spence, I., & Probasco, L. (2001). Traceability Strategies for Managing Requirements with Use Cases. *Rational Software White Paper*. Recuperado el 13 de 12 de 2021, de https://www.semanticscholar.org/paper/Traceability-Strategies-for-Managing-Requirements-Spence-Probasco/4ed79d38b02b070b165ab91267a7390b89903fc6

ur Rehman, T., Khan, M. N., & Riaz, N. (2013). Analysis of requirement engineering processes, tools/techniques and methodologies. *International Journal of Information Technology and Computer Science (IJITCS), 5*(3), 40.