NExt ApplicationS of Quantum Computing

D6.11: Preliminary QRBS software and IDC application specification

Document Properties

Contract Number	951821
Contractual Deadline	June-2023
Dissemination Level	Public
Nature	Report
Editors	Vicente Moret-Bonillo, UDC Andrés Gómez Tato, Cesga
Authors	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC Diego Álvarez-Estévez, UDC
Reviewers	Mohamed Hibti, EDF Alfons Laarman, ULEI
Date	15-jun-2023
Keywords	software implementation, IDC application, QRBS
Status	Under Review
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, Organisation	Description of Changes
1.0	26/05/2023	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC Diego Álvarez-Éstevez, UDC	First version.
1.1	15/06/2023	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC Diego Álvarez-Éstevez, UDC	First revision.

© NEASQC Consortium Partners. All rights reserved.

Table of Contents

1. Executive Summary **4**

2. Context **5**
2.1. Project 5
2.2. Work package 5

3. Preliminary QRBS Software **6**
3.1. Introduction 6
3.2. Code Example 6
3.2.1. QRBS Creation 6
3.2.2. QRBS Evaluation 7
3.2.3. QRBS Execution 8
3.3. Code Documentation 8

4. IDC Application Specification **9**
4.1. Introduction 9
4.1.1. Categorical reasoning in medicine 9
4.1.2. Description of the clinical problem 9
4.2. A model for the staging of IDC 10
4.3. Quantum system design for breast IDC staging 11

5. Testing **14**
5.1. Traceability with use cases 14
5.2. Testing methodology and tools 15
5.2.1. Testing approach 15
5.2.2. Measuring test coverage 15
5.2.3. Testing tools 15
5.3. Test example 16

6. Conclusions **18**

List of Acronyms **19**

List of Figures **20**

List of Tables **21**

Bibliography **22**

A. QRBS software documentation **23**
A.1. neasqc_qrbs.knowledge_rep module 23
A.2. neasqc_qrbs.qrbs module 26

© NEASQC Consortium Partners. All rights reserved.

1. Executive Summary

This report is the fourth 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.

D6.11 (M32) will include a preliminary version of the RBS and QRBS software along with the specification of the IDC application that will be developed.

The report begins with an introduction of the preliminary version of the QRBS software, presenting how the project is structured, providing some use examples of common operations users will make with the library, and a brief commentary on the library's documentation which is presented in an appendix.

Following that, we present the specification for the Invasive Ductal Carcinoma (IDC) application, based on previous work on quantum computing techniques and applying them to the clinical problem specifically.

We continue by extending the work on previous deliverables regarding traceability and testing of the QRBS software, which is a critical part on making sure that the developed library provides valid and verified functionalities.

To close the report, the conclusions obtained during the development of the work carried out are presented and some ideas for future work to be included in upcoming deliverables are shown.

© NEASQC Consortium Partners. All rights reserved.

2. Context

Some of the concepts referenced in this document have been explained in deliverables D6.2, D6.5 and D6.9. We encourage to read them for contextual information.

2.1. Project

In the context of this project, this document provides insight into two different objectives: the preliminary version of the Quantum Rule-Based Systems software library and the initial specification for the Invasive Ductal Carcinoma application.

On the one hand, following the work of previous deliverables, the development of the QRBS software library has been continued by implementing a preliminary version, while still following the Unified Process of software development (Kruchten, 2004). This preliminary version plays several roles: (1) the implementation of the software library goes one step further in the software development life cycle, (2) gives insight into how adequate the previous steps were, depending on how they ease the process of coding, and (3) provides enough functionalities to model any desired QRBS and run it on a basic myQLM simulator, so it is already a working version.

On the other hand, we provide the IDC application specification, following the current manuals of medicine for cancer staging (Amin et al., 2017). This work was introduced as an appendix in the previous deliverable, and now we delve into how the model presented then can be implemented in a quantum routine.

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 software implementation) that must be followed in order to develop the framework of Quantum Rule-Based Systems, and the initial specification for the IDC application.

In our previous deliverable, "D6.9 QRBS software speficications" (Moret-Bonillo, Gomez Tato, et al., 2022), we established the specifications that the software library for Quantum Rule-Based Systems must comply. These specifications allow for the work of the analysis and design stages to be tied with the software programming process, by providing the different attributes that each class must provide, as well as their methods and how they must behave.

Regarding the IDC application, we provide its specification, providing the first step on the use case of breast cancer diagnosing. This first phase involves researching how IDC is classified, also known as staging, in order to build a model (which we did in our previous deliverable) so we can implement it using the QRBS software library. However, this is a complicated process, so we delve on categorical reasoning and the clinical problem specifically, making a great effort on "entangling" symbolic AI and Quantum Computing.

Future deliverables are complementary to this one, since they delve into the following versions of the QRBS software and the continuation of the developement of the IDC application.

© NEASQC Consortium Partners. All rights reserved.

3. Preliminary QRBS Software

In this chapter, we continue the work carried out in the previous deliverables regarding the developement of the QRBS software library, this time presenting the preliminary version of the software. For that, we (1) introduce the software library, its structure and repository, (2) present a use example with code snippets, and (3) give a glimpse of its documentation so far.

3.1. Introduction

Following the objectives established at the beggining of this project, we are developing a Python programming library to design and use Quantum Rule-Based Systems. This library is currently hosted in https://github.com/neasqc/qrbs.

Among the contents of this repository, we can find:

- **neasqc_qrbs**: the directory with the source code of the library
  - **knowledge_rep.py**: the source code for all the classes regarding the elements for knowledge representation (facts, operators, rule, knowledge islands, etc.)
  - **qrbs.py**: the source code for all the classes regarding the management of QRBS and QPUs (creation, evaluation, exeuction, etc.)
- **tests**: the directory with the source code of the tests for the library
  - **test_knowledge_rep.py**: the tests for the knowledge representation elements
  - **test_qrbs.py**: the tests for the QRBS and QPUs
- **doc**: the directory with the Sphinx files to build the documentation (the content itself is coded with PyDoc inside their corresponding classes).

Regarding the repository organization, we are following a worflow similar to Gitflow:

- **Branches**:
  - **main**: for releases of final versions.
  - **develop**: to integrate the different features before making a release.
  - **feature/[FEATURE_NAME]**: one per each feature being developed (these are deleted once the feature is finished).
- **Tags**: we tag each version release.

With the publishing of this deliverable, we are making available version 0.1.0 of the library. This version is not complete (it is a preliminary version), but provides the functionalities to create a QRBS (defining its elements) and both evaluate it and execute it in myQLM's PyLinAlg simulator. We have the objective of releasing version 1.0.0 at the end of the project.

3.2. Code Example

In this section we provide an example of how a user would use the library to create a QRBS, evaluate it against a QPU and execute it to obtain the results. Since the objective is to demonstrate the use of the library rather, we are using a QRBS with a single knowledge island, which is derived from the inferential circuit illustrated in Figure 1. The values of the facts will all be set to 1.0 (since they are abstract facts their value is not relevant), and imprecision and uncertainty are initialized randomly.

3.2.1. QRBS Creation

Firstly, we initialize a new QRBS object, and assert in it the different knowledge representation elements (facts, rules and knowledge islands). We need to store them in variables so we can use them later to define the higher elements of the hierarchy, as well as to check their values after we run the system. Listing 3.1 illustrates how this would be done.

© NEASQC Consortium Partners. All rights reserved.

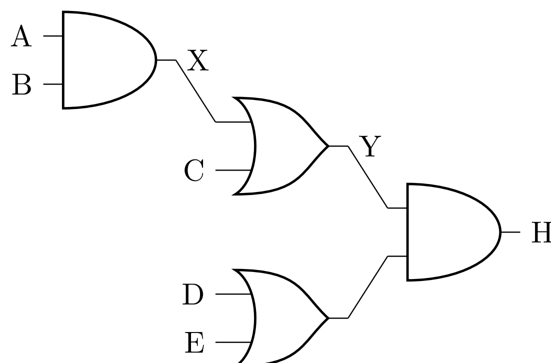

Figure 1: Example inferential circuit

Listing 3.1: QRBS creation code

```
# Create QRBS
system = QRBS()

# Assert precedent facts
a = system.assert_fact('a', 1.0, 0.8)
b = system.assert_fact('b', 1.0, 0.3)
c = system.assert_fact('c', 1.0, 0.5)
d = system.assert_fact('d', 1.0, 0.7)
e = system.assert_fact('e', 1.0, 0.4)

# Assert consequent facts (their imprecision comes from executing the system)
x = system.assert_fact('x', 1.0)
y = system.assert_fact('y', 1.0)
h = system.assert_fact('h', 1.0)

# Assert rules
rule_1 = system.assert_rule(AndOperator(a, b), x, 0.6)
rule_2 = system.assert_rule(OrOperator(x, c), y, 0.2)
rule_3 = system.assert_rule(AndOperator(y, OrOperator(d, e)), h, 0.9)

# Assert island
island = system.assert_island([rule_1, rule_2, rule_3])
```

3.2.2. QRBS Evaluation

The next step is to evaluate the created QRBS against a specific QPU in order to check its fitting (whether a QRBS can be executed in a specific QPU). This process is carried on inside the execution too, but we provide it individually as it can be useful in several scenarios (for example, a user creates a QRBS dynamically and wants to check whether the system fits a specific QPU at any point). This evaluation can be by default (evaluate every knowledge island) or specify which knowledge islands to evaluate. Listing 3.2 continues the code example.

Listing 3.2: QRBS evaluation code

```
try:
    MyQlmQPU.evaluate(system)
except ValueError, err:
    print('System evaluation failed: ', err)
```

© NEASQC Consortium Partners. All rights reserved.

3.2.3. QRBS Execution

Finally, the QRBS is executed in a specific QPU. This execution can be by default (execute every knowledge island) or specify which knowledge islands to execute. The same strategy will be followed for the evaluation carried on within the execution process. Once the execution is complete, the imprecision values of the consequents' facts can be checked by simply accessing it like a usual attribute of an object. Listing 3.3 shows the reading of the imprecision of hypothesis H of the knowledge island.

Listing 3.3: QRBS execution code

```
try:
    MyQlmQPU.execute(system)
    print(h.imprecision) # Outputs the corresponding value, between 0 and 1
except ValueError, err:
    print('System evaluation failed: ', err)
```

3.3. Code Documentation

We provide source documentation for the library, which is available at https://neasqc.github.io/qrbs/neasqc_qrbs.html.

This documentation is autogenerated from PyDoc strings from the source files of the library (available at https://github.com/neasqc/qrbs), and it is updated every time a new version is made available in the **main** branch.

So far, it illustrates the software specifics of classes, attributes, methods and exceptions, but we intend to complete it in future versions with more details on how it works, as well as commented examples regarding its use, such as the one illustrated in this deliverable.

Appendix A presents the software library documentation up to this point.

© NEASQC Consortium Partners. All rights reserved.

4. IDC Application Specification

In this section we present the application of quantum computing techniques for the staging of Invasive Ductal Carcinoma (IDC) of the breast. It includes: (1) a brief explanation of a classical, and well-established, approach for medical reasoning, (2) a description of the clinical problem, (3) a conceptual model for staging invasive ductal carcinoma, and (4) a step-by-step explanation of the proposed approach for quantum staging of the invasive ductal carcinoma. This work has been published as a preprint in (Moret-Bonillo et al., 2023) and it is currently under review for its publication in a scientific journal.

4.1. Introduction

4.1.1. Categorical reasoning in medicine

One of the very first papers about artificial intelligence reasoning in medicine was the one published by Ledley and Lusted in 1959 titled "Reasoning Foundations of Medical Diagnosis" (Ledley & Lusted, 1959). There, the authors explore the challenges of medical diagnosis and propose the use of computers to simplify the process. However, during that time, computer capabilities were limited, which restricted the size of the problems they could handle.

Ledley and Lusted identify three key factors in diagnostic tasks: medical knowledge, observed symptoms, and possible diagnoses consistent with the symptoms and physician's knowledge. They approached the problem from the perspective of differential diagnosis, aiming to eliminate incompatible associations between symptoms and diagnoses based on the knowledge domain. They create an Expanded Logic Base (ELB) by taking the Cartesian product of associations between symptoms and diagnoses. The ELB contains all possible combinations, but some are eliminated based on medical knowledge to obtain a Reduced Logic Base (RLB). The RLB only includes associations that are compatible with the available knowledge.

Although their procedure was categorical, with no imprecision or uncertainty considered, uncertainty appears spontaneously and naturally because it is inherent in human reasoning due to several factors: the nature of heuristic knowledge, inaccuracies, lack of knowledge, and subjectivity in information interpretation.

4.1.2. Description of the clinical problem

A more complete description of the clinical problem can be found in deliverable "*D6.2: QRBS models, architecture and formal specification*" (Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021), but we include here a brief summary as a reminder.

Invasive Ductal Carcinoma, sometimes referred to as infiltrating ductal carcinoma, is the most common type of breast cancer. About 80% of all cases of breast cancer are invasive ductal carcinomas.

Staging is the process used to estimate the extent of invasive ductal carcinoma spread from its original location. The stage of invasive ductal carcinoma is described on a scale from stage I (the earliest stage) to stage IV (the most advanced stage) (Amin et al., 2017).

Stage I describes invasive breast cancer (cancer cells take in or invade the normal breast tissue around them). Stage I is divided into subcategories, known as I-A and I-B. Stage I-A describes invasive breast cancer in which the tumour is up to 2 cm and no lymph nodes are affected. Stage I-B describes invasive breast cancer in which: (1) there is no tumour in the breast, (2) small groups of cancer cells greater than 0.2 mm but less than 2 mm are observed in the lymph nodes, or (3) there is a breast tumour smaller than 2 cm and small groups of cancer cells larger than 0.2 mm but smaller than 2 mm in the lymph nodes.

Stage II is divided into subcategories II-A and II-B. Stage II-A describes invasive breast cancer in which: (1) there is no tumour in the breast, but cancer cells are found in 1-3 axillary lymph nodes under the arm or in lymph nodes near the breastbone, or (3) the tumour is 2 cm or smaller and has spread to the axillary lymph nodes, or (4) the tumour is 2 to 5 cm and has not spread to the axillary lymph nodes. Stage II-B describes invasive breast cancer in which: (1) the tumour is between 2 and 5 cm, and small groups of cancer cells larger than 0.2 mm but smaller than 2 mm are seen in the lymph nodes, or (2) the tumour is 2 to 5 cm, and the cancer has spread to 1-3 axillary lymph nodes or lymph nodes near the breastbone, or (3) the tumour is larger than 5 cm but has not spread to the axillary lymph nodes.

Stage III is divided into subcategories III-A, III-B, and III-C. Stage III-A describes invasive breast cancer in which: (1) the tumour may be any size, and cancer was found in 4-9 axillary lymph nodes or lymph nodes near the breastbone, or

© NEASQC Consortium Partners. All rights reserved.

(2) the tumour is larger than 5 cm, and small clusters of cancer cells larger than 0.2 mm but smaller than 2 mm are seen in the lymph nodes, or (3) the tumour is larger than 5 cm, and the cancer has spread to 1-3 axillary lymph nodes or lymph nodes near the breastbone. Stage III-B describes invasive breast cancer in which the tumour is indefinite in size and has spread to the chest wall or skin of the breast. Stage III-C describes invasive breast cancer in which: (1) there may be no evidence of disease in the breast or, (2) if a tumour is present, it may be any size and may have spread to the chest wall or skin of the breast and cancer has spread to 10 or more axillary lymph nodes, or (3) cancer has spread to lymph nodes above or below the collarbone or cancer has spread to axillary lymph nodes or lymph nodes near the breastbone.Stage IV describes invasive breast cancer that has spread beyond the breast and surrounding lymph nodes to other organs in the body, such as the lungs, distant lymph nodes, skin, bones, liver, and brain.

4.2. A model for the staging of IDC

Knowledge engineering is a sub-field of artificial intelligence whose purpose is the design and development of expert systems. This is supported by instructional methodologies, trying to represent the human knowledge and reasoning in a certain domain, within an artificial system. The work of knowledge engineers consists of extracting the knowledge of human experts, and in coding said knowledge so that it can be processed by a computer system. The problem is that the knowledge engineer is not an expert in the field that tries to model, while the expert in the subject has no experience modelling his/her knowledge (based on heuristics) in a way that can be represented generically in the computer system (Waterman, 1985).

Knowledge engineering encompasses the scientists, technology and methodology required to process knowledge. The goal is to extract, articulate, and computerize knowledge from an expert. The result is a knowledge model, ready to be implemented and tested. In this case, we use the method provided by the TNM staging system (Giuliano et al., 2018) to describe the amount and spread of cancer in a patient's body, where:

- T describes the size of the tumour and any spread of cancer into nearby tissue.
- N describes spread of cancer to nearby lymph nodes.
- M describes metastasis (spread of cancer to other parts of the body).

Each of these categories is furtherly detailed with numbers or letters, differentiating the severeness of the case. We use a reduced version of this system, only employing the numbers notation and using X/Y to denote that any number would fit. Figure 2 illustrates the variables we are considering in our knowledge model for staging IDC, and Table 1 shows the correspondence between TNM classifications and IDC stages.

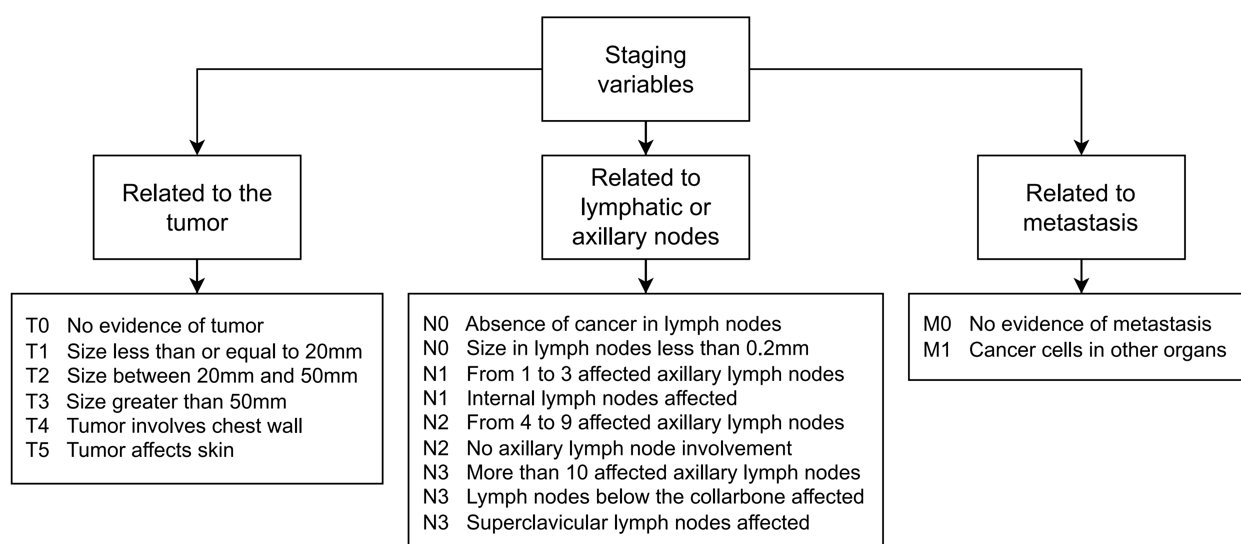

Figure 2: Variables or symptoms that are considered in the knowledge model for IDC staging.

© NEASQC Consortium Partners. All rights reserved.

Table 1: Invasive Ductal Carcinoma stages according with TNM classification system.

IDC Stage	Compatible TNM classification
I-A	T1 N0 M0
I-B	T0 N1 M0 / T1 N1 M0
II-A	T0 N1 M0 / T1 N1 M0 / T2 N0 M0
II-B	T2 N1 M0 / T3 N0 M0
III-A	T0 N2 M0 / T1 N2 M0 / T2 N0 M0 / T3 N2 M0 / T3 N1 M0
III-B	T4 N0 M0 / T4 N1 M0 / T4 N2 M0
III-C	TX N3 M0
IV	TX NY M1

4.3. Quantum system design for breast IDC staging

The next step is to represent our model for the stagin of IDC as a Quantum Rule-Based Systems (QRBS). A QRBS is a Rule-Based Systems (RBS) that uses the formalism of Quantum Computing (QC) for representing knowledge and for making inferences (Moret-Bonillo, Magaz-Romero, et al., 2022; Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021).

We have to consider the following set of assumptions and domain restrictions, which will condition the design of the proposed classic-quantum hybrid solution for breast IDC staging: (1) the participation and collaboration of clinical experts is essential, (2) the magnitude of the problem is limited by the small number of qubits from the NISQ era of Quantum Computing, which requires us to optimize the overall process by reducing as much as possible the number of qubits required , (3) the final solution must be neither pure quantum nor pure classical. It must be a hybrid solution due to the interactions with the experts, and (4) the results obtained must be comparable to what a human expert would obtain. This framework can be represented as shown in Figure 3.

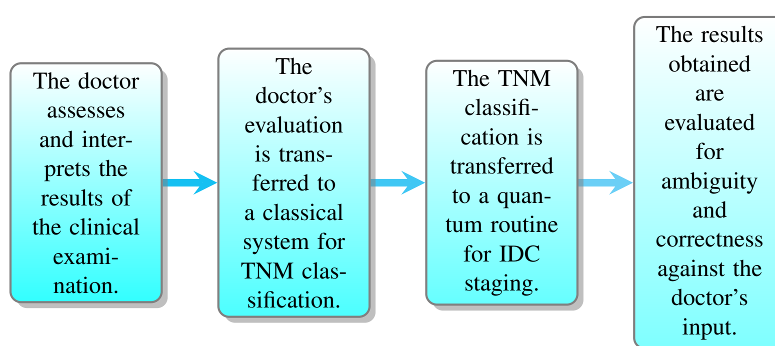

Figure 3: Framework for Beast Invasive Ductal Carcinoma Staging.

Next, and considering the knowledge model detailed in Figure 2 and Table 1 , we will focus on the description of the quantum routine for IDC staging. If we consider the TNM staging system, and in reference to the categorial reasoning notation introduced in previous sections, notice that for this case $S = \{T0, T1, T2, T3, T4, T5, N0, N1, N2, N3, M0, M1\}$ and $D = \{I\text{-}A, I\text{-}B, II\text{-}A, II\text{-}B, III\text{-}A, III\text{-}B, III\text{-}C, IV\}$. Applying now the corresponding medical reasoning model, only the complexes shown in Table 2 are relevant. For each of these relevant complexes, which can be considered as vectors, a qubit will be associated to it, which will be the input to the quantum subroutine. Since a given patient cannot be in two different TNM states at the same time, the quantum subroutine is activated only when one and only one of the input qubits is in state 1, and all the others are in state 0.

© NEASQC Consortium Partners. All rights reserved.

Table 2: Relevant TNM states and corresponding input qubits to the quantum system.

TNM	Input qubit														
	q0	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
T0	X	X													
T1			X	X	X										
T2						X	X								
T3								X	X	X					
T4											X	X	X		
N0			X			X		X			X				
N1	X			X			X		X			X			
N2		X			X					X			X		
N3														X	
M0	X	X	X	X	X	X	X	X	X	X	X	X	X		
M1															X

On the other hand, we must design our quantum routine in such a way that—after the measurement process—we obtain a string of conventional bits that can be directly associated with the stage of the cancer. Also, since there are eight possible stages of cancer, we need an eight-bit string in the output. Table 3 illustrates the matching between the output bit string and the different stages of cancer.

Table 3: Correspondence between the output bits and IDC stage.

Output Bit	c7	c6	c5	c4	c3	c2	c1	c0
IDC Stage	IV	III-C	III-B	III-A	II-B	II-A	I-B	I-A

An added problem is that, according to what is stated in Table 1, in each specific case the same TNM classification may be compatible with several different stages of cancer. This fact must be considered in the design of our quantum routine. This implies defining and implementing a set of quantum rules (relating the TNM classification input to the IDC staging output) and configuring the inferential circuits that represent the different possible cases. For this we need to use an extra set of qubits on which the logical operations imposed by the quantum rules are carried out. In our case, we are going to need 10 extra qubits to solve the defined inferential circuits . Thus, for the quantum resolution of the IDC staging, we need: (a) 15 input qubits to set up the problem, (b) 10 extra qubits to perform the logical operations imposed by the inferential circuits, and (c) a string of 8 conventional bits, on which -after the corresponding measurements- the results obtained by the quantum subroutine will be written. The resulting architecture of the quantum subroutine is illustrated in Figure 4.

© NEASQC Consortium Partners. All rights reserved.

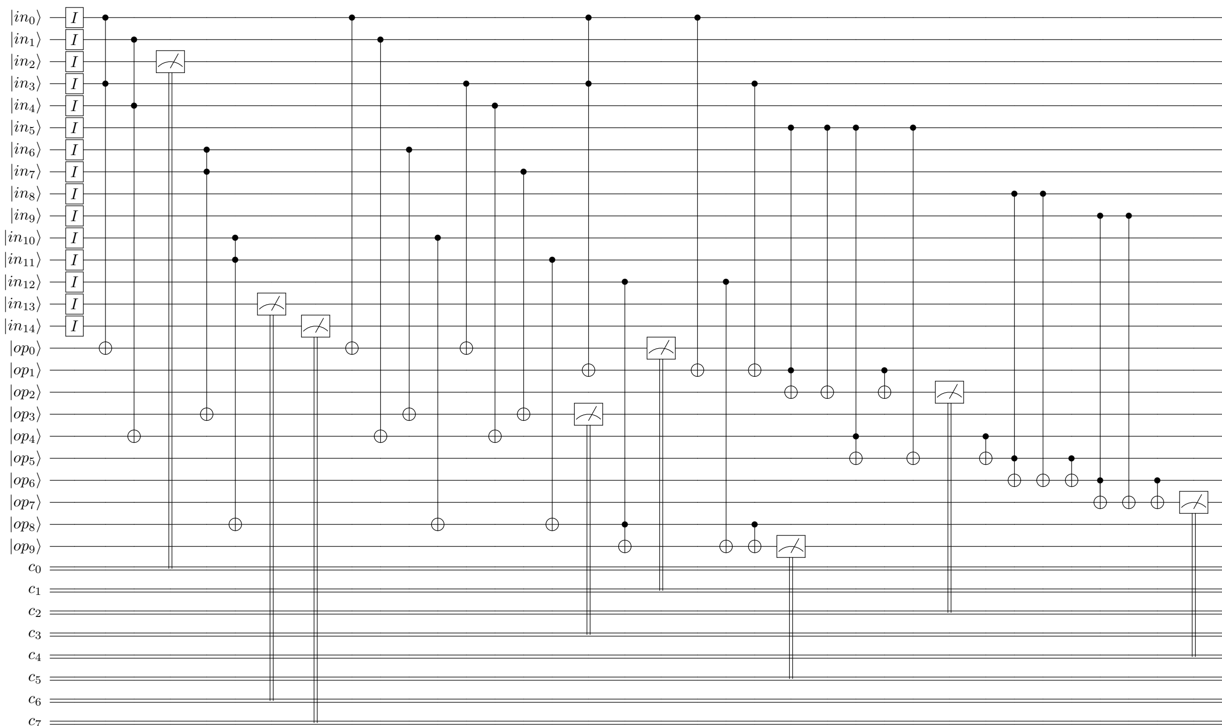

Figure 4: *Architecture of the Quantum Subroutine for Breast Invasive Ductal Carcinoma Staging.*

The quantum circuit works as follows: note that, conventionally, the initial state of any qubit is always 0. To activate it, one -and only one- of the input qubits must be “negated” (set to state 1) and the quantum system begins to solve the corresponding quantum circuits to finally obtain the string of conventional bits, which allows us to identify the IDC stages that are compatible with the input information. For this quantum subroutine, there are 15 different inputs that can activate it, one for each individual qubit.

© NEASQC Consortium Partners. All rights reserved.

5. Testing

In this chapter, we continue the work on testing that we have been developing in previous deliverables. We follow up with the traceability of the use cases and the tests of their scenarios, give some insight into the testing methodology and tools employed and provide some test examples.

5.1. Traceability with use cases

Following the traceability hierarchy established in (Moret-Bonillo, Mosqueira-Rey, & Magaz-Romero, 2021) (see Figure 1, page 8), we defined the tests of Table 4 (this table is extracted from (Moret-Bonillo, Gomez Tato, et al., 2022)).

Test case ID	Test method	Test class
T-01-1	`test_fact_creation`	`TestFact`
T-01-2	`test_fact_deletion`	`TestFact`
T-01-3	`test_fact_modification`	`TestFact`
T-02-1	`test_rule_creation`	`TestRule`
T-02-2	`test_rule_deletion`	`TestRule`
T-02-3	`test_rule_modification`	`TestRule`
T-03-1	`test_island_creation`	`TestKnowledgeIsland`
T-03-2	`test_island_deletion`	`TestKnowledgeIsland`
T-03-3	`test_island_modification`	`TestKnowledgeIsland`
T-03-4	`test_island_creation_error`	`TestKnowledgeIsland`
T-04-1	`test_imprecision`	`TestUncertainty`
T-04-2	`test_uncertainty`	`TestUncertainty`
T-04-3	`test_imprecision_uncertainty`	`TestUncertainty`
T-05-1	`test_qrbs_creation`	`TestQRBS`
T-05-2	`test_qrbs_deletion`	`TestQRBS`
T-05-3	`test_qrbs_modification`	`TestQRBS`
T-06-1	`test_positive_evaluation`	`TestEvaluation`
T-06-2	`test_negative_evaluation`	`TestEvaluation`
T-07-1	`test_successful_default`	`TestExecution`
T-07-2	`test_failed_default_evaluation`	`TestExecution`
T-07-3	`test_failed_default_execution`	`TestExecution`
T-07-4	`test_successful_specified`	`TestExecution`
T-07-5	`test_failed_specified_evaluation`	`TestExecution`
T-07-6	`test_failed_specified_execution`	`TestExecution`

***Table 4**: Test suite*

These tests take into consideration several possible scenarios: (a) categorical quantum reasoning with no imprecision nor uncertainty, (b) quantum imprecision in facts, (c) quantum uncertainty propagation, and (d) both quantum imprecision and quantum uncertainty.

As we presented in Chapter 3, tests are defined inside the **tests** directory, where we can find a testing module per source code module. Therefore, the tests classes from Table 4 are organized as follows:

- **test_knowledge_rep.py**: TestFact, TestRule, TestKnowledgeIsland, TestUncertainty.
- **test_qrbs.py**: TestQRBS, TestEvaluation, TestExecution.

To these tests, we have added also the test classes for the working memory and inference engine classes. These are TestWorkingMemory and TestInferenceEngine, and can be found in **test_qrbs.py** as it is their corresponding module.

While neither TestEvaluation and TestExecution were specifically defined to test any QPU in particular, we have used the MyQlmQPU implementation that we previously defined. This means that in future version these tests may change when we implement more QPUs, whether by including them in these test classes or by adding more to test their functionalities.

© NEASQC Consortium Partners. All rights reserved.

5.2. Testing methodology and tools

In this section we go over some aspects about the testing methodology and tools employed: (1) the approach we have followed regarding testing, (2) how we have measured how the tests cover the source code and (3) the testing tools we have used.

5.2.1. Testing approach

According to the testing approach we have followed, test cases represent the requirements that we expect our software to cover. Since we have defined them following the different scenarios of the use cases, most execution flows are covered with these test cases.

This approach provides several advantages when developing our software library:

- **Focus on the observable behavior of the software**: having to develop the code for the tests implies approaching our software from the perspective of the final users, focusing on it from the interface it provides rather than its implementation.
- **Ease to detect errors**: when tests cover most execution flows, errors made when programming are easier to detect, since they will come up when running their corresponding tests.
- **Automation**: having the test suite automated eases the development process overall, since relevant changes on the internal logic or even refactorization will make the tests fail in case they are done incorrectly.

Regarding how *deep* we should test, we have opted to get down to unit tests for our classes. While it may seem excessive to test at this level, we think that it is beneficial; we are developing a software library that will be used by researchers and people conducting different experiments, so it is important to test it deeply and make sure it will not be easy to "break".

5.2.2. Measuring test coverage

There are several measures we can consider when evaluating how good our tests are doing:

- **Quantity**: the more test cases we have, the more trustworthy our tests are.
- **Representation**: test cases must be representative of the different situations that can take place. In order to measure representation, we use test coverage:
  - **Line coverage**: amount of lines of source code that are run with the tests.
  - **Branch coverage**: amount of branches (options of conditional statements) are run with the tests.

The objective behind coverage is to find out which segments of source code are not covered by tests. However, numerical values (e.g. number of lines covered, percentage of branches covered) are not a measure of quality of the tests.

On top of that, tests do not have to cover the complete source code (a coverage of 100% is not mandatory), since they should be focused towards those segments of code that can reasonably fail, and other segments can be "overlooked" if they are too simple to break.

5.2.3. Testing tools

Testing tools vary across programming languages and project paradigms, so it is important to use the tools that better address the needs in each scenario. In our case, we are developing a Python software library, so **pytest** fits our needs perfectly.

The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries, like is our case. It is widely extended across the Python community, and has a rich plugin architecture with over 800+ external plugins, covering mostly any need that can arise when testing our software.

Thanks to the auto-discovery of test modules and functions, testing our software with pytest is very simple. As we presented on chapter 3, we have all our tests inside the corresponding repository. Then, when invoking pytest, it will search in those files for classes and methods whose names begin with "test" and run them. Inside these tests we use

© NEASQC Consortium Partners. All rights reserved.

our software as required, following the test cases, and use the `assert` directive to check the desired values. We can also catch the exceptions to test them, and assert the information they provide.

For coverage, we use the **pytest-cov** plugin, which provides a clean minimal set of command line options that are added to pytest. These options add the coverage information to the original pytest report, providing information like lines and branches covered as desired. Moreover, pytest-cov reports can be exported to different formats, and the HTML format presents the source code with the coverage information highlighted, which is really useful when analyzing the tests overall.

5.3. Test example

We provide an example of the tests to illustrate how they work. In this case, Listing 5.1 shows the test `test_qrbs_modification`, since it is a very complete case of testing different values and exceptions.

Listing 5.1: QRBS modification test

```
def test_qrbs_modification(self):
    """
    Test QRBS modification
    """
    system = QRBS()

    # Modify the system by asserting several elements
    fact_1 = system.assert_fact('fact_1', 0.8)
    fact_2 = system.assert_fact('fact_2', 0.3)
    fact_3 = system.assert_fact('fact_3', 0.5)
    rule_1 = system.assert_rule(fact_1, fact_2)
    rule_2 = system.assert_rule(fact_2, fact_3)
    island_1 = system.assert_island([rule_1])
    island_2 = system.assert_island([rule_1, rule_2])
    assert system._memory._facts == [fact_1, fact_2, fact_3]
    assert system._engine._rules == [rule_1, rule_2]
    assert system._engine._islands == [island_1, island_2]

    # Modify the system by retracting an existing knowledge island
    system.retract_island(island_2)
    assert system._engine._islands == [island_1]

    # Modify the system by retracting an existing rule
    system.retract_rule(rule_2)
    assert system._engine._rules == [rule_1]

    # Modify the system by retracting an existing fact
    system.retract_fact(fact_3)
    assert system._memory._facts == [fact_1, fact_2]

    # Raises an error due to retracting a rule that is part of a knowledge
        island
    with pytest.raises(AttributeError) as ex_info:
        system.retract_rule(rule_1)
    assert ex_info.match('The rule to be retracted is part of a knowledge
        island and cannot be retracted')

    # Raises an error due to retracting a fact that is part of a rule
    with pytest.raises(AttributeError) as ex_info:
        system.retract_fact(fact_1)
    assert ex_info.match('The fact to be retracted is part of a rule and
        cannot be retracted')
```

© NEASQC Consortium Partners. All rights reserved.

This test checks several operations:

- Creates a QRBS and populates it.
- Asserts the elements of the QRBS have been correctly initialized.
- Retracts several items and asserts they are no longer part of the QRBS.
- Attempts to retract several items that cannot be retracted and asserts the exceptions risen.

This approach (initializing items, asserting their values, operating with the items, asserting their new values or the exceptions) is the same for the rest of the tests implemented, which can be found in https://github.com/NEASQC/qrbs/tree/main/tests.

© NEASQC Consortium Partners. All rights reserved.

6. Conclusions

In this deliverable we have presented the preliminary version of the QRBS software library. While most of the effort has been dedicated to the library's repository and the implementation of its functionalities, we have taken the opportunity to present here how it is structured and how to make use of it so far. Following deliverables will continue this work, presenting the different additions we include in our QRBS software.

Both the software library and its documentation, we have developed them with the user in mind, focusing on conciseness and clarity, in order to make it as easy to use as possible. We aim to keep next developments along this line, as we consider it a priority to introduce users to what we consider a powerful and useful tool.

This approach is supported by the focus on traceability and testing that has been carried from the beginning of the project, which has seen a major payout in this deliverable. We achieve this by covering every possible scenario so that the final product is of the maximum quality.

On the IDC application, we have made an effort on using categorical reasoning and the description of the clinical problem that we introduced in our earliest work to build a model for staging IDC. Moreover, we have taken that model and applied it into a quantum routine with great success. This effort will allow us to combine the research of breast cancer diagnosis with our methodology of QRBS, to obtain in the end a system programmed with our software library for the diagnosis of IDC, which is the final objective of the use case.

In summary, this report represents an important achievement in the work of this use case, starting both the final phase of the development of the QRBS software library and the development of the IDC application.

© NEASQC Consortium Partners. All rights reserved.

List of Acronyms

Term	Definition
AI	Artificial Intelligence
ELB	Expanded Logic Base
IDC	Invasive Ductal Carcinoma
QC	Quantum Computing
QRBS	Quantum Rule-Based System
RBS	Rule-Based System
RLB	Reduced Logic Base

Table 5: Acronyms and Abbreviations

© NEASQC Consortium Partners. All rights reserved.

List of Figures

Figure 1.: Example inferential circuit 7

Figure 2.: Variables or symptoms that are considered in the knowledge model for IDC staging. 10

Figure 3.: Framework for Beast Invasive Ductal Carcinoma Staging. 11

Figure 4.: Architecture of the Quantum Subroutine for Breast Invasive Ductal Carcinoma Staging. 13

© NEASQC Consortium Partners. All rights reserved.

List of Tables

Table 1.: Invasive Ductal Carcinoma stages according with TNM classification system. 11
Table 2.: Relevant TNM states and corresponding input qubits to the quantum system. 12
Table 3.: Correspondence between the output bits and IDC stage. 12

Table 4.: Test suite 14

Table 5.: Acronyms and Abbreviations 19

© NEASQC Consortium Partners. All rights reserved.

Bibliography

Amin, M. B., Greene, F. L., Edge, S. B., Compton, C. C., Gershenwald, J. E., Brookland, R. K., Meyer, L., Gress, D. M., Byrd, D. R., & Winchester, D. P. (2017). The eighth edition ajcc cancer staging manual: Continuing to build a bridge from a population-based to a more "personalized" approach to cancer staging. *CA: A Cancer Journal for Clinicians*, *67*(2), 93–99. https://doi.org/10.3322/caac.21388

Giuliano, A. E., Edge, S. B., & Hortobagyi, G. N. (2018). Eighth edition of the ajcc cancer staging manual: Breast cancer. *Annals of Surgical Oncology*, *25*(7), 1783–1785. https://doi.org/10.1245/s10434-018-6486-6

Kruchten, P. (2004). *The rational unified process: An introduction*. Addison-Wesley.

Ledley, R. S., & Lusted, L. B. (1959). Reasoning foundations of medical diagnosis. *Science*, *130*(3366), 9–21. https://doi.org/10.1126/science.130.3366.9

Moret-Bonillo, V., Gomez Tato, A., Magaz Romero, S., Mosqueira-Rey, E., & Alvarez-Estevez, D. (2022). D6.9: Qrbs software specifications. https://doi.org/10.5281/zenodo.7299193

Moret-Bonillo, V., Magaz-Romero, S., & Mosqueira-Rey, E. (2022). Quantum computing for dealing with inaccurate knowledge related to the certainty factors model. *Mathematics*, *10*(2). https://doi.org/10.3390/math10020189

Moret-Bonillo, V., Mosqueira-Rey, E., & Magaz-Romero, S. (2021). D6.5 Quantum Rule-Based System (QRBS) Requirement Analysis. https://doi.org/10.5281/zenodo.5949157

Moret-Bonillo, V., Mosqueira-Rey, E., Magaz-Romero, S., & Alvarez-Estevez, D. (2023). Hybrid classic-quantum computing for staging of invasive ductal carcinoma of breast. *arXiv preprint arXiv:2303.10142*. https://arxiv.org/abs/2303.10142

Moret-Bonillo, V., Mosqueira-Rey, E., Magaz-Romero, S., & Gómez-Tato, A. (2021). Quantum rule-based systems (qrbs) models, architecture and formal specification (D6. 2). https://www.neasqc.eu/wp-content/uploads/2021/05/NEASQC_D6.2_QRBS-Models-Architecture-and-Formal-Specification-V1.5-Final.pdf

Waterman, D. A. (1985). *A guide to expert systems*. Addison-Wesley Longman Publishing Co., Inc.

© NEASQC Consortium Partners. All rights reserved.

A. QRBS software documentation

This appendix presents the software library documentation up to the point of development when this document is published. The current version of the documentation can be found in https://neasqc.github.io/qrbs/neasqc_qrbs.html.

A.1. neasqc_qrbs.knowledge_rep module

***class* neasqc_qrbs.knowledge_rep.AndOperator(*left_child*, *right_child*)** Bases: `neasqc_qrbs.knowledge_rep.LeftHandSide`

Class representing an AndOperator.

An AndOperator relates the statements of its children with an AND relationship. This class is used to model the Composite design pattern, acting as (one of) the Composite class.

left_child One of the children which is relating.

Type `LeftHandSide`

right_child One of the children which is relating.

Type `LeftHandSide`

build(*builder*) → QRoutine

***class* neasqc_qrbs.knowledge_rep.Buildable** Bases: `abc.ABC`

Interface for knowledge elements that can be built into quantum routines.

***abstract* build(*builder*) → QRoutine**

***class* neasqc_qrbs.knowledge_rep.Builder** Bases: `abc.ABC`

Interface for building the corresponding quantum routine from a Buildable element.

***abstract static* build_and() → QRoutine** Builds the quantum routine of an and operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

***abstract static* build_fact(*fact*) → QRoutine** Builds the quantum routine of a fact.

Parameters **fact** (`Fact`) – The Fact whose quantum routine is being built.

Returns The corresponding quantum routine.

Return type `QRoutine`

Parameters **island** (`KnowledgeIsland`) – The KnowledgeIsland whose quantum routine is being built.

Returns A tuple containing the corresponding quantum routine and the index of which qubit corresponds to each LeftHandSide element.

Return type Tuple[`QRoutine`, Dict[`LeftHandSide`, int]]

***abstract static* build_not() → QRoutine** Builds the quantum routine of a not operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

***abstract static* build_or() → QRoutine** Builds the quantum routine of an or operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

***abstract static* build_rule(*rule*) → QRoutine** Builds the quantum routine of a rule.

© NEASQC Consortium Partners. All rights reserved.

Parameters **rule** (`Rule`) – The Rule whose quantum routine is being built.

Returns The corresponding quantum routine.

Return type `QRoutine`

class **neasqc_qrbs.knowledge_rep.BuilderImpl** Bases: `neasqc_qrbs.knowledge_rep.Builder`

Implementation of Builder interface.

static **build_and()** → **QRoutine** Builds the quantum routine of an and operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

static **build_fact(*fact*)** → **QRoutine** Builds the quantum routine of a fact.

Parameters **fact** (`Fact`) – The Fact whose quantum routine is being built.

Returns The corresponding quantum routine.

Return type `QRoutine`

static **build_island(*island*)** → **Tuple[QRoutine, Dict[neasqc_qrbs.knowledge_rep.LeftHandSide, int]]**
Builds the quantum routine of a knowledge island.

Parameters **island** (`KnowledgeIsland`) – The KnowledgeIsland whose quantum routine is being built.

Returns A tuple containing the corresponding quantum routine and the index of which qubit corresponds to each LeftHandSide element.

Return type Tuple[`QRoutine`, Dict[`LeftHandSide`, int]]

static **build_not()** → **QRoutine** Builds the quantum routine of a not operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

static **build_or()** → **QRoutine** Builds the quantum routine of an or operator.

Returns The corresponding quantum routine.

Return type `QRoutine`

static **build_rule(*rule*)** → **QRoutine** Builds the quantum routine of a rule.

Parameters **rule** (`Rule`) – The Rule whose quantum routine is being built.

Returns The corresponding quantum routine.

Return type `QRoutine`

class **neasqc_qrbs.knowledge_rep.Fact(*attribute*, *value*, *imprecission=0.0*)** Bases: `neasqc_qrbs.knowledge_rep.LeftHandSide`

Class representing a Fact.

A Fact is the smallest unit of knowledge that can be represented. This class is used to model the Composite design pattern, acting as the Leaf class.

attribute Attribute that the fact is representing.

Type str

value Value of the attribute that the fact is representing.

Type float

imprecission Imprecission of the fact; the certainty of the attribute having said value (0 if not specified). Must be in range [0,1].

Type float, optional

© NEASQC Consortium Partners. All rights reserved.

build(*builder*) → QRoutine

***property* imprecission**

***class* neasqc_qrbs.knowledge_rep.KnowledgeIsland(*rules*)** Bases: `neasqc_qrbs.knowledge_rep.Buildable`

Class representing a Knowledge Island.

A Knowledge Island is a set of rules that conform the inferential reasoning towards a hypothesis.

rules Set of rules that conform the knowledge island.

Type List[`Rule`]

build(*builder*) → QRoutine

***class* neasqc_qrbs.knowledge_rep.LeftHandSide** Bases: `neasqc_qrbs.knowledge_rep.Buildable`

Interface for elements that can be part of the left hand side of a rule. This class is used to model the Composite design pattern, acting as the Component interface.

build() → QRoutine

***class* neasqc_qrbs.knowledge_rep.NotOperator(*child*)** Bases: `neasqc_qrbs.knowledge_rep.LeftHandSide`

Class representing a NotOperator.

A NotOperator negates the statement of its child. This class is used to model the Composite design pattern, acting as (one of) the Composite class.

child Child which statement is negating.

Type `LeftHandSide`

build(*builder*) → QRoutine

***class* neasqc_qrbs.knowledge_rep.OrOperator(*left_child*, *right_child*)** Bases: `neasqc_qrbs.knowledge_rep.LeftHandSide`

Class representing an OrOperator.

An OrOperator relates the statements of its children with an OR relationship. This class is used to model the Composite design pattern, acting as (one of) the Composite class.

left_child One of the children which is relating.

Type `LeftHandSide`

right_child One of the children which is relating.

Type `LeftHandSide`

build(*builder*) → QRoutine

***class* neasqc_qrbs.knowledge_rep.Rule(*lefthandside*, *righthandside*, *uncertainty=0.0*)** Bases: `neasqc_qrbs.knowledge_rep.Buildable`

Class representing a Rule.

A Rule which establishes a relationship (to some level of uncertainty) between a left hand side element and a right hand side, which in this context is a Fact.

leftHandSide Left hand side element of the rule (also known as precedent).

Type `LeftHandSide`

rightHandSide Right hand side element of the rule (also known as consecuent).

Type `Fact`

uncertainty Uncertainty of the relationship between precedent and consecuent (0 if not specified). Must be in range [0,1].

Type float, optional

© NEASQC Consortium Partners. All rights reserved.

build(*builder*) → **QRoutine**

***property* uncertainty**

A.2. neasqc_qrbs.qrbs module

***class* neasqc_qrbs.qrbs.InferenceEngine(*rules=[]*, *islands=[]*)** Bases: `object`

Class representing an Inference Engine.

An Inference Engine is an element of a Rule-Based System that manages its rules and knowledge islands, providing the tools to evaluate them in order.

_rules List of rules established for the system.

Type List[`Rule`], optional

_islands List of knowledge island established for the system.

Type List[`KnowledgeIsland`], optional

assert_island(*island*) → **neasqc_qrbs.knowledge_rep.KnowledgeIsland** Asserts a knowledge island into the engine.

Parameters **island** (`KnowledgeIsland`) – The knowledge island to be asserted.

Returns The asserted knowledge island.

Return type `KnowledgeIsland`

Raises **AttributeError** – In case the rules that compose the knowledge island are not asserted in the system's inference engine or the rules that compose the knowledge island are not chained.

assert_rule(*rule*) → **neasqc_qrbs.knowledge_rep.Rule** Asserts a rule into the engine.

Parameters **rule** (`Rule`) – The rule to be asserted.

Returns The asserted rule.

Return type `Rule`

retract_island(*island*) → **None** Retracts a knowledge island from the engine.

Parameters **island** (`KnowledgeIsland`) – The knowledge island to be retracted.

retract_rule(*rule*) → **None** Retracts a rule from the engine.

Parameters **rule** (`Rule`) – The rule to be retracted.

Raises **AttributeError** – In case the rule to be retracted is part of a knowledge island.

***class* neasqc_qrbs.qrbs.MyQlmQPU** Bases: `neasqc_qrbs.qrbs.QPU`

myQLM implementation of a Quantum Processing Unit (QPU).

MAX_ARITY *= 20*

***static* evaluate(*qrbs*, *eval_islands=[]*)** → **bool** Evaluates whether a QRBS can be executed on this QPU.

Parameters

- **qrbs** (`QRBS`) – The QRBS to be evaluated.
- **eval_islands** (List[`KnowledgeIsland`], optional) – A list of specific KnowledgeIsland to be evaluated.

Raises **ValueError** – In case an specified knowledge island is not part of the QRBS or an evaluated knowledge island requires more qubits than supported.

***static* execute(*qrbs*, *eval_islands=[]*)** → **None** Executes the QRBS on this QPU.

Parameters

© NEASQC Consortium Partners. All rights reserved.

- **qrbs** (`QRBS`) – The QRBS to be executed.
- **eval_islands** (List[`KnowledgeIsland`], optional) – A list of specific KnowledgeIsland to be executed.

class **neasqc_qrbs.qrbs.QPU** Bases: `abc.ABC`

Interface defining the structure to implement Quantum Processing Units (QPU).

abstract static **evaluate(*qrbs*)** → **bool** Evaluates whether a QRBS can be executed on this QPU.

Parameters **qrbs** (`QRBS`) – The QRBS to be evaluated.

abstract static **execute(*qrbs*)** → **None** Executes the QRBS on this QPU.

Parameters **qrbs** (`QRBS`) – The QRBS to be executed.

class **neasqc_qrbs.qrbs.QRBS** Bases: `object`

Class representing a Quantum Rule-Based System.

A Quantum Rule-Based System (QRBS) is a Rule-Based System implemented in a quantum computer, taking advatange of some of its capabilities, like quantum superposition, to represent certain aspects such as imprecission and uncertainty.

_memory The Working Memory of the system.

Type `WorkingMemory`

_engine The Inference Engine of the system.

Type `InferenceEngine`

assert_fact(*attribute*, *value*, *imprecission=0.0*) → Creates a fact and asserts it into the system.

Parameters

- **attribute** (*str*) – The attribute of the fact.
- **value** (*float*) – The value of the fact.
- **imprecission** (*float, optional*) – The imprecission of the fact.

Returns The asserted fact.

Return type `Fact`

assert_island(*rules*) → Creates a knowledge island and asserts it into the system.

Parameters **rules** (List[`Rule`]) – The rules of the knowledge island.

Returns The asserted knowledge island.

Return type `KnowledgeIsland`

assert_rule(*lefthandside*, *righthandside*, *uncertainty=0.0*) → Creates a rule and asserts it into the system.

Parameters

- **lefthandside** (`LeftHandSide`) – The left hand side of the rule.
- **righthandside** (`Fact`) – The right hand side of the rule.
- **uncertainty** (*float, optional*) – The uncertainty of the rule.

Returns The asserted rule.

Return type `Rule`

retract_fact(*fact*) → **None** Retracts a fact from the system.

Parameters **fact** (`Fact`) – The fact to be retracted.

retract_island(*island*) → **None** Retracts a knowledge island from the system.

© NEASQC Consortium Partners. All rights reserved.

Parameters **island** (`KnowledgeIsland`) – The knowledge island to be retracted.

retract_rule(*rule*) → **None** Retracts a rule from the system.

Parameters **rule** (`Rule`) – The rule to be retracted.

class **neasqc_qrbs.qrbs.WorkingMemory(*facts=[]*)** Bases: `object`

Class representing a Working Memory.

A Working Memory is an element of a Rule-Based System that manages its facts, keeping trace of their state.

_facts List of facts asserted into the system.

Type List[`Fact`], optional

assert_fact(*fact*) → Asserts a fact into the memory.

Parameters **fact** (`Fact`) – The fact to be asserted.

Returns The asserted fact.

Return type `Fact`

retract_fact(*fact*) → **None** Retracts a fact from the memory.

Parameters **fact** (`Fact`) – The fact to be retracted.

© NEASQC Consortium Partners. All rights reserved.