NExt ApplicationS of Quantum Computing

D6.14: Final QRBS software and IDC application

Document Properties

Contract Number	951821
Contractual Deadline	December-2023
Dissemination Level	Public
Nature	Other
Editors	Vicente Moret-Bonillo, UDC
Authors	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC Diego Álvarez-Estévez, UDC
Reviewers	Mohamed HIBTI, EDF Alfons W. Laarman, ULEI
Date	19-mar-2024
Keywords	software implementation, IDC application, QRBS
Status	Final
Release	1.0

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
0.1	20/11/2023	Samuel Magaz-Romero, UDC	First draft.
0.2	19/01/2024	Samuel Magaz-Romero, UDC	First review.
0.3	12/03/2024	Samuel Magaz-Romero, UDC	Second review.
1.0	19/03/2024	Samuel Magaz-Romero, UDC	Final review.

Table of Contents

1. Executive Summary

This report is the fifth 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 (Moret-Bonillo, Mosqueira-Rey, & Magaz-Romero, 2021; Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021; Moret-Bonillo et al., 2022, 2023).

D6.14 (M40) will include the final version of the RBS and QRBS software along with the IDC application that has been developed.

The report begins with an introduction of the final version of the QRBS software, presenting the different functionalities it provides, as well as the revisions that have taken place regarding the previous work, the different models developed to implement QRBS, and a commentary on the library's documentation current state, which is extended in the appendix.

Following that, we present the implementation of 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 both the QRBS software and the IDC application, which is a critical part on making sure that the developed software projects provide valid and verified functionalities.

Closing the report, we present the conclusions obtained during the elaboration of this work, along with ideas for future work.

2. Context

Some of the concepts referenced in this document have been explained in deliverables D6.2, D6.5, D6.9 and D6.11. 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 final version of the Quantum Rule-Based Systems software library and the implementation 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 adding new functionalities for the final version, while still following the Unified Process of software development (Kruchten, 2004). This final version serves as a milestone regarding the development of the project, by achieving one of the main goals established at its beginning: the development of the QRBS library. There is room yet for more revisions and the addition of new functionalities, but this version satisfies the requirements defined for the software library.

On the other hand, we provide the IDC application implementation, following the current manuals of medicine for cancer staging (Amin et al., 2017). While this work has already been introduced with the knowledge model in the previous deliverables, we provide a brief revision here to clarify its implementation.

2.2. Work package

In the context of the Work Package 6 – “Symbolic AI and graph algorithmics”, this document illustrates the final step (prior to maintenance) of the process that must be followed in order to develop the framework of Quantum Rule-Based Systems, and the implementation details for the IDC application.

In our previous deliverable, “D6.11 Preliminary QRBS software and IDC application specification” (Moret-Bonillo et al., 2023), we provided an initial version that illustrated what the software library for Quantum Rule-Based Systems must achieve. With this version, the implementation stage was integrated into the overall process, providing a basic version of the software which we used to test whether the former steps were well defined. Using that work as a base, we now expand the capabilities of the library to provide the final version, with new models, like fuzzy logic or Bayesian networks, to implement QRBS.

Regarding the IDC application, we provide an implementation to be used for the case of breast cancer diagnosing. This application is implemented using the QRBS software library, and made available through a web API. This approach has been followed in order to facilitate the incorporation of this useful tool in the workflow of users that want to implement it.

The final deliverable will be complementary to this one, since it will provide the evaluation of the work that has been carried out through the previous deliverables, including this one.

3. Final QRBS Software

In this chapter, we continue the work carried out in the previous deliverables regarding the development of the QRBS software library, this time presenting the final version of the software. We divide this chapter in: (1) introducing the functionalities of the software library, (2) presenting the different models implemented for QRBS, and (3) giving a glimpse of its documentation.

3.1. Introduction

For the final version of the QRBS software library, we have focused on providing different models to implement the systems. This effort, along with the work carried out up to this point, has led to the following functionalities:

- **RBS modelling**: through the package `knowledge_rep`, users can model any rule-based system by encoding declarative and procedural knowledge into a system, establishing different facts with their respecting imprecision, and relate them through rules with uncertainty.
- **Automated QRBS implementation in different models**: after defining a RBS, users can employ the `qrbs` package to automatically obtain the corresponding QRBS in the different models available (as we see in Section 3.2) without providing any additional information.
- **Evaluate and execute QRBS**: through the `QPU` classes (for example, the `MyQlmQPU` class provided in the library) users can evaluate their QRBS to analyse whether their system can be run on said QPU, and if so execute it to obtain the corresponding results. The results are encoded as the certainty values of the facts that conform the right-hand side of the rules, also known as consequents, which users can consult after the execution.

With these functionalities, the QRBS software library provides the tools and means for users to work and experiment in the area of Artificial Intelligence and Quantum Computing.

3.1.1. Revised use cases

The addition of new models to implement QRBS's quantum circuits calls for the revision of the use cases involved in this process. Since this scenario was already contemplated (having to revisit any of the previous steps when developing new functionalities or extending the existing ones), few modifications were necessary (Moret-Bonillo, Mosqueira-Rey, & Magaz-Romero, 2021).

In this case, the revised use cases are *UC-06: QRBS evaluation* and *UC-07: QRBS execution*, which are affected by the selection of the model to implement the QRBS in. While the changes are minimal, it is crucial to document them to preserve the traceability of the whole software engineering process. Tables 1 and 2 illustrate the new specification of *UC-06* and *UC-07* respectively.

***Table 1**: Specification of use case UC-06*

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 model to implement the QRBS is selected. 4. The QRBS is evaluated regarding the quantum backend and the model.
Alternative flows	3a. The user selects a model. 1. The QRBS is implemented in the model indicated by the user. 3b. The user selects no model or a model that does not exist. 1. The QRBS is implemented in the default model. 4a. The evaluation returns a positive result. 1. The system displays information of each knowledge island (number of qubits and gates required). 4b. 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 certain operation).
Scenarios	1. Model indicated, positive evaluation: Basic flow, alternative 3a, alternative 4a 2. Model indicated, negative evaluation: Basic flow, alternative 3a, alternative 4b 3. No model indicated/non-existing, positive evaluation: Basic flow, alternative 3b, alternative 4a 4. No model indicated/non-existing, negative evaluation: Basic flow, alternative 3b, alternative 4b
Additional information	

Table 2: Specification of use case UC-07

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 model to implement the QRBS is selected. 4. The QRBS is evaluated: *Includes UC-06: QRBS Evaluation*. 5. The QRBS is executed regarding the quantum backend and the model.
Alternative flows	1a. User does not specify 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. 3a. The user selects a model. 1. The QRBS is implemented in the model indicated by the user. 3b. The user selects no model or a model that does not exist. 1. The QRBS is implemented in the default model. 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 initialised, a runtime error occurred).
Scenarios	1. Default execution, model indicated, successful: Basic flow, alternative 1a, alternative 3a, alternative 3a (UC-06), alternative 4a 2. Default execution, model indicated, failed (because of evaluation): Basic flow, alternative 1a, alternative 3a, alternative 3b (UC-06) 3. Default execution, model indicated, failed (because of execution): Basic flow, alternative 1a, alternative 3a, alternative 3a (UC-06), alternative 4b 4. Specified execution, model indicated, successful: Basic flow, alternative 1b, alternative 3a, alternative 3a (UC-06), alternative 4a 5. Specified execution, model indicated, failed (because of evaluation): Basic flow, alternative 1b, alternative 3a, alternative 3b (UC-06) 6. Specified execution, model indicated, failed (because of execution): Basic flow, alternative 1b, alternative 3a, alternative 3a (UC-06), alternative 4b 7. Default execution, no model indicated/non-existing, successful: Basic flow, alternative 1a, alternative 3b, alternative 3a (UC-06), alternative 4a 8. Default execution, no model indicated/non-existing, failed (because of evaluation): Basic flow, alternative 1a, alternative 3b, alternative 3b (UC-06) 9. Default execution, no model indicated/non-existing, failed (because of execution): Basic flow, alternative 1a, alternative 3b, alternative 3a (UC-06), alternative 4b 10. Specified execution, no model indicated/non-exisiting, successful: Basic flow, alternative 1b, alternative 3b, alternative 3a (UC-06), alternative 4a 11. Specified execution, no model indicated/non-exisiting, failed (because of evaluation): Basic flow, alternative 1b, alternative 3b, alternative 3b (UC-06) 12. Specified execution, no model indicated/non-exisiting, failed (because of execution): Basic flow, alternative 1b, alternative 3b, alternative 3a (UC-06), alternative 4b
Additional information	

3.2. Models

In this section we present the three models of QRBS implementation that have been defined for the final version of the software library. These models are classical-inspired quantum models, and while they provide sufficient requirements to develop QRBS, it is important to keep in mind that some of their aspects may evolve with future research.

3.2.1. Certainty factors

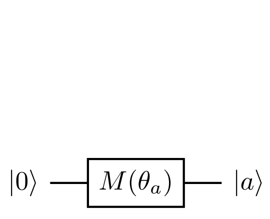
This model is the one initially proposed at the beginning of the project, as the first approach to Quantum Rule-Based Systems (Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021).

Proposed by Shortliffe and Buchanan (Shortliffe & Buchanan, 1975) the certainty factors model shook the foundations of the, at that moment, incipient world of artificial intelligence. It was immediately accepted due to its easy understanding and the quality of the results obtained after its application.

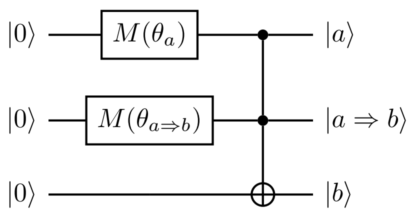
The idea behind certainty factors can be condensed as the fact that, for any hypothesis h, an evidence e cannot simultaneously increase belief and disbelief in h. Therefore, Shortliffe and Buchanan define the Measure of Increasing Belief, $MB(h,e)$, and the Measure of Increasing Disbelief, $MD(h,e)$. This measures are used to give a third index, the Certinaty Factor $CF(h,e) = MB(h,e) - MD(h,e)$.

This new index allows to solve several problems that can be found in RBS, like splitting the implications when several evidences point at the same hypothesis, which can be used to handle the uncertainty found in implication. It appears that, despite its ad hoc nature, probabilities are in the core of the certainty factors (Heckerman, 1986), a fact that inspired its quantum counterpart that we refresh here.

This quantum model provides the M gate (Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021) to encode inaccurate knowledge into the system (mapping their $[0,1]$ value into a $[0,\pi]$ angle rotation), for both the imprecision of the facts and the uncertainty of the rules (see Figure 1).



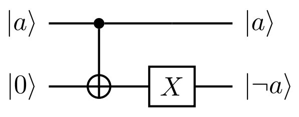
(a) M gate



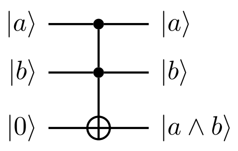
(b) Implication operator

***Figure 1**: Certainty factors' uncertainty managers*

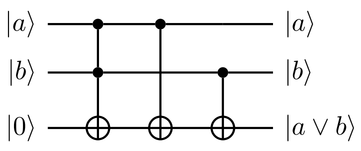
The model provides the quantum equivalent of the logical operators NOT, AND and OR to relate different facts for the precedents of the rules that require them (see Figure 2). Notice how for each of the operators the input qubits are preserved, to be reused if needed, reducing the total amount of qubits for the system. This approach is followed in all of the models presented.



(a) NOT gate



(b) AND gate



(c) OR gate

***Figure 2**: Certainty factors' operators*

Since this model has been under development the longest, and therefore has been tested and experimented with the most, it is the one that we have defined as default for our software library.

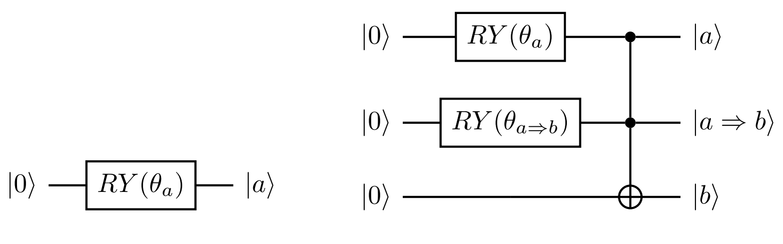
3.2.2. Fuzzy logic

This model implements the quantum operators by using the logical connectives of fuzzy logic as a base.

Fuzzy logic, initially introduced by Lofti A. Zadeh in 1965 (Zadeh, 1965), is an extension of classical logic, where variables can have any value in the range $[0, 1]$ rather than being limited to the binary set. This extension allows for a wider representation of knowledge, as it is sustained by the fact that people do not make decisions categorically, but based on imprecise information.

This new domain for the knowledge variables calls for a revision of the logical connectives: the operators of classical logic are not defined for values different than 0 or 1. However, the newly defined operators for fuzzy variables must preserve the behaviour of their classical counterparts, as fuzzy logic is an extension of classical logic. For our case, we need to redefine the NOT, AND and OR operators.

In the first place, we must model the fuzzy variables in a quantum circuit. In this case, we employ the RY gate, in which we map the value of a fuzzy variable in the range $[0, 1]$ to the rotation range $[0, \pi]$. After applying this gate with the corresponding rotation to any qubit, we will have a qubit in a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = cos(\frac{\theta}{2})|0\rangle + sin(\frac{\theta}{2})|1\rangle$, where β will be used as the representation of the fuzzy value. We use this gate to model both the imprecision of the facts and the uncertainty of the rules (see Figure 3).



(a) RY gate *(b) Implication operator*

***Figure 3**: Fuzzy logic's uncertainty managers*

As previously mentioned, the logical connectives must be redefined to work with the fuzzy variables. In fuzzy logic, a common approach to model these connectives are triangular norms and conorms (Klement et al., 2000).

A triangular norm (t-norm for short) is a function $T : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ such that satisfies:

$$\forall x, y, x \in [0,1] \begin{cases} (T1) & T(x,y) = T(y,x) & \text{(commutativity)} \\ (T2) & T(x,T(y,z)) = T(T(x,y),z) & \text{(associativity)} \\ (T3) & T(x,y) \leq T(x,z) \text{ whenever } y \leq z & \text{(monotonicity)} \\ (T4) & T(x,1) = x & \text{(boundary condition)} \end{cases}$$

A triangular conorm (t-conorm for short) is a function $S : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ such that satisfies $(T1 - 3)$ and:

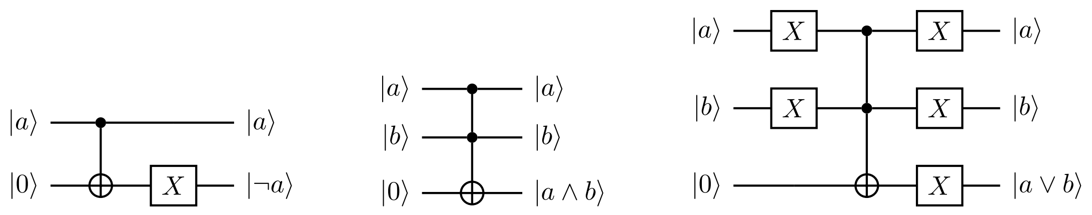
$$(S4) \quad S(x,0) = x \qquad \text{(boundary condition)}$$

Furthermore, a pair (T, S) is said to be conjugated if they related though DeMorgan's law:

$$\left.\begin{aligned} T(x,y) &= \neg S(\neg x, \neg y) \\ S(x,y) &= \neg T(\neg x, \neg y) \end{aligned}\right\} T \text{ and } S \text{ are conjugated}$$

In fuzzy logic, t-norms can be used to model the conjunction logical connective and t-conorms for the disjunction logical connective. With these terms presented, we can define a quantum approach for the logical connectives.

For the negation, we use the X gate, mainly because of its importance in the Quantum Computing literature as the NOT operator. For the conjunction, we use a $CCNOT$ gate, where the two control qubits act as the input of the logical connective and the third qubit is used to store the result. Following this conjunction operator, we built its conjugated operator for the disjunction, using DeMorgan's law (see Figure 4).



(a) NOT gate *(b) AND gate* *(c) OR gate*

***Figure 4**: Fuzzy logic's operators*

One of the aspects of this model that will evolve as we research on it is the implication, since for now we use the same *modus ponens* approach as in the certainty factors model.

3.2.3. Bayesian networks

The final model included in the software library is based on Bayesian networks. Since there is work already published on this subject, we found it interesting to include it as a model to build QRBS, with the required adjustments.

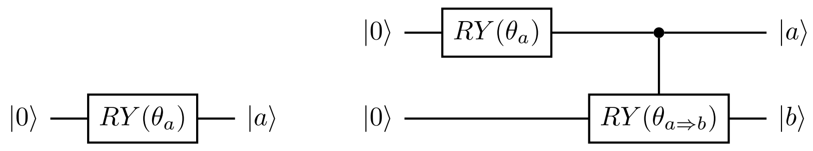
A Bayesian network is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG) that represents the joint probability distribution over those variables. A significant advantage of Bayesian networks is that, exploiting conditional dependencies present in the original distribution, we can reduce the space complexity of the model. This is done by using the edges of the graph to represent conditional dependencies associated to a conditional probability table (or a conditional probability distribution) (Russell & Norvig, 2010).

In (Borujeni et al., 2021) we find the following framework proposal for quantum Bayesian networks:

1. Map each node in a Bayesian network to one or more qubits (depending on the number of discrete states of the node).
2. Map the marginal/conditional probabilities of each node to the probability amplitudes (or probabilities) associated with various states of the qubit(s).
3. Apply the required probability amplitudes of quantum states using (controlled) rotation gates.

With these concepts in mind, we define our proposal for a classical-inspired quantum approach to QRBS through Bayesian networks.

We begin by modelling the codification of knowledge into the system. On the one hand, we use the RY gate to model the imprecision of the facts. On the other hand, we use its controlled version, the CRY gate, to model the uncertainty of the facts and at the same time the implication itself (see Figure 5).

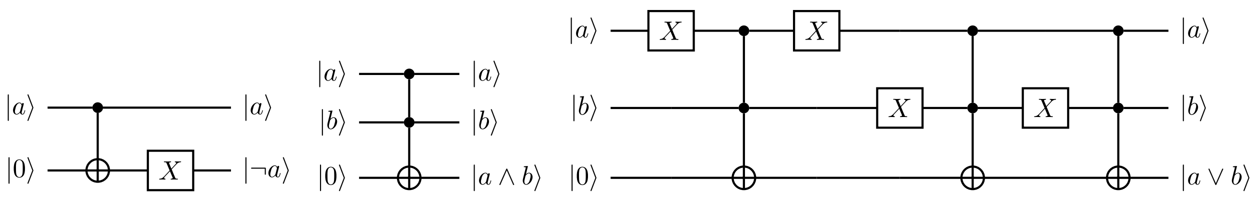


(a) RY gate *(b) Implication operator*

***Figure 5**: Bayesian networks' uncertainty managers*

Regarding the operators to relate the facts (see Figure 6), we make use once again of the NOT implementation that we previously used in the other models, and base the AND and OR operators on the relationship between nodes of a Bayesian network. Both the AND and OR operators are inspired by their quantum implementation in (Borujeni et al., 2021), where they are "activated" depending on the state of the qubits that represent the nodes of the network.

While this model is not as developed as the two former ones, we included it in this version of the software library because (1) there is room to further develop its software implementation as we research it and (2) the use of the CRY



(a) NOT gate *(b) AND gate* *(c) OR gate*

***Figure 6**: Bayesian networks' operators*

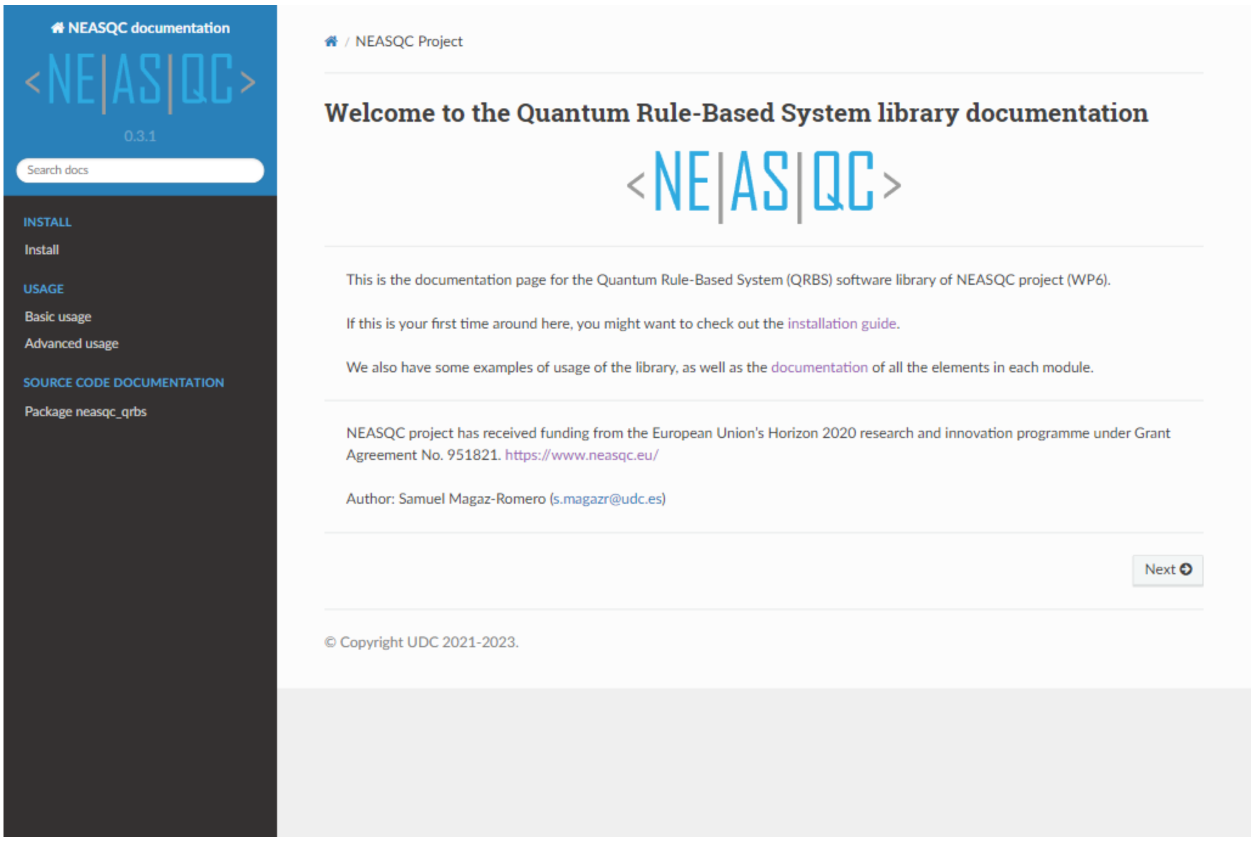
gate for the implication reduces the number of qubits required to implement a QRBS, and can therefore be interesting for those edge cases that need to reduce the amount of qubits of their systems.

We are aware of the similarities that are present between the models, for example regarding their operators, as they are derived from the same classical logical operators. The idea behind having different implementations is to further develop them as we keep researching the subject; we ask the reader to keep this in mind.

3.3. Code documentation

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

In comparison to the last documented version (Moret-Bonillo et al., 2023), we have added sections for (1) the installation of the library, (2) usage examples with some basic ones and some more advanced ones, and (3) the updated version of the software specification (see Fig. 7).



Welcome to the Quantum Rule-Based System library documentation

This is the documentation page for the Quantum Rule-Based System (QRBS) software library of NEASQC project (WP6).

If this is your first time around here, you might want to check out the installation guide.

We also have some examples of usage of the library, as well as the documentation of all the elements in each module.

NEASQC project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 951821. https://www.neasqc.eu/

Author: Samuel Magaz-Romero (s.magazr@udc.es)

© Copyright UDC 2021-2023.

***Figure 7**: Snapshot of the web documentation's landing page*

Appendix B presents the software specification section of the library documentation up to this point.

4. IDC application

In this chapter we illustrate the IDC (Invasive Ductal Carcinoma) application developed for this use case, by (1) introducing how it works, (2) providing its specification and (3) presenting the results of the experiments carried out.

4.1. Introduction

The IDC application that we have developed is supported by a knowledge model based on the TNM classification system (Giuliano et al., 2018), where the three sets of variables *T*, *N* and *M* (see Figure 3) are used to classify the current state of the patient (Moret-Bonillo et al., 2023).

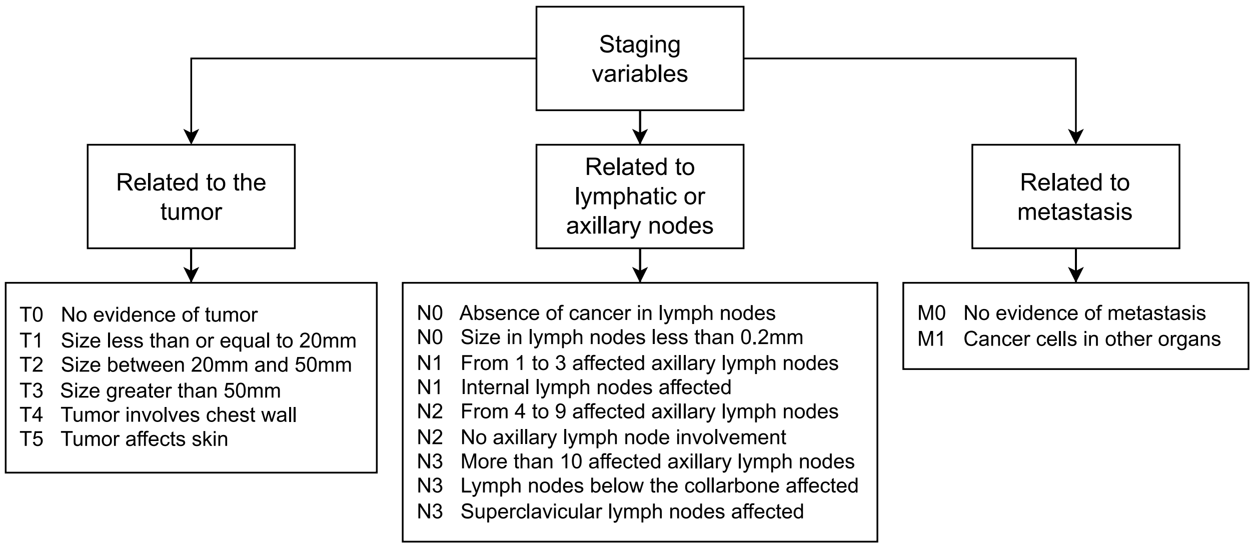

Staging variables
Related to the tumor
Related to lymphatic or axillary nodes
Related to metastasis
T0 No evidence of tumor
T1 Size less than or equal to 20mm
T2 Size between 20mm and 50mm
T3 Size greater than 50mm
T4 Tumor involves chest wall
T5 Tumor affects skin
N0 Absence of cancer in lymph nodes
N0 Size in lymph nodes less than 0.2mm
N1 From 1 to 3 affected axillary lymph nodes
N1 Internal lymph nodes affected
N2 From 4 to 9 affected axillary lymph nodes
N2 No axillary lymph node involvement
N3 More than 10 affected axillary lymph nodes
N3 Lymph nodes below the collarbone affected
N3 Superclavicular lymph nodes affected
M0 No evidence of metastasis
M1 Cancer cells in other organs

Figure 8: *Variables or symptoms that are considered in the knowledge model for IDC staging.*

Once the patient has been classified regarding the TNM system, the IDC stage is determined through the correspondence of Table 3. However, there are several TNM possible classifications that fit in more than one IDC stage (for example, the classification *T0N1M0* corresponds to both stages *I-B* and *II-A*).

Table 3: *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

With this context, we have developed as an application for IDC a web API (Application Programming Interface), a type of software interface for two programs to interact between each other. In this case, any software project that can use web APIs (most of them nowadays) will be able to integrate our application as it suits them best.

We have opted to implement our application as a web API since it follows the project goal of aiming its use cases towards users. With a web API, users are able to use the application in different ways: making requests to a server with the API running (public or private), modifying our proposal to achieve different results, etc.

4.2. Specification

In this section we present (1) the technical details of the implemented web API and (2) the specifications to use it.

4.2.1. API implementation

Since the IDC application makes use of the QRBS software library, which is developed in Python, we have chosen the same programming language to develop the web API, and use Flask to implement it. Flask is a lightweight Web Server Gateway Interface web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. In our case, it provides more than enough utilities to provide the level of functionality we want to achieve.

We use Flask to define our web API, which users can send requests to with the corresponding information about a patient, and obtain the IDC stage calculated with the QRBS software library. This is done by having previously defined the QRBS based on the IDC knowledge model, and initialising the variables with the data from the users.

We provide two options for the users: (1) they can send their TNM classification, just like specified in the knowledge model, or (2) they can provide the values for each of the TNM variables, and the systems calculates the TNM classification. In both cases, the output that users receive is the corresponding IDC stage to their input data (this is further specified in subsection 4.2.2). The source code of the web API can be found in https://github.com/NEASQC/idc-app.

4.2.2. API documentation

Tables 4 and 5 provide the API specification of the two options we have defined for the IDC application.

***Table 4**: API specification for endpoint /idc*

API endpoint: idc	
Route	/idc
Method	POST
Parameters	
Name	body
Required	Yes
Model	`{ "model": { "enum": ["cf", "fuzzy", "bayes"] }, "classification": { "tnm": { "enum": ["t1n0m0", "t0n1m0", "t1n1m0", "t2n0m0", "t2n1m0", "t3n0m0", "t0n2m0", "t1n2m0", "t3n2m0", "t3n1m0", "t4n0m0", "t4n1m0", "t4n2m0", "txn3m0", "txnym1"], }, "value": { "type": "number", "minimum": 0.0, "maximum": 1.0 } } }`
Responses	
Code	200 OK
Description	The result is returned, everything went as expected
Code	400 BAD REQUEST
Description	Validation exception, some data is incorrect

Table 5: API specification for endpoint /idc-deep

API endpoint: idc-deep	
Route	/idc-deep
Method	POST
Parameters	
Name	body
Required	Yes

Model

```
{
  "model": {
    "enum": ["cf", "fuzzy", "bayes"]
  },
  "variables": {
    "t0": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "t1": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "t2": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "t3": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "t4": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "t5": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n0a": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n0b": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n1a": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n1b": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n2a": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n2b": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n3a": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n3b": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "n3c": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "m0": { "type": "number", "minimum": 0.0, "maximum": 1.0 },
    "m1": { "type": "number", "minimum": 0.0, "maximum": 1.0 }
  }
}
```

Responses	
Code	200 OK
Description	The result is returned, everything went as expected
Code	400 BAD REQUEST
Description	Validation exception, some data is incorrect

4.3. Experiments and results

Once the endpoints of the API have been implemented, we test them in order to check that they provide the corresponding results to the inputs provided.

To do this, we use **pytest** once again, following the guidelines of Flask testing, and test the results for the three different models, for each possible stage of IDC. We also check the validation of the inputs, according to the model specified for each endpoint. These tests can be found in https://github.com/NEASQC/idc-app/tree/main/tests.

In this case, the coverage information is not that relevant, since a single call to the API will already check all the steps of the function. Here, we focus on the quantity of tests, which is a total of: 3 models $\times$ 15 TNM classifications $\times$ 2 API endpoints + 2 input validation tests = 90 tests. The results obtained are as expected for each of the possible diagnoses.

Regarding response times, on our experiments we have an average of 400ms for most of the requests, which is more than acceptable considering the amount of quantum circuits being executed with 1024 shots. We have run into some delays with certain routines, mainly for the ones of stages III-A and III-B due to their complexity, where we have come across response times of 1.5s. However, this is solvable by running the application in a higher end machine, but we wanted to acknowledge it since we are focusing on developing a user-friendly application.

5. Testing

In this chapter we continue the work on testing that we have been developing in previous deliverables. We present the new tests to cover the modification of the use cases and present the coverage information for the testing of the QRBS software library and the IDC application.

5.1. Traceability with use cases

The revision of use cases *UC-06: QRBS evaluation* and *UC-07: QRBS execution* has produced new scenarios that must be tested. We define the corresponding test cases for said scenarios, following the hierarchy established for traceability (Moret-Bonillo, Mosqueira-Rey, Magaz-Romero, & Gómez-Tato, 2021). With that, the new tests for these use cases can be found in Table 6.

We have structured the new test cases in separate classes, one for each of the models of QRBS. Therefore, the test classes `TestEvaluation` and `TestExecution` (without a model on its name) corresponds to the default model, the certainty factors model. The classes `TestEvaluationFuzzy` and `TestExecutionFuzzy` correspond to the fuzzy logic model and the classes `TestEvaluationBayes` and `TestExecutionBayes` correspond to the Bayesian network model, as their names suggest.

Test case ID	Test method	Test class
T-06-1	`test_positive_evaluation`	`TestEvaluation`
T-06-2	`test_negative_evaluation`	`TestEvaluation`
T-06-3	`test_positive_evaluation`	`TestEvaluationFuzzy`
T-06-4	`test_negative_evaluation`	`TestEvaluationFuzzy`
T-06-5	`test_positive_evaluation`	`TestEvaluationBayes`
T-06-6	`test_negative_evaluation`	`TestEvaluationBayes`
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`
T-07-7	`test_successful_default`	`TestExecutionFuzzy`
T-07-8	`test_failed_default_evaluation`	`TestExecutionFuzzy`
T-07-9	`test_failed_default_execution`	`TestExecutionFuzzy`
T-07-10	`test_successful_specified`	`TestExecutionFuzzy`
T-07-11	`test_failed_specified_evaluation`	`TestExecutionFuzzy`
T-07-12	`test_failed_specified_execution`	`TestExecutionFuzzy`
T-07-13	`test_successful_default`	`TestExecutionBayes`
T-07-14	`test_failed_default_evaluation`	`TestExecutionBayes`
T-07-15	`test_failed_default_execution`	`TestExecutionBayes`
T-07-16	`test_successful_specified`	`TestExecutionBayes`
T-07-17	`test_failed_specified_evaluation`	`TestExecutionBayes`
T-07-18	`test_failed_specified_execution`	`TestExecutionBayes`

Table 6: Test suite

Besides the new test cases, we have implemented unitary tests for the different `Builder` implementations we have for each model. These unitary tests, along with the test cases, can be found in https://github.com/NEASQC/qrbs/tree/main/tests.

5.2. Testing metrics

We present the coverage report for the QRBS software library and the IDC application at this point. More specifically, we cover the modelling of RBS, their respectives quantum circuits as we have one per model presented (see Section 3.2), and the results obtained after running those circuits with the myQLM simulator. We keep following the testing methodology that has been used during the development of the software projects (Moret-Bonillo et al., 2023).

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
neasqc_qrbs\knowledge_rep.py	329	8	45	69	6	96%
neasqc_qrbs\qrbs.py	124	0	12	55	1	99%
Total	455	8	57	124	7	**97%**

Table 7: Coverage report for QRBS software library

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
idc.py	150	3	0	4	0	98%
Total	150	3	0	4	0	**98%**

Table 8: Coverage report for IDC application

On both software projects we have a total coverage $\geq 95\%$, which is more than acceptable (most of the percentage missing belongs to fail-safe statements). These metrics are nothing but the result of the strong methodology we have followed throughout the development of the project, from the hierarchy for tracebility to the testing methodology.

6. Conclusions

In this deliverable we have presented the final version of the QRBS software library. This version provides the functionalities that were established as requirements at the first stages of this use case, completing the development process up to this point. On top of that, three models have been developed by taking inspiration from classical models to implement QRBS.

While we call this version final, we do not discard the possibility of adding more features in the future, as user get to know this tool and experiment with it, developing new ideas on their own.

The user-first approach has been supported by the traceability effort that has accompanied this work from the beginning, along with the testing. As we saw, a couple of the use cases of the library had to be revisited, but the documentation tracing all the process made this task easier to carry out.

Regarding the IDC application, we have revised the knowledge model that was presented in the previous deliverable and implemented a web API to use it. We provide two endpoints for flexibility, depending on the machine that runs the application, which we hope aids to the adaptation of this tool in already established workflows of IDC staging.

In summary, this report represents an important achievement in the work of this use case, providing the final version of the QRBS software library and the implementation of the IDC application. The following deliverable will provide the evaluation of the work carried out.

List of Acronyms

Term	Definition
API	Application Programming Interface
IDC	Invasive Ductal Carcinoma
QRBS	Quantum Rule-Based System
RBS	Rule-Based System
UC	Use Case

***Table 9**: Acronyms and Abbreviations*

List of Figures

List of Tables

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

Borujeni, S. E., Nannapaneni, S., Nguyen, N. H., Behrman, E. C., & Steck, J. E. (2021). Quantum circuit representation of bayesian networks. *Expert Systems with Applications*, *176*, 114768. https://doi.org/https://doi.org/10.1016/j.eswa.2021.114768

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

Heckerman, D. (1986). Probabilistic interpretations for mycin's certainty factors. In L. N. Kanal & J. F. Lemmer (Eds.), *Uncertainty in artificial intelligence* (pp. 167–196, Vol. 4). North-Holland. https://doi.org/https://doi.org/10.1016/B978-0-444-70058-2.50017-6

Klement, E. P., Mesiar, R., & Pap, E. (2000). *Triangular norms*. Springer Netherlands. https://doi.org/10.1007/978-94-015-9540-7

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

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

Moret-Bonillo, V., Gomez Tato, A., Magaz-Romero, S., Mosqueira-Rey, E., & Alvarez-Estevez, D. (2023, July). D6.11 Preliminary QRBS software and IDC application specification. https://doi.org/10.5281/zenodo.8108580

Moret-Bonillo, V., Mosqueira-Rey, E., & Magaz-Romero, S. (2021, December). 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., & 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

Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach*. Pearson Education, Inc.

Shortliffe, E. H., & Buchanan, B. G. (1975). A model of inexact reasoning in medicine. *Mathematical Biosciences*, *23*(3), 351–379. https://doi.org/https://doi.org/10.1016/0025-5564(75)90047-4

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*(3), 338–353. https://doi.org/doi.org/10.1016/S0019-9958(65)90241-X

A. Achievements

This appendix presents the achievements that have been produced during the development of the project up to this point with the work carried out.

A.1. Scientific material

A.1.1. Articles

- **Uncertainty in Quantum Rule-Based Systems**. Vicente Moret-Bonillo, Diego Alvarez-Estevez, Isaac Fernandez-Varela, *Archives of Clinical and Biomedical Research (Fortune Journals)*, 11/01/2021, 10.26502/acbr.50170149
- **Quantum Computing for Dealing with Inaccurate Knowledge Related to the Certainty Factors Model**. Vicente Moret-Bonillo, Samuel Magaz-Romero and Eduardo Mosqueira-Rey, *Mathematics (MDPI)*, 08/01/2022, 10.3390/math10020189
- **Hybrid Classic-Quantum Computing for Staging of Invasive Ductal Carcinoma of Breast**. Vicente Moret-Bonillo, Eduardo Mosqueira-Rey, Samuel Magaz-Romero and Diego Alvarez-Estevez, 17/03/2023, 10.48550/arXiv.2303.10142
- **Quantum Factory Method: A Software Engineering Approach to Deal with Incompatibilities in Quantum Libraries**. Samuel Magaz-Romero, Eduardo Mosqueira-Rey, Diego Alvarez-Estevez and Vicente Moret-Bonillo, *Proceedings of the ICCS - International Conference on Computational Science*, 26/06/2023, 10.1007/978-3-031-36030-5_6

A.1.2. Poster presentations

- **Quantum Rule-Based Systems: Managing Uncertain Information with Quantum Computing**. Vicente Moret-Bonillo, Eduardo Mosqueira-Rey, Samuel Magaz-Romero, Andrés Gómez-Tato and Daniele Musso, EQTC 2021, 29/11-02/12/2023, Virtual Conference
- **Quantum Factory Method: A Software Engineering Approach to Deal with Incompatibilities in Quantum Libraries**. Samuel Magaz-Romero, Eduardo Mosqueira-Rey, Diego Alvarez-Estevez and Vicente Moret-Bonillo, ICCS 2023, 03-05/07/2023, Prague (Czech Republic)

A.1.3. Dissemination activities

- **NEASQC webinar on QRBS**. Media source
- **EDF Scientific Rounds**. Media source
- **NEASQC in Corunna Innovate Summit (DataSpartan)**. Media source
- **Teaching presentation of the course on “Nanoscience, Quantum Computing and Nanotechnology”**. Media source

A.2. Academical works

A.2.1. BSc theses

- **Methodological study of quantum representation of fuzzy knowledge**. Alejandro Mayorga Redondo, 30/06/2023, Computer Science
- **Benchmarking of classical digital system, simulated quantum system and real quantum system**. Miguel Pérez Gómara, In progress, Computer Science

A.2.2. MSc theses

- **Web application for quantum rule-based systems management**. Samuel Magaz Romero, 22/09/2022, Software Engineering
- **Development of a knowledge based model for the staging of invasive ductal carcinoma**. Romina Riveron Martinez, 20/07/2023, Medicine
- **Study and implementation of quantum bayesian networks**. Alejandro Mayorga Redondo, In progress, Quantum Computing

A.2.3. PhD theses

- **Quantum treatment of classical uncertainty**. Samuel Magaz Romero, In progress, Computer Science

B. 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.

B.1. Module knowledge_rep

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

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

***abstract* build(*builder*) → qat.lang.AQASM.QRoutine**

***class* neasqc_qrbs.knowledge_rep.LeftHandSide** Bases: `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(*builder*) → qat.lang.AQASM.QRoutine

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

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

Type: float, optional

***property* precision**

build(*builder*) → qat.lang.AQASM.QRoutine

***class* neasqc_qrbs.knowledge_rep.AndOperator(*left_child*, *right_child*)** Bases: `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*) → qat.lang.AQASM.QRoutine

***class* neasqc_qrbs.knowledge_rep.OrOperator(*left_child*, *right_child*)** Bases: `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*) → **qat.lang.AQASM.QRoutine**

class **neasqc_qrbs.knowledge_rep.NotOperator(*child*)** Bases: `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*) → **qat.lang.AQASM.QRoutine**

class **neasqc_qrbs.knowledge_rep.Rule(*left_hand_side*, *right_hand_side*, *certainty=0.0*)** Bases: `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.

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

Type: `LeftHandSide`

right_hand_side Right hand side element of the rule (also known as consequent).

Type: `Fact`

certainty Certainty of the relationship between precedent and consequent (0 if not specified). Must be in range [0,1].

Type: float, optional

property **certainty**

build(*builder*) → **qat.lang.AQASM.QRoutine**

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

build(*builder*) → **qat.lang.AQASM.QRoutine**

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

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

abstract static **build_fact(*fact*)** → **qat.lang.AQASM.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`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

***abstract static* build_rule(*rule*) → qat.lang.AQASM.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`

***abstract static* build_island(*island*) → Tuple[qat.lang.AQASM.QRoutine, Dict[, 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]]

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

Implementation of Builder interface.

M = *'M'*

***static* build_fact(*fact*) → qat.lang.AQASM.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_and() → qat.lang.AQASM.QRoutine** Builds the quantum routine of an and operator.

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

***static* build_rule(*rule*) → qat.lang.AQASM.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`

***static* build_island(*island*) → Tuple[qat.lang.AQASM.QRoutine, Dict[, 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]]

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

Implementation of Builder interface for the fuzzy logic model.

***static* build_fact(*fact*) → qat.lang.AQASM.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_and() → qat.lang.AQASM.QRoutine** Builds the quantum routine of an and operator.

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

***static* build_rule(*rule*) → qat.lang.AQASM.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`

***static* build_island(*island*) → Tuple[qat.lang.AQASM.QRoutine, Dict[, 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]]

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

Implementation of Builder interface for the bayesian model.

CRY = *'CRY'*

***static* build_fact(*fact*) → qat.lang.AQASM.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_and() → qat.lang.AQASM.QRoutine** Builds the quantum routine of an and operator.

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

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

Returns: The corresponding quantum routine.

Return type: `QRoutine`

static **build_rule(*rule*)** → **qat.lang.AQASM.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`

static **build_island(*island*)** → **Tuple[qat.lang.AQASM.QRoutine, Dict[, 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]]

B.2. Module qrbs

class **neasqc_qrbs.qrbs.WorkingMemory(*facts=None*)** 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*) → **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.

class **neasqc_qrbs.qrbs.InferenceEngine(*rules=None*, *islands=None*)** 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_rule(*rule*) → **Rule** Asserts a rule into the engine.

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

Returns: The asserted rule.

Return type: `Rule`

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.

assert_island(*island*) → **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.

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

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

***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 precision and certainty.

_memory The Working Memory of the system.

Type: `WorkingMemory`

_engine The Inference Engine of the system.

Type: `InferenceEngine`

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

Parameters:

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

Returns: The asserted fact.

Return type: `Fact`

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

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

assert_rule(*lefthandside*, *righthandside*, *certainty=0.0*) → **Rule** 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.
- **certainty** (*float, optional*) – The certainty of the rule.

Returns: The asserted rule.

Return type: `Rule`

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

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

assert_island(*rules*) → **KnowledgeIsland** 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`

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

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

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

myQLM implementation of a Quantum Processing Unit (QPU).

MAX_ARITY ***= 20***

BUILDERS = {
'cf':<class'neasqc_qrbs.knowledge_rep.BuilderImpl'>,
'bayes': <class'neasqc_qrbs.knowledge_rep.BuilderBayes'>,
'fuzzy':<class'neasqc_qrbs.knowledge_rep.BuilderFuzzy'>
}

static **evaluate(*qrbs*, *eval_islands=None*, *model='cf'*)** → **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.
- **model** (*str, optional*) – The code of the model indicated.

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

static **execute(*qrbs*, *islands=None*, *model='cf'*)** → **None** Executes the QRBS on this QPU.

Parameters:

- **qrbs** (`QRBS`) – The QRBS to be executed.
- **islands** (List[`KnowledgeIsland`], optional) – A list of specific KnowledgeIsland to be executed.
- **model** (*str, optional*) – The code of the model indicated.