

Reverse Engineering of Classical-Quantum Programs

Luis Jiménez-Navajas¹^a, Ricardo Pérez-Castillo¹^b and Mario Piattini²^c

¹*aQuantum, Faculty of Social Sciences and IT, University of Castilla-La Mancha Talavera de la Reina, Spain*

²*aQuantum, Information Technology and Systems Institute, University of Castilla-La Mancha Ciudad Real, Spain*

Keywords: Reverse Engineering, Quantum Computing, Software Modernization, Qiskit, Knowledge Discovery Metamodel.

Abstract: Quantum computing has emerged as a crucial technology, which is expected to be progressively integrated into current, traditional information systems. Society could be benefited from several potential, promising applications based on quantum computing. To achieve such advantages, this new paradigm will require integrating the quantum software into the new hybrid (classical-quantum) information systems. Thus, it is necessary to adapt well-known and validated software engineering methods and techniques, such as software evolution methods based on Model-Driven Engineering principles. In particular, the proposal of this paper is framed in the Quantum Software Modernization process, and, in particular, it addresses the reverse engineering phase. The main contribution is a reverse engineering technique that analyses quantum (Qiskit) and classical (Python) code and builds a common, abstract model that combines both classical and quantum elements. The models are built in a technology-agnostic manner through the Knowledge Discovery Metamodel. Within this technique, relationships have been established between classical and quantum elements which can help to preserve knowledge and provide meaningful insights during the evolution toward hybrid information systems. The functioning of this technique is demonstrated through a running example with a program from the Qiskit Github repository.


1 INTRODUCTION


In recent years, a great number of achievements have been beheld in the field of quantum computing. This new technology is leading to promising applications like the discovery of new drugs (Cao et al., 2018) or new industrial solutions related to optimization problems (Bayerstadler et al., 2021) which, if they finally succeed, will inevitably imply an impact on society. Furthermore, the various providers of quantum computing have enabled a truly accessible ecosystem for the development of quantum software which favors the development of this technology.


Although quantum computing is gaining more and more importance, software engineering for quantum software should be considered equally important (Hoare and Milner, 2005). In order to develop high-quality quantum software in an industrial and controlled manner, it is necessary to adapt the knowledge and lessons learned from the software engineering field, so the Quantum Software Engineering (QSE)

approach is essential (Piattini et al., 2020; Serrano et al., 2022).

Companies that are eager to get benefited from quantum software, may want to adapt their classical software systems in order to adopt the quantum computing paradigm. For example, companies researching the creation of certain materials or pharmaceuticals will be forced to adopt quantum software, since its performance is better than classical software with these type of problems (Aaronson, 2008). However, discarding the classical information systems (as a whole) and migrating them into quantum information systems makes no sense as quantum computing performs better than classical computing for specific scenarios (those that may take advantage of quantum parallelism or simulation of elements (Zhao, 2020)). Thus, the migration toward pure quantum information systems is very unlikely since the performance-cost ratio could be degraded, among other reasons. In contrast, it is expected to integrate quantum software into the current information systems, leading to the evolution toward the so-called hybrid information systems. Typically, in these hybrid information systems, the classical part controls and receives all the

^a <https://orcid.org/0000-0001-6257-7153>

^b <https://orcid.org/0000-0002-9271-3184>

^c <https://orcid.org/0000-0002-7212-8279>

outputs generated by the quantum software components, which is, generally, executed in remote quantum computers in the cloud (Nguyen et al., 2022).

One of the approaches to address the evolution toward hybrid software systems is software modernization. Software Modernization embraces the traditional reengineering process by following the Model-Driven Engineering (MDE) principles. Software modernization has allowed organizations to develop and evolve high-quality software in compliance with standards and following well-proven practices (Durelli et al., 2014). To make a successful evolution from classical information systems to hybrid information systems, the software modernization process has been adapted to the field of quantum software (Perez-Castillo et al., 2021). This process of Quantum Software Modernization alleviates problems that may occur during the analysis, design, and generation of hybrid information systems (Pérez-Castillo et al., 2024).

The Quantum Software Modernization process involves three main phases: reverse engineering, restructuring, and forward engineering. This paper specifically focuses on the reverse engineering phase. This phase is crucial since is aimed at generating abstract representations of quantum and classical programs in combination. This proposal produces abstract models in a technology-agnostic manner through the usage of the Knowledge Discovery Metamodel (KDM) (Pérez-Castillo et al., 2011). The reverse engineering technique analyzes the source code of hybrid software programs developed in Python, which import the Qiskit library (Aleksandrowicz et al., 2019).

The remainder of the paper is structured as follows: Section 2 depicts the current state of the art of quantum computing and classical-quantum software modernization. Then, Section 3 introduces a running example. Section 4 explains the proposed reverse engineering technique and illustrates how it works through the development of the running example. Finally, Section 5 presents the conclusions obtained from this research and future work.

2 BACKGROUND

This section introduces the underlying concepts related to the technique. Section 2.1 describes how quantum software works. Section 2.2 presents the software modernization process. Finally, Section 2.3 explains the adaptation of software modernization to be applied to the evolution of hybrid information systems.

2.1 Quantum Software

The Quantum Software Manifesto stated that “given the recent rapid advances in quantum hardware, it is urgent that we step up our efforts in quantum software” (Ambainis et al., 2017), and the European Quantum Flagship’s Strategic Research Agenda (EQF, 2020) proposes to investigate the “development of software stacks to integrate quantum computing into current computing environments”.

Currently, there are a large number of providers that allow us to develop quantum software. The world’s largest companies are already providing programming languages and libraries that enable the development of quantum software, i.e., OpenQASM, Q#, Qiskit (IBM), Cirq (Google), pyQuil (Rigetti), among others (Garhwal et al., 2019; Zhao, 2020).

Unfortunately, several quantum computer scientists do not know the software engineering principles and techniques, so several errors could be repeated, and some expensive “rediscoveries” could happen (Piattini et al., 2021). Nevertheless, with the advent of “industrial” quantum software and its increasing use in various business domains, there is a growing demand for quantum software to be produced more systematically, as stated in the Talavera Manifesto for Quantum Software Engineering and Programming (Piattini et al., 2020).

2.2 Software Modernization

The course of time also affects information systems, forcing them to evolve continuously. Those systems that were developed in the past can be affected by degradation or aging, turning them into legacy information systems, which means that the source code which has been developed could be technologically obsolete (Ulrich, 2002). The evolution of legacy systems is a duty that has been addressed for years since many companies cannot simply discard such systems, as they possibly embed a large amount of business logic and business rules that are not depicted elsewhere (Sommerville, 2011).

Reengineering is a successful practice in the software industry, consisting of three phases: reverse engineering, restructuring, and forward engineering. However, when faced with specific challenges, this practice has more than half of the traditional reengineering projects fail because of a lack of standardized and automated processes (Sneed, 2005). This is why the OMG proposed ADM as a possible mechanism for software evolution, which consists of applying reengineering with an MDE approach. For the reverse engineering phase, ADM advocates the use of

KDM to abstract and represent all the software artifacts that can be found in a legacy system. This representation is done in a standard and technology-independent way.

2.3 Quantum Software Modernization

The future implementation of quantum software or services in our information systems will inevitably lead to an evolution of information systems to adopt this software. This does not imply the complete replacement of classical information systems by quantum software but rather a modernization of these systems integrating new quantum software components. To this end, the software modernization process has been adapted to the domain of quantum computing (see Fig. 1).

The Quantum Software Modernization process was first proposed in (Jiménez-Navajas et al., 2020) and consists of the same phases as classical reengineering: reverse engineering, restructuring, and forward engineering. In the reverse engineering phase (the scope of this paper), software artifacts are analyzed, and their components and interrelationships are represented at a higher level of abstraction. For this proposal, classical and quantum code has been analyzed and represented using KDM models. The restructuring phase consists of improving the internal structure of the model (either by modifying or adding functionalities) but preserving the behavior of the system. Finally, in the forward engineering phase through generative and low-code techniques, the target classical-quantum information system is generated in a semi-automatic way.

3 RUNNING EXAMPLE

This section presents a running example which in the following sections will be elaborated on based on the proposed technique to illustrate how the reverse engineering technique works. It should be mentioned that this example has been entirely conducted using a supporting tool that has been developed (Jiménez-Navajas et al., 2023a). The input of the technique is a basic program extracted from the Qiskit repository (Aleksandrowicz et al., 2019) and using Qiskit's version 0.41. This example can be seen in Listing 1. Although the complete program consists of two circuits, only the first circuit (everything referred to with `qc1`) will be explained hereinafter due to length limits. The KDM representation of the complete program can be obtained from (Jiménez-Navajas et al., 2023b).

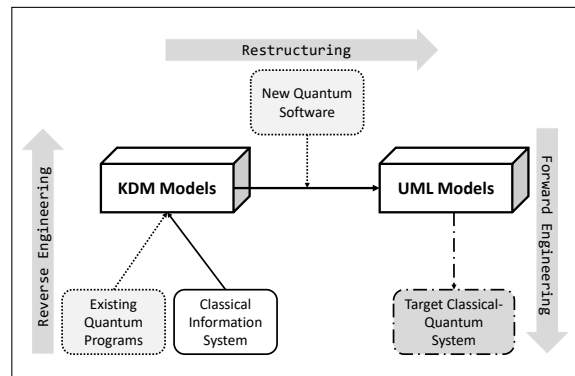


Figure 1: Quantum software modernization process.

```

1 # making first circuit: bell state
2 qc1 = QuantumCircuit(2, 2)
3 qc1.h(0)
4 qc1.cx(0, 1)
5 qc1.measure([0, 1], [0, 1])
6
7 # making another circuit: superpositions
8 qc2 = QuantumCircuit(2, 2)
9 qc2.h([0, 1])
10 qc2.measure([0, 1], [0, 1])
11
12 # setting up the backend
13 print("(BasicAER Backends)")
14 print(BasicAer.backends())
15
16 # running the job
17 job_sim = execute([qc1, qc2], BasicAer.get_backend("
18     qasm_simulator"))
19 sim_result = job_sim.result()
20
21 # Show the results
22 print(sim_result.get_counts(qc1))
23 print(sim_result.get_counts(qc2))
  
```

Listing 1: Python program consisting of two quantum circuits.

The declaration of a quantum circuit can be seen in line 2. This circuit will have two qubits and two classical registers. These last two registers will be used later to store the values obtained from the measurements of the qubits. Next, in line 3, a Hadamard gate is applied to the first qubit of the `qc1` circuit. This gate has the function of setting the qubits in a superposition state, i.e., it causes the qubits to have two different states simultaneously. In line 4, the Controlled Not gate is applied to qubits 0 and 1. The function of this gate is to perform a negation on the second qubit (target) if the state of the first qubit (control) is $|1\rangle$. In line 5, a measurement operation is applied to qubits 0 and 1 and stored in the classical registers 0 and 1, respectively. This measurement undoes the superposition of the qubits and reveals their state, setting them to $|0\rangle$ or $|1\rangle$ until another quantum gate is reapplied and changes their state. Finally, in lines 17 and 18, the circuit is executed in the backend of `qasm_simulator`, and the results are stored in the variable `sim_result`.

4 REVERSE ENGINEERING OF HYBRID QUANTUM SOFTWARE

For the representation of the hybrid programs, an approach where two models are represented within the same KDM segment, composed of a classical and a quantum software model has been followed. On the one hand, in the classical software model (cf. Section 4.1), the whole Python program is represented, including the Qiskit code. This means that Qiskit (quantum) elements are depicted as mere classical functions. On the other hand, in the quantum software model (cf. Section 4.2), the underlying quantum circuit is abstracted from the embedded Qiskit code. The representation of these models in combination is the main contribution of this technique since it enables the creation of valuable relations between elements of the two models (cf. Section 4.3). Modelling such relationships can help us to answer interesting, valuable questions during the software modernization process. These questions are the following (although this is not a limited list):

- **RQ1.** What classical data elements are going to receive and manage results from the quantum software component?
- **RQ2.** What classical parts are involved in the dynamic generation of a quantum circuit?
- **RQ3.** What quantum circuits are controlled and executed by specific classical drivers?

This technique has been divided into two phases. The first phase consists of the static analysis of the hybrid programs and their representation through an Abstract Syntax Tree (AST). The second phase takes the AST as input and generates KDM representations of each element embodied in the tree depending on its function in the program.

For the first phase analysis of the hybrid programs, ANother Tool for Language Recognition (ANTLR) has been employed (Parr, 2013). This tool allows parsing source code and representing it in an AST by defining the grammar and lexicon on which the file is based. Program parsing is divided into two phases: lexical analysis and parsing. In the lexical analysis phase, characters are grouped into words or other characters (tokenize), and this is done by a lexer based on Python code. The second stage is the actual parser which feeds off these tokens to recognize the sentence structure, in this case, an assignment statement.

In the lexical analysis phase, the Python3 lexicon and grammar available in the ANTLR GitHub repository has been used as a starting point since Qiskit is a

library of this language (Everet, 2020). However, this grammar had to be modified so that in the next phase, some quantum elements could be identified in a better way. For the running example presented, Listing 2 shows an example of the modification in the Python3 lexer file. In the running example, there are only three quantum gates: a Hadamard gate, a Controlled Not gate, and a measure operation (the implementation of these can be seen in Listing 1). Adding these lines to the lexicon file three new tokens are defined. Afterwards, in the parser file it is defined how each these tokens are grouped and represented.

```
1 // Qiskit's Quantum Gates
2 HADAMARD      : 'h';
3 CONTROLLEDX  : 'cx';
4 MEASURE       : 'measure';
```

Listing 2: Quantum gates defined on the lexer.

Having analyzed the programs and built the AST model, it is then necessary to define how each of the elements of the tree will be represented in certain parts of the KDM model. Section 4.1 explains how these elements are modelled for the classical software model, and Section 4.2 the same mapping for the quantum software model. Finally, Section 4.3 discusses the relationships that are established between the two models.

4.1 Classical Software Model

The classical representation of the hybrid programs (based on Qiskit-Python code) defines how each element must be represented in KDM. All variables that will save any result are represented using the `code:StorableUnit` type, while functions are represented using the `action:ActionElement` type. Also, each element appearing in the classical software model has a child of type `action:CompliesTo` aiming to its corresponding `LanguageUnit` definition. A `LanguageUnits` represents predefined data types and other common elements determined by a particular programming language. This `LanguageUnit` has the function of helping the model to preserve as much information as possible.

Fig. 2 shows (through the Eclipse IDE model viewer) part of the outgoing KDM model representing the classical software model of Listing 1. At the beginning of the model, there are three `code:StorableUnit` elements. One is the quantum register where all the quantum operations point to any of the respective indexes, another is the classical register where the measurements of the operations performed on the quantum register are stored and the last one is the variable that stores the two registers (so, the quantum circuit). When a circuit is de-

clared using the Qiskit library, the variable that declares it holds a quantum register and a classical register with the size indicated. The next element is `qc1=QuantumCircuit(2,2)` which represents the action of declaring a circuit with two qubits and two classical registers. Afterward, all the quantum gates used in the circuit are represented (Hadamard, Controlled Not, and Measure) and then two `print` statements (classical functions). Finally, the algorithm is executed with the `execute` statement and the results are stored in the variable `job_sim`, so another `code:Storable Unit` is declared. With the function `result` the results of the execution of a circuit are saved. Since those results must be stored in another variable, an additional `sim_result` was declared. Finally, there is a `print` statement that displays the results of `sim_result`.

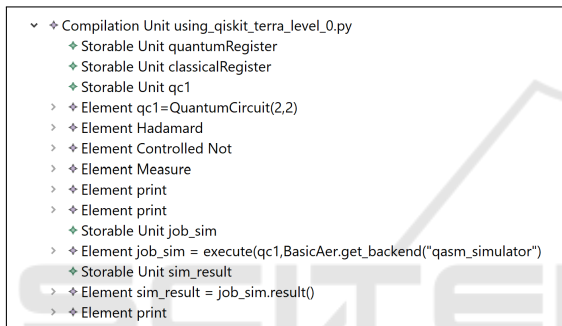


Figure 2: Classical software model generated.

Fig. 3 shows the `LanguageUnit` generated from the circuit. Two `LanguageUnit` have been modelled: one for Qiskit and another for Python. Each of these `LanguageUnit` elements contain the description of the elements of the programming language to which this `LanguageUnits` corresponds to. Thus, the Qiskit `LanguageUnit` only contains those elements that are used in the program and defined by Qiskit (data types, quantum gates, etc.). This means that in the classic model of the example, two Python elements are used during the development of the program: a `print` statement and an integer value definition.

4.2 Quantum Software Model

The quantum software model abstracts the underlying quantum circuit that is derived from the embedded Qiskit source code. Abstracting this model is not straightforward since many quantum circuits are built dynamically with external data sources. The approach taken for the representation could be considered similar to (Jiménez-Navajas et al., 2020), where all the different quantum elements are associated with a stereotype from the extension family.

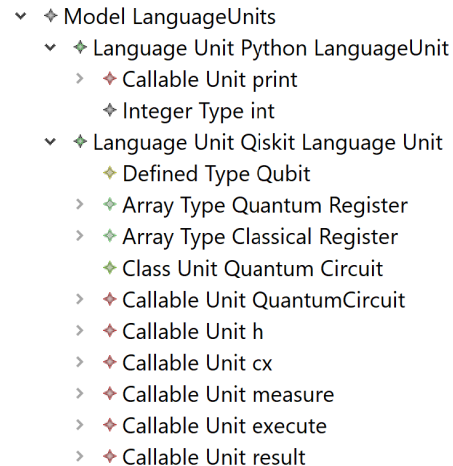


Figure 3: LanguageUnits generated.

That extension family allows the extension of the semantics of KDM to represent all the quantum elements that may appear in a program, applying them to the elements represented with the metamodel definition.

Fig. 4 provides the KDM representation of the quantum software model in Listing 1. In contrast to the classical software model, this model is shorter. This is because the underlying quantum circuit derived from the Qiskit code is abstracted. This implies representing only the quantum gates applied to the defined circuit. Also, it can be seen how the different quantum gates that perform actions to the circuit `qc1` are nested to it, as well as the number of qubits that were declared. Each of these quantum gates has a `stereotype` attribute referencing the stereotype identifier in the extension family. Another important detail is that two `measure` operations appear since in this pure quantum software model the measurement that was performed (see Listing 1) over two qubits at the same time has been represented as two different `measure` operations. All these elements of the quantum software model are defined in the Qiskit `LanguageUnit` (see Fig. 3).

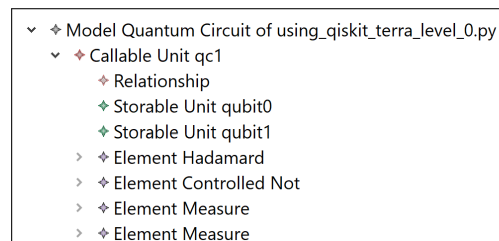


Figure 4: Quantum software model generated.

4.3 Modelling of Classical-Quantum Relationships

The definition of these relationships is the main contribution of this research. In the proposed technique, two different models (one classical and one hybrid) are represented in the same KDM segment, and relationships between elements of those two models are established. Three KDM mechanisms are used to define relationships: the `code:CodeRelationship` element and the `to` attribute of the `action:Writes` of the qubit measurement operations, and the association between the element `action:Reads` in the representation of the `execute` function with the quantum circuit declared in the classical software model.

Fig 5 shows an excerpt of the outgoing quantum and classical KDM model representations for the declaration of the quantum circuit, the measurement of the two qubits declared, and the execution of `qc1` (see lines 2, 5, and 7 in Listing 1, respectively).

In the classical software model, the three first elements represent the `code:StorableUnit` elements. The two first `code:StorableUnit` elements have a `type` attribute whose value points, respectively, to the description of the Quantum and Classical registers. Then, the `action:ActionElement` represents the declaration of the circuit that it has as children:

- The value of the first and parameters with the `type` pointing to the definition of the Quantum and Classical Registers in the `LanguageUnit`, respectively.
- The relation indicating that the declaration of the circuit is compliant with the Qiskit definition, pointing to the element "Quantum Circuit" of the `LanguageUnit`.
- The relation that indicates on which `code:StorableUnit` the circuit will be declared, i.e., `qc1`

Regarding the measurement operation representation, it is represented employing an `action:ActionElement` that has as children:

- The programming language to which it belongs.
- The relation pointing to which quantum circuit is acting, in this case, `qc1`.
- The relation pointing to which registers it affects.
- Both the child with `type Calls` and `action:CompliesTo` point to the same identifier, which is the description of the measurement operation in the `LanguageUnit`.
- The value of the parameter used in the gate, with `type` pointing to the definition of the Quantum Register in the `LanguageUnit`.

- The relation indicates that the value sent by the parameter is read.
- The relationship that associates the quantum register's `itemUnit`.
- The value of the parameter used in the gate where the value of the qubit is going to be stored, with `type` pointing to the definition of the Classical Register in the `LanguageUnit`.
- The relation indicates that the value sent by the parameter is read.
- The relationship that associates the classical register's `itemUnit`.

Concerning the representation of the execution of the circuit, first, the declaration of the variable where the results of the operation will be stored is represented. Then, an element of `action:ActionElement` that represents what the function `execute` performs is shown. This element has the children:

- An `action:Reads` element that refers to the quantum circuit that is reading.
- An `action:CompliesTo` element that indicates which element of the `LanguageUnit` it complies to.
- An `action:Writes` element associated with the variable declared in the previous line to represent the writing on it.

Regarding the quantum software model, the quantum circuit is defined using a nested `code:CallableUnit`, containing all the quantum elements related to it (e.g., the qubits declared and the quantum gates used in the circuit). In particular, the nested elements are the following:

- The `code:CodeRelationship` element relating the quantum circuit representation in the quantum and the classical software models, with the stereotype associated with the quantum circuits relationship.
- The two qubits declare the circuit through the `code:StorableUnit` element. Both point to the extension family stereotype `qubit`.
- The gate `measure` with the `type action:ActionElement` to indicate that it does an action type, which in this case indicates that it does a read on `qubit0` thanks to its child element of `type action:Reads` that points from the quantum gate identifier to the identifier of the qubit on which it is applied. Immediately afterward, using a child element of `type action:Writes` that references to the classical



Figure 5: Excerpt of the output KDM model.

register of the circuit represented in the classical software model, it indicates where the results of the operations performed are stored.

The lines have been chosen to be represented to see graphically the relationships between both models. The green line indicates which classical register modelled in the classical software model will store the results from a measurement operation of a quantum circuit. This relationship relates to the classical and quantum software models, pointing out which classical element is saving the state of a qubit. With this relation, we can answer (RQ1), as we can obtain which data elements are receiving the results from the qubit measuring as well as know which elements could manage those results. The black line for the relationship of the representation of the quantum circuit between the classical and the quantum software model. This relationship also associates the description of the quantum circuit in the classical and the quantum software model, but this time we could answer questions related to the influence of classical elements in the generation of quantum circuits (RQ2). Finally, the yellow line indicates which quantum circuit will be executed. With this relationship, the final executed circuit can be tracked if the relationship is followed, and the variable which stores the quantum circuit can be obtained. This can help us to answer questions related to which circuit is controlled and executed by specific classical drivers (RQ3).

5 CONCLUSIONS

This paper has presented a reverse engineering technique that analyses hybrid (classical-quantum) programs developed using Python-Qiskit. For this purpose, a parser for Python source code is adapted to represent the different quantum elements that may be used in a hybrid program in the AST model. Then, the relevant nodes of the AST model are represented based on the standard KDM. This representation is divided into two models: a classical software model representing the program; and a quantum software model representing the final underlying quantum circuit. These two models are related to a set of interrelationships that associate the quantum elements in both models. A running example is conducted to illustrate how this new technique works.

This technique helps to improve the quantum software modernization process since the representations of both models and the relationships established between them help to extract valuable insights and answer interesting questions. This is important since in the near future, different organizations will have to address the task of modernizing their information systems to be benefited from the promising applications derived from quantum computing.

In future work, the design in KDM of more elements will be addressed. With this, the representation of as many programs as possible using this technique is an objective in the near term.

ACKNOWLEDGEMENTS

This work is part of the projects PID2022-137944NB-I00 (SMOOTH Project) and PDC2022-133051-I00 (QU-ASAP Project) funded by MCIN/AEI/10.13039/501100011033 / PRTR, EU.

REFERENCES

- Aaronson, S. (2008). The limits of quantum. *Scientific American*, 298(3):62–69.
- Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., and et al. (2019). Qiskit: An Open-source Framework for Quantum Computing.
- Ambainis, A. et al. (2017). Quantum software manifesto.
- Bayerstadler, A., Becquin, G., Binder, J., Botter, T., Ehm, H., Ehmer, T., Erdmann, M., Gaus, N., Harbach, P., Hess, M., et al. (2021). Industry quantum computing applications. *EPJ Quantum Technology*, 8(1):25.
- Cao, Y., Romero, J., and Aspuru-Guzik, A. (2018). Potential of quantum computing for drug discovery. *IBM Journal of Research and Development*, 62(6):6–1.
- Durelli, R. S., Santibáñez, D. S. M., Marinho, B., Honda, R., Delamaro, M. E., Anquetil, N., and de Camargo, V. V. (2014). A mapping study on architecture-driven modernization. In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pages 577–584.
- EQF (2020). European quantum flagship strategic research agenda.
- Everet, T. (2020). Python3 grammar developed in antlr4.
- Garhwal, S., Ghorani, M., and Ahmad, A. (2019). Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering*, 28.
- Hoare, T. and Milner, R. (2005). Grand challenges for computing research. *The Computer Journal*, 48(1):49–52.
- Jiménez-Navajas, L., Pérez-Castillo, R., and Piattini, M. (2020). Reverse engineering of quantum programs toward kdm models. In Shepperd, M., Brito e Abreu, F., Rodrigues da Silva, A., and Pérez-Castillo, R., editors, *Quality of Information and Communications Technology*, pages 249–262, Cham. Springer International Publishing.
- Jiménez-Navajas, L., Pérez-Castillo, R., and Piattini, M. (2023a). Github’s repository of the technique.
- Jiménez-Navajas, L., Pérez-Castillo, R., and Piattini, M. (2023b). KDM representation of the program ”using_qiskit_terra_level_0.py”.
- Nguyen, H. T., Usman, M., and Buyya, R. (2022). Qfaas: A serverless function-as-a-service framework for quantum computing. *arXiv preprint arXiv:2205.14845*.
- Parr, T. (2013). The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, pages 1–326.
- Pérez-Castillo, R., De Guzman, I. G.-R., and Piattini, M. (2011). Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532.
- Pérez-Castillo, R., Serrano, M. A., and Piattini, M. (2021). Software modernization to embrace quantum technology. *Advances in Engineering Software*, 151:102933.
- Piattini, M., Peterssen, G., Pérez-Castillo, R., Hevia, J. L., Serrano, M. A., Hernández, G., de Guzmán, I. G. R., Paradela, C. A., Polo, M., Murina, E., et al. (2020). The talavera manifesto for quantum software engineering and programming. In *QANSWER*, pages 1–5.
- Piattini, M., Serrano, M., Pérez-Castillo, R., Petersen, G., and Hevia, J. L. (2021). Toward a quantum software engineering. *IT Professional*, 23(1):62–66.
- Pérez-Castillo, R., Serrano, M. A., Cruz-Lemus, J. A., and Piattini, M. (2024). Guidelines to use the incremental commitment spiral model for developing quantum-classical systems. *Quantum Information and Computation*, 24(1&2):71–88.
- Serrano, M. A., Pérez-Castillo, R., and Piattini, M. (2022). *Quantum Software Engineering*. Springer Nature.
- Sneed, H. M. (2005). Estimating the costs of a reengineering project. In *12th Working Conference on Reverse Engineering (WCRE’05)*. IEEE.
- Sommerville, I. (2011). *Software Engineering, 9/E*. Pearson Education India.
- Ulrich (2002). *Legacy systems: transformation strategies*. Prentice Hall PTR.
- Zhao, J. (2020). Quantum software engineering: Landscapes and horizons.