QRev: migrating quantum code towards hybrid information systems

Ricardo Pérez-Castillo^{1,2} · Luis Jiménez-Navajas¹ · Mario Piattini¹

Accepted: 14 September 2021 / Published online: 29 October 2021
© The Author(s) 2021

Abstract

Quantum computing is now a reality, and its incomparable computational power has led companies to show a great interest in being able to work with quantum software in order to support part of their current and future business operations. However, the quantum computing paradigm differs significantly from its classical counterparts, which has brought about the need to revolutionise how the future software is designed, built, and operated in order to work with quantum computers. Since companies cannot discard all their current (and probably mission-critical) information systems, they must adapt their classical information systems to new specific quantum applications, thus evolving towards hybrid information systems. Unfortunately, there are no specific methods with which to deal with this challenge. We believe that reengineering, and more specifically, software modernisation using model-driven engineering principles, could be useful as regard migrating classical systems and existing quantum programs towards hybrid information systems. This paper, therefore, presents QRev, a reverse engineering tool that analyses quantum programs developed in Q# in order to identify its components and interrelationships, and then generates abstract models that can be used in software modernisation processes. The platform-independent models are generated according to the Knowledge Discovery Metamodel (KDM) standard. QRev is validated in a case study involving five quantum programs in order to demonstrate its effectiveness and scalability. The main implication of the study is that QRev can be used in order to attain KDM models, which can subsequently be employed to restructure or add new quantum functionality at a higher abstraction level, i.e. independently of the specific quantum technology.

Keywords Quantum computing · Quantum programming · Reverse engineering · KDM · QRev

1 Introduction

Quantum computing has gained importance during the last few years (Gayrard et al., 2019; Panetta, 2019; Piattini et al., 2020) owing to the latest advances in quantum technologies. While traditional computer science applies the rules of Boolean algebra, in which data can

✉ Ricardo Pérez-Castillo
ricardo.pdelcastillo@uclm.es

Extended author information available on the last page of the article

only have a true or false value (i.e. bits), the new quantum computing paradigm is the result of applying certain counterintuitive principles of quantum mechanics to classical computing (i.e. superposition and entanglement) (Rieffel & Polak, 2011). The classical bit has, therefore, evolved, and the qubit is now the basic unit of quantum information that is able to represent various values at the same time (e.g. the spin of an electron or the polarisation of a photon). This is the main foundation of and reason for the exponential speed up that quantum computing brings to us.

The advances in quantum computing have taken place simultaneously with the development of quantum programming languages and their respective compilers (Mueck, 2017). The combination of quantum physics and computer science changes the way in which programs are designed, developed, and executed. While classical computers are deterministic in its core and most classical programs usually compute results deterministically, quantum programs must always address uncertainty and are thus designed and coded in order to explore a narrowed range of possibilities after exploring the search space in a more probabilistic and parallel manner. Most of today's quantum programs are remotely executed on cloud quantum computers, and the results they return are then interpreted by classical programs.

Despite the fact that the quantum computing field is still not sufficiently mature, most the companies are preparing in order to take advantage of this new computing paradigm in an immediate future (Bozzo-Rey et al., 2019; Panetta, 2019). The global market for software and services related to quantum computing will be worth \$1.9bn in 2023, and it is estimated that this figure will have increased to \$8bn by 2027 (Vernacchia & Leap, 2019).

In spite of these predictions, companies will probably not discard their classical, mission-critical information systems. For some problems, quantum computers suffer from the same limitations as classical ones (Aaronson, 2008), while at other times, if the task is quite simple, there is no reason to use quantum computing (a more expensive technology) to address what classical computers can still perform quite successfully. Rather than a complete replacement, classical and quantum computers will operate together in the so-called hybrid information systems (Jiménez-Navajas et al., 2020; Pérez-Castillo et al., 2021). The software evolution of the current classical information systems towards hybrid ones will become a must. However, this research area has not been extensively studied. According to our vision, the software evolution of hybrid systems will bring about at least two important challenges that must be addressed. First, existing quantum algorithms need to be integrated into existing information systems, and the removal of their respective classical counterparts must take place in an integrated manner. Second, the ordinary evolution of classical information systems, e.g. through software reengineering, should consider mechanisms with which to introduce new quantum-based business functionalities into the target hybrid systems.

This paper focuses on the first challenge, and this research particularly addresses a specific problem: the absence of reverse engineering techniques with which to build abstract representations of the quantum programs. These high-level models are important as regard managing the classical and quantum software elements in an abstract and common specification. This might allow the management and exploitation of relationships between both quantum and classical software during software evolution (and in particular, during reengineering or refactoring). This paper extends the previous research published in Jiménez-Navajas et al. (2020). The main contribution of this article is twofold. First, an extension of the reverse engineering tool (with additional technical details) is provided by covering a broader part of Q#, whose syntax and elements are changing continuously. Second, a case

study that applies the tool with various quantum programs to generate KDM models has been conducted to demonstrate its applicability.

This study focuses on a solution based on reengineering and, more specifically, on Architecture-Driven Modernization (Pérez-Castillo et al., 2011a), which is the evolution of traditional reengineering by applying the Model-Driven Engineering (MDE) principles. Among other principles, ADM advocates the usage of the Knowledge Discovery Meta-model (KDM) (Pérez-Castillo et al., 2011b) to represent the aforementioned abstract representations in a standardised manner. The abstraction achieved with the KDM ensures the interoperability between reverse engineering tools and other software modernisation tools. This paper particularly addresses the reverse engineering of quantum software (Q# code) towards KDM. The proposed reverse engineering tool, QRev, consists of a quantum parser that analyses the Q# syntax and builds the corresponding abstract syntax tree (AST). A KDM generator module then inspects the AST and recognises all the quantum elements and their relationships, which are subsequently mapped onto and represented in a KDM model.

It should be noted that a prototype of this tool has already been presented in Jiménez-Navajas et al. (2020). Moreover, the main contribution of this extended paper is a multi-case study conducted with 5 quantum programs. The results of the empirical validation demonstrate that the proposed reverse engineering tool can be used to generate KDM models from Q# code with a precision of 94% and a recall of 89%. The usage of the tool is additionally scalable to larger Q# programs. The main implication of this study is the discovery that well-proven techniques for the reverse engineering of quantum software can help to reduce the gap in the reengineering of classical information systems, and the way in which these hybrid systems will evolve. Thanks to this novel study in the field of quantum software engineering, we demonstrate that it is possible to employ traditional reengineering techniques (which have been proven to be effective in the past) for the evolution of traditional information systems and quantum software toward hybrid information systems. Finally, but of no less importance, it may contribute to the effective adoption of quantum computing.

The remainder of the case study is structured as follows: Sect. 2 introduces the core background concepts, such as quantum computing, software reengineering and KDM. Section 3 summarises other works related to our proposal, while Sect. 4 presents QRev, the reverse engineering tool proposed. Section 5 describes the case study conducted with the application of QRev to various quantum programs and, finally, Sect. 6 shows our conclusions and future work.

2 State of the art

The terms “quantum computing” and “KDM” might seem to be very different to each other. However, in this study, both have the same importance and their combination might acquire higher relevance. This section is divided into three sections. Section 2.1 explains how quantum computing has emerged and evolved, along with the essential points required to understand this research, while Sect. 2.2 describes software reengineering and software modernisation, focusing on how the KDM standard serves to address the modernisation of legacy information systems. Finally, Sect. 2.3 discusses the challenge of managing classical-quantum information systems and how classical information systems can be evolved towards hybrid ones.

2.1 Quantum computing

As stated previously, quantum computing is the result of applying certain counterintuitive principles of quantum mechanics (superposition and entanglement) to computer science. Quantum computing replaces the traditional bit, the smallest unit of information, with a “qubit” or quantum bit. A qubit is usually represented with the electron spin or photons among other subatomic particles. A qubit is a two-state quantum system (not defined exclusively by zero and one, as occurs with a classical bit), i.e. a linear combination of the two base states. This means that a qubit state might be zero and one, or both at the same time. This occurs because of a phenomenon known as superposition (Schrödinger, 1935). Superposition is the cornerstone of the exponential speed up that quantum algorithms can bring to us. Since n qubits can represent 2^n states. For a clearer visualisation of a qubit, it is usually represented with the Felix Bloch sphere.

The qubit’s mathematical definition of its state is represented by a vector following a special notation named *ket-bra* notation. This quantum state is represented with the letter ψ , where $|0\rangle$ and $|1\rangle$ act for depiction of the two different energy levels (which are 0 and 1). The letters α and β indicate the probabilities or proportion of the qubit to be $|0\rangle$ and $|1\rangle$, i.e. $|\psi\rangle = (\alpha|0\rangle + \beta|1\rangle)$. Thus, the probabilities are $\|\alpha\|^2$ for $|0\rangle$ and $\|\beta\|^2$ for $|1\rangle$.

Quantum Computing has a magnificent computational power, but the volatility of the state of its qubits is a challenge to be addressed when coding quantum programs. When a qubit is measured (in order to retrieve its actual value), it collapses and, therefore, loses the superposition state. In order to use that qubit again, most common programming languages provide the special operation ‘reset’ to use again the qubit in the superposition state. Typically, the ‘reset’ operation sets the qubit to the $|0\rangle$ state but it could vary in different platforms. This occurs because quantum computing, and likewise quantum mechanics, belongs to the field of probabilistic physics. This means that one quantum experiment consists of exploring a probabilistic search space and for reporting the global minimum solution, it should be repeated several times. The entanglement principle (firstly defined by Einstein, Podolsky, and Rosen as the EPR paradox (Einstein et al., 1935)) implies the existence of global states of a composite system which cannot be written as a product of the states of individual subsystems (Horodecki et al., 2009). This is useful as regard keeping some actual qubit values while the quantum algorithm continues exploring the probabilistic search space.

Working with qubits and their states is done through the use of quantum circuits and quantum gates (Experience, 2020). Quantum circuits are collections of quantum gates applied to a set of qubits that progressively change the states of such qubits. Quantum gates interact with one or more qubits and change their states in order to vary the probability of being 0 or 1. Some quantum gates may be internally defined as quantum circuits at a lower level of abstraction and still using phenomena of quantum mechanics, such as classical logical gates, in which, for example, the NAND gate is built internally with “AND” and “NOT” gates. One example of atomic gates is the Hadamard gate, which is typically used in order to initialise the qubits, that is, to force the qubit to have the same probabilities of being zero and one or, in other words, change the qubit’s state into that of superposition. Another example of a quantum gate is CNOT, which acts on two qubits, making a comparison between them and changing the status of one qubit depending on that of the other.

Figure 1 shows the teleportation algorithm as an example of a quantum circuit (which is one of the algorithms analysed in the case study). This algorithm supports quantum

Fig. 1 Example of a quantum circuit for the teleportation algorithm

teleportation (Bennett et al., 1993), a technique used to transfer quantum information from source to destination by employing entangled states. In this example, q_0 is the qubit that represents the message to be sent, q_1 is an auxiliary qubit, and q_2 is the target qubit that will receive the information coded in q_0 . In this example, the gates used are Hadamard (H) gates in order to put a qubit into a state of superposition. The Controlled Not gate (CNOT) flips the second (target) qubit if and only if the first (control) qubit is $|1\rangle$. Together with these gates, two measures are taken in qubit q_0 (message) and q_1 (auxiliary) that collapse these qubits and take certain values. At the end of the algorithm, q_2 (target) will have the same value that q_0 had. Both values are measured and stored in classical bits, q_0 into `msg1` and q_2 into `register1`.

A critical question for the field of quantum computing in the near future is “whether quantum devices without error correction can perform a well-defined computational task beyond the capabilities of state-of-the-art classical computers, achieving so-called quantum supremacy” (Boixo et al., 2018). Quantum supremacy is achieved when a formal computational task is performed with an existing quantum device which cannot be performed using any known algorithm running on an existing classical supercomputer in a reasonable amount of time (Arute et al., 2019). There are many participants in this race, but the head-to-head competitor’s race are IBM and Google. In fact, Google recently announced that it has already achieved quantum supremacy with a 53-qubit computer (Quantum & Blog., 2019). Quantum computers have been simultaneously developed with various quantum programming languages, such as Q# (Svore et al., 2018) or QASM (Cross et al., 2017), among others. These programming languages include abstractions with which to build quantum gates and other quantum operations (Garhwal et al., 2019). Most of these programming languages are open source and can be used to code programs that are simulated or executed in remote quantum computers in the cloud, such as the IBM Q experience (IBM, 2020).

2.2 Software reengineering

This section explains the foundations of software reengineering and software modernisation as the solution framed in this research to address the challenge of migrating quantum and classical systems towards the aforementioned hybrid information systems.

Since software technology evolves over time, information systems must evolve continuously. But software evolution can have negative effects on those information systems that were developed in the past, such as degradation or aging, thus making these information

systems *legacy*, signifying that the source code that was developed could be technologically obsolete (Ulrich, 2002). Reengineering allows the preservation of business knowledge, by carrying out the evolutionary maintenance of the legacy information systems with low risks and low costs in comparison with (re)development from scratch (Lucia et al., 2007). Reengineering has proved to be effective to accomplish software evolution processes in which the business knowledge embedded in various software artefacts (especially source code) can be preserved in the target information systems.

The overall reengineering process is typically presented as a “horseshoe” model (Kazman et al., 1998) (see Fig. 2), in which reengineering consists of three main stages:

1. Reverse engineering: the system artefacts (including the source code) are analysed in order to identify its components and interrelationships, and it then creates abstract representations of the system in another form or at a higher level of abstraction.
2. Restructuring: the transformation from one representation form to another at the same relative abstraction level. This stage may consist of refactoring; i.e. the internal structure is improved while preserving the external behaviour of the subject system (functionality and semantics), or it may additionally add new functionality at this abstraction level.
3. Forward engineering: the final stage consists of renovation by generating the new source code and other software artefacts at a lower abstraction level.

Software reengineering projects have traditionally failed when dealing with specific challenges like the standardisation and automation of the reengineering process (Sneed, 2005). First, standardisation constitutes a challenge since the reengineering processes have been typically carried out in many different ad hoc manners. Reengineering projects must, therefore, focus their efforts on a better definition of the process.

Fig. 2 The horseshoe modernisation model (taken from OMG (2016))

Furthermore, the code cannot be the only software asset that the standardisation covers, since “the code does not contain all the information that is needed” (Müller et al., 2000). The reengineering process must be formalised to ensure an integrated management of all of the knowledge involved in the process, such as source code, data, and business rules. Second, automation is also a very important problem. In order to prevent failure in large complex legacy systems, the reengineering process must be mature and repeatable (Canfora & Penta, 2007). In addition, the reengineering process needs to be aided by automated tools in order to enable companies to handle the maintenance costs (Sneed, 2005).

The first approximations of software reengineering attempted to address the aforementioned problems by evolving towards Architecture-Driven Modernization (ADM) (Ulrich & Newcomb, 2010). ADM consists of the application of Model-Driven Engineering (MDE) (Schmidt, 2006) principles so as to facilitate the modernisation of the analysis, refactoring and transformation of an existing system in order to support new requirements, and the migration of systems or even their interoperability. The software reengineering stages are, therefore, accomplished by defining abstract models and the automatic transformations among them. The most frequently models are the computation-independent model (CIM), the platform-independent model (PIM), and the platform-specific model (PSM) (see Fig. 2).

As a part of the ADM initiative, the OMG released the Knowledge Discovery Metamodel (KDM), which was later adopted by ISO/IEC (Pérez-Castillo et al., 2011b). KDM defines a metamodel that represents the software artefacts involved in the legacy information system, providing an accurate view of its functions and structures. Reverse engineering techniques can use KDM to build high-abstraction level models in a bottom-up manner, starting from software legacy artefacts. KDM can be viewed as a metamodel with which to represent the PIM model. KDM can also work as a common interchange format shared by reverse engineering tools, software analysis tools, and any other modernisation tool. KDM models can be physically persisted as an XMI file.

The KDM specification has different perspectives (Pérez-Castillo et al., 2011b) and, in order to simplify the managing of its structure, 4 layers were designed. Each layer is, therefore, based on the previous one, and each of them contains several packages representing different concerns related to legacy information systems. Different KDM packages and layers could be used depending on the artefacts analysed. In order to illustrate how the KDM model is built, we focus on source code. The specific KDM packages employed to represent the source code are ‘Code’ and ‘Action’ in the ‘Program Elements’ layer, whose objective is to provide a language-independent intermediate representation for various constructs determined by common programming languages. The ‘Code’ package represents the named items from the source code and several structural relationships among them, and the ‘Action’ package focuses on behaviour descriptions and the control- and data-flow relationships that they determine. Figure 3 shows the most important meta-elements of these KDM packages. According to KDM, each of the source code artefacts analysed is represented as a ‘CodeModel’ element: the root meta-element. A ‘CodeModel’ is, therefore, composed of a set of code elements (‘AbstractCodeElements’), such as ‘CallableUnit’ and ‘StorableUnit’. The code elements can be interrelated to each other (see ‘AbstractActionRelationships’) through the use of relationships with different semantics, such as ‘Flow’, ‘Calls’, ‘Reads’, and ‘Writes’.

Fig. 3 A simplified view of the Code and Action packages of the KDM metamodel

2.3 The challenges of migrating hybrid information systems

In a near future, organisations will develop hybrid information systems that combine classical and quantum information systems (Jiménez-Navajas et al., 2020; Pérez-Castillo et al., 2021). It is, however, supposed that classical information systems will still be useful. First, it makes sense for classical systems to support simplistic business processes. Otherwise, their implementation in quantum information systems would not make a great difference to performance, while the cost would skyrocket. Second, classical information systems are required in order to manage different requests to quantum computers, and they are also responsible for receiving replies regarding quantum algorithms and translating them into end-user answers.

Since the classical and quantum information systems will operate together, the software evolution of information systems implies a great challenge as regard the migration toward hybrid information systems. In order to accomplish the evolution towards hybrid information systems, reengineering practices must, therefore, be brought into the domain of quantum computing so as to deal with the aforementioned challenges. Reengineering, and software modernisation in general (which can be achieved by adding an MDE approach), can be used to deal with the challenges associated with hybrid information systems. Quantum reengineering must, therefore, address three complementary challenges. The study presented in this paper is framed in the second challenge.

1. Migrate classical legacy information systems towards hybrid architectures that support the integration of classical-quantum information systems.
2. Migrate existing, isolated quantum algorithms and integrate them into the classical information systems to evolve them into hybrid information systems.
3. Transform or add new business operations supported by quantum software that will be integrated into the target hybrid systems.

3 Related work

Not much work related to the reverse engineering of quantum software has been carried out to date. Quantum software engineering is a relatively new discipline (Piattini et al., 2021) that attempts to bring well-known practices and methods from traditional software engineering (among other types of reverse engineering) to quantum software development and programming (Piattini et al., 2020).

Today, there is a need to describe a broader set of quantum circuits beyond the language of qubits and gates. Such tools to represent quantum circuits from the source code are also reverse engineering tools like QRev. However, none of these ad hoc tools provide an extensible and standard-based reverse engineering method as QRev does. The goal of QRev is not, in fact, the representation of graphical quantum circuits from code, but rather the generation of abstract representations of quantum programs (based on KDM) that can be integrated with other elements originating from classical information systems. There are a large number of tools have been developed in order to draw the respective graphical representation of the quantum circuit written in different programming languages. For example, *qasM.2circ* (MIT Quantua Group. Quantum circuit viewer, 2020) converts the syntax of the QASM (Quantum assembler programming language) into a graphical quantum circuit using a quantum assembly language as input, which is a classical reduced-instruction-set computing assembly language extended by a set of quantum instructions based on the quantum circuit model (Svore et al., 2006). This tool is based on Latex, as are other similar tools such as *Quantikz* (Kay, 2020), although this tool does not recreate circuits from any specific quantum Programming language, but only from raw Latex clauses. As occurs with *qasM.2circ*, the IBM Quantum Experience suite (IBM, 2020) also makes it possible to draw quantum circuits from *OpenQASM*, which is IBM's quantum assembly language based on QASM. *OpenQASM 2* was proposed as an imperative programming language for quantum circuits based on earlier QASM dialects. *OpenQASM 3* (Cross et al., 2017) has been consequently proposed with support for arbitrary control flow as well as calling external classical functions. IBM Quantum Experience provides a quantum circuit composer that automatically generates QASM, although the code can also be modified while the circuit is instantly redrawn. *Quantum Programming Studio* (Quantastica, 2019) likewise provides a similar functionality, i.e. bidirectional generation between QASM and quantum circuits. *Qiskit* (qiskit.org, 2020), a quantum software development kit based on Python, also has a module with which to reverse a quantum circuit from the source code.

In order to deal with the software evolution of hybrid information systems, Microsoft has proposed *QIR* (Quantum Intermediate Representation) (Geller, 2020). *QIR* can be used as a bridge between classical and quantum computing, although *QIR* is simply a language rather than a full reverse engineering technique. In some respects, *QIR* solves the same problems as KDM. However, *QIR* is a quantum specific-purpose language based on code syntax, while KDM is more general-purpose specification with which to represent any kind of software at a higher abstraction level.

KDM has never been used for the challenge of migrating quantum–classical information systems. However, KDM has previously been used to solve similar issues in classical software with different programming languages and frameworks. For example, *MODISCO* (Brunelière et al., 2014) is a whole model-driven reverse engineering framework based on KDM. This framework generates KDM models from Java, JSP, or XML code, among other programming languages. KDM has also been used to port iOS to Android mobile apps (Lamhaddab et al., 2019), in which Xcode is inspected and KDM models are generated as

an intermediate representation. Android user interfaces (based on Java) are similarly analysed in order to generate KDM models. MARBLE (Pérez-Castillo et al., 2011c) employs KDM for a business process archaeology method in which KDM is generated from Java code and then used to abstract business process models. In (Pérez-Castillo et al., 2012), executions logs are reversed and represented in KDM with the purpose of transforming them into process mining logs.

With regard to the related work, the main contribution of QRev is the usage of KDM to abstract quantum source code, which enables the integration of reverse engineering tools that also retrieve elements from classical information systems. QRev consequently obtains a common, technology-agnostic knowledge repository (based on KDM) that can integrate both quantum and classical software elements, and the relationships among them.

4 QRev. reverse engineering of Q# to KDM

QRev is the reverse engineering tool designed to generate a KDM from quantum source code written in Q#. QRev is one piece defined within many others that will be developed in long-term research being carried out to address the migration of hybrid systems. The overall quantum reengineering process is first presented in Sect. 4.1, after which Sects. 4.2 and 4.3 present details of the Q# parser and the KDM generator defined. Finally, Sect. 4.4 provides some details of the QRev tool that has been developed.

4.1 Quantum reengineering

Figure 4 shows the overall quantum reengineering process (Pérez-Castillo et al., 2021), which proposes a software modernisation process for hybrid information systems based on well-known standards such as KDM and UML. Figure 4 employs numbers to highlight the three challenges previously discussed in Sect. 2.3.

The reverse engineering stage (which is the scope of this project) consists of analysing existing information systems artefacts such as the source code and database schemas. It could also analyse classical systems (challenge 1) plus quantum programs if these already exist (challenge 2). The output of the reverse engineering phase is a set of KDM models that comprise a KDM repository. As previously explained, this KDM repository holistically represents all the different perspectives and concerns of the legacy information systems in a technology-agnostic manner. This allows the preservation of previous knowledge and business rules, while the impact of the integration of quantum programs is reduced. It is, of course, necessary to extend the KDM (through its ordinary extension mechanisms) so as to enable it to support the representation of aspects of quantum software. It must be noted that although tools that build mathematical abstractions of the programs could be used instead of KDM, those mathematical abstractions do not help to manage the system's previous knowledge in an integrated way as KDM facilitates. Furthermore, KDM is a standard that is supported by several tools, allowing its partial representation and validation.

During the restructuring stage (see Fig. 4), KDM models are (semi)automatically transformed into high-abstraction level models representing the analysis and design aspects of the target hybrid systems. To achieve this, the metamodel employed in this case might be UML. As occurs with the KDM extension, UML should be extended to support the new

Fig. 4 Quantum reengineering process

systems' analysis and design elements concerning quantum computing. In this respect, software engineers can use this UML extension to manually model quantum aspects for new target systems (challenge 3) which are integrated with the existing elements previously gathered by means of reverse engineering.

Finally, the forward engineering stage (see Fig. 4) consists of a set of techniques that are able to generate many parts of the source code for the target hybrid systems. Many well-proven generators for different classical programming languages with which to produce code from UML models currently exist. However, there are no generators for quantum programming languages from high abstraction models. In our opinion, these must be provided and integrated with other existing generative techniques. Neither restructuring nor forward engineering are within of the scope of this research.

QRev is the main contribution of this paper, and its scope is the reverse engineering of quantum programs, particularly for the Q# programming language (i.e. it deals with challenge 2). QRev is divided into two modules. First, a Q# parser takes and analyses Q# files. The parser considers a grammar that will allow Q# to recognise Q# elements and build the respective AST (cf. Sect. 4.2). Second, the KDM generator takes the AST and generates a KDM model (cf. Sect. 4.3). QRev considers some mappings between Q# elements (as defined in the AST) and KDM-compliant elements.

Although QRev focuses on Q#, it has been designed to be extended to other quantum programming languages. It should be noted that there are several frameworks with which to develop quantum software without coding, i.e. graphically, with a quantum circuit composer only. In our case, none of these graphical circuits are within the scope of this proposal, because no code is directly available. Source code is required in order to analyse the syntax and derive the AST. Bearing this in mind, most of these quantum circuit composers are able to translate quantum circuits into the syntax of specific programming languages, signifying that additional parsers could be employed to extend QRev to other programming languages by adding the grammar-based rules for those languages. The same occurs with the output of the proposal, and in those cases in which there is some interest in using another metamodel that is not KDM, such as UML, the code of the KDM Generator can be modified by following the necessary rules.

4.2 Q# parser

The Q# Parser is in charge of building the abstract syntax tree based on the Q# files that it receives. Parser generators can be used to build the Q# parser, and these automatically generate the parser code from the formal definition of language grammars. For example, in this research, we have employed ANTLR (Parr, 2014), a well-known and widely used parser generator. The effort involved in building the Q# parser is consequently reduced when defining the Q# grammar in the ANTLR notation.

It should be noted that Q# was developed by Microsoft by reusing many of the C# syntax elements. We, therefore, began the definition of the Q# grammar as the extension of another existing ANTLR grammar for C#. Nevertheless, several clauses (e.g. loops, method definitions) are completely different in Q#, and other elements related to quantum information are also specific. In particular, the definition and usage of qubits and quantum gates, which are the mechanisms employed in order to work with qubits, needed to be specifically defined in the Q# grammar. The entire definition of the Q# grammar can be viewed in (Pérez-Castillo & Jiménez-Navajas, 2020b). Figure 5 shows an excerpt of the Q# grammar. This example includes the minimum elements required to understand the following example, in which a quantum programme is analysed in order to build the respective AST.

In order to illustrate how the parser works, Fig. 6 shows a short programme written in Q# that implements the quantum circuit previously presented in Fig. 1. The programme first defines the operation *Teleport* in line 6, which receives two qubits. The operation additionally defines the auxiliary qubit in line 7. Lines 8 to 17 apply the quantum gates according to the quantum circuit in Fig. 1. The conditional Z and X gates are applied after qubits *msg* and *register* are respectively measured in lines 12 and 15. Figure 7 shows the respective AST that is generated using the Q# parser developed. This is a representation in the form of a tree with elements of the grammar defined. The leaf elements are both the certain Q# reserved words and the qubit and operation identifiers provided by the user.

4.3 KDM generator

The KDM generator takes the AST generated by the Q# parser and builds a KDM model. The AST is fully inspected by the KDM generator and builds the respective elements for some of the tree nodes in the target KDM model. Different structures and types of KDM elements are chosen and generated depending on the type of tree node. All the AST elements that are important to the architecture of the programme need to be specified in the KDM generator, as do the relationships among those elements.

The generation of quantum elements in the target KDM model is based on the KDM standard. As explained previously, KDM elements are independent of programming languages and technology, which enables them to be annotated with specific stereotypes in order to provide the specific semantics regarding quantum information. For example, KDM does not define quantum entities, such as quantum gates, variables whose defined type are qubits or the way in which to measure them. We, therefore, define a KDM extension family, the default extension mechanism provided by the KDM standard. All the components that exist in quantum programming language are thus represented in this extension family (see lines 3 to 12 in Fig. 8). The extension family groups a set of stereotypes that are then used in the ordinary elements provided by KDM. The definition of this extension family makes it possible to extend the KDM representation of classical information systems towards quantum programming languages, signifying that

LEXER	
QUBIT:	'Qubit';
HADAMARD:	'H';
CNOT:	'CNOT';
PAULIZ:	'Z';
MEASURE:	'M';
RESET:	'Reset';
PARSER	
<pre>statement : labeled_Statement quantum_using (local_variable_declaration local_constant_declaration) ';' embedded_statement quantum_gate_op ; quantum_gate_op : one_gate_op two_gate_op three_gate_op ; quantum_gates_one_op : PAULIX PAULIZ HADAMARD MEASURE RESET ; quantum_gates_two_op : CNOT CONTROLLEDX ; one_gate_op : quantum_gates_one_op OPEN_PARENS (identifier array_def) CLOSE_PARENS quantum_gates_one_op OPEN_PARENS identifier array_def CLOSE_PARENS ;</pre>	

Fig. 5 Excerpt of the Q# grammar defined in (Pérez-Castillo et al., 2021)

relationships among the elements of the classical and quantum programs can be established, which is a key point as regards achieving quantum software reengineering.

Table 1 shows a simplified view of the element mapping applied by the KDM generator. The left-hand column shows elements in the Q# AST, while the two right-hand columns contain both the KDM element and the stereotype (if any) used to annotate the KDM element.

Figure 8 shows the KDM model generated after applying the transformations depicted in Table 1 to the AST model (see Fig. 7) that was generated from the Q# code shown in Fig. 6. The whole KDM model is represented as a *Segment* element (see line 2 in Fig. 8). The extension family discussed above is then added in lines 3 to 12, which is common to every KDM model generated with this proposal. From line 13 to the end (see Fig. 8), a code model is defined using all the compilation units analysed, defined

```

1 namespace Microsoft.Quantum.Samples.Teleportation {
2     open Microsoft.Quantum.Intrinsic;
3     open Microsoft.Quantum.Canon;
4     open Microsoft.Quantum.Measurement;
5
6     operation Teleport (msg : Qubit, target : Qubit) : Unit {
7         using (register = Qubit()) {
8             H(register);
9             CNOT(register, target);
10            CNOT(msg, register);
11            H(msg);
12            if (MResetZ(msg) == One) {
13                Z(target);
14            }
15            if (IsResultOne(MResetZ(register))) {
16                X(target);
17            }
18        }
19    }
20 }

```

Fig. 6 Teleport algorithm coded in Q# (adapted from Microsoft. (2020))

in rows (see line 14). The compilation unit analysed might contain a *LanguageUnit* element that defines the specific data types used in the compilation unit. A sequence of code elements (with *type* = *ActionElement*) is then defined for all the statements in the Q# code. For example, the first Hadamard gate applied (see line 8 in Fig. 6) is transformed into a KDM block (see lines 22 to 26 in Fig. 8). The *actionRelation* elements are used to define the qubit operands of the quantum gate (*type* = *Address*), along with the flow to the next statement in the Q# code (*type* = *Flow*). It is important to note that code snippets are attached to all the code elements so as to prevent the loss of the semantics

Fig. 7 Abstract Syntax Tree generated for the Q# code of the teleportation algorithm

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <kdm:Segment xmlns:kdm="http://www.omg.org/spec/KDM/20160201/kdm"
  xmlns:action="http://www.omg.org/spec/KDM/20160201/action"
  xmlns:code="http://www.omg.org/spec/KDM/20160201/code" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmi:version="2.0"
  name="Teleportation.kdm">
3   <extensionFamily xmi:id="id.0">
4     <stereotype name="quantum programming language" xmi:id="id.1" />
5     <stereotype name="quantum program" xmi:id="id.2" />
6     <stereotype name="quantum operation" xmi:id="id.3" />
7     <stereotype name="quantum gate" xmi:id="id.4" />
8     <stereotype name="qubit" xmi:id="id.5" />
9     <stereotype name="qubit measure" xmi:id="id.6" />
10    <stereotype name="control qubit" xmi:id="id.7" />
11    <stereotype name="qubit array" xmi:id="id.8" />
12  </extensionFamily>
13  <model xmi:id="id.9" xmi:type="code:CodeModel" name="Teleportation.kdm">
14    <codeElement xsi:type="code:CompilationUnit" name="Teleportation.qs" xmi:id="id.10">
15      <codeElement xmi:type="code:LanguageUnit" name="Q# Common definitions" xmi:id="id.11">
16        <codeElement xmi:id="id.12" xmi:type="code:PrimitiveType" name="Primitive" />
17        <codeElement xmi:id="id.13" xmi:type="code:IntegerType" name="Integer" />
18        <codeElement xmi:id="id.14" xmi:type="code:BooleanType" name="Boolean" />
19      </codeElement>
20      <codeElement xmi:type="code:CallableUnit" name="Teleport" xmi:id="id.15"
        stereotype="id.3">
21        <codeElement xmi:type="code:StorableUnit" xmi:id="id.16" name="register"
          stereotype="id.5" />
22        <codeElement xmi:type="action:ActionElement" xmi:id="id.17" kind="operator"
          name="Hadamard" stereotype="id.4">
23          <source language="Q#" snippet="H(register)" />
24          <actionRelation xsi:type="action:Addresses" from="id.17" to="id.16" />
25          <actionRelation xsi:type="action:Flow" from="id.17" to="id.18" />
26        </codeElement>
27        <codeElement xmi:type="action:ActionElement" xmi:id="id.18" kind="operator"
          name="Controlled Not" stereotype="id.4">
28          <source language="Q#" snippet="CNOT(register,target)" />
29          <actionRelation xsi:type="action:Addresses" from="id.18" to="id.16" />
30          <actionRelation xsi:type="action:Flow" from="id.18" to="id.19" />
31        </codeElement>
32        <codeElement xmi:type="action:ActionElement" xmi:id="id.19" kind="operator"
          name="Controlled Not" stereotype="id.4">
33          <source language="Q#" snippet="CNOT(msg,register)" />
34          <actionRelation xsi:type="action:Addresses" from="id.19" to="id.16" />
35          <actionRelation xsi:type="action:Flow" from="id.19" to="id.20" />
36        </codeElement>
37        <codeElement xmi:type="action:ActionElement" xmi:id="id.20" kind="operator"
          name="Hadamard" stereotype="id.4">
38          <source language="Q#" snippet="H(msg)" />
39          <actionRelation xsi:type="action:Addresses" from="id.20" />
40          <actionRelation xsi:type="action:Flow" from="id.20" to="id.21" />
41        </codeElement>
42        <codeElement xmi:type="action:ActionElement" xmi:id="id.21" kind="operator"
          name="Pauli Z" stereotype="id.4">
43          <source language="Q#" snippet="Z(target)" />
44          <actionRelation xsi:type="action:Addresses" from="id.21" />
45          <actionRelation xsi:type="action:Flow" from="id.21" to="id.22" />
46        </codeElement>
47        <codeElement xmi:type="action:ActionElement" xmi:id="id.22" kind="operator"
          name="Pauli X" stereotype="id.4">
48          <source language="Q#" snippet="X(target)" />
49          <actionRelation xsi:type="action:Addresses" from="id.22" />
50        </codeElement>
51      </codeElement>
52    </codeElement>
53  </model>
54 </kdm:Segment>

```

Fig. 8 KDM file generated from the Q# code of the teleportation algorithm

so characteristic in reverse engineering techniques (see lines 23, 28, 33, 38, 43, or 48 in Fig. 8). The remaining Q# statements are generated in the KDM model in a similar way.

Table 1 Q# AST-KDM mapping applied using the KDM generator

AST elements	KDM element	KDM stereotype
Q# Program	CodeModel	
Q# Program	CompilationUnit	Quantum program
Q# Operation	CallableUnit	Quantum operation
Qubit_Declaration	DefinedType	Qubit
Quantum_Gate	ActionElement	Quantum gate
Qubit_identifier (used as operand in a quantum_gate)	ActionRelation (type = Addresses)	
* among AST elements appearing in a sequence	ActionRelation (type = Flow)	

4.4 Integration of the Q# parser and KDM generator

In order to facilitate the empirical validation of the proposed reverse engineering tool, in addition to demonstrating its applicability, QRev has been developed as a REST API (implemented in Java) that shows a service operation employed to transform Q# files and generate KDM model files.

As mentioned previously, the tool consists of two main components, the Q# parser and the KDM generator. The Q# Parser was developed using ANTLR, which is a powerful parser generator for the reading, processing, execution, or translation of structured text or binary files (Parr, 2014). The grammar of the Q# parser was specified in ANTLR syntax. This grammar divides the Q# syntax into two elements: (i) the *lexer*, which defines the lexicon and reserved words of the language; and (ii) the *parser*, which specifies the structures of the language. Since Q# is based on C# and ANTLR has an open repository of defined languages, such as Java, C, and C++, rather than starting from scratch, we took an ANTLR project for C# and adapted it to Q# by defining the necessary key words, structures and modules.

The KDM generator was implemented in Java. It receives the AST generated by the Q# parser as input, and inspects the whole tree node by node using the JDOM library. The generator manages all the elements and their references in Hash maps in order to define relationships between KDM elements. The previously detailed mapping between code elements and KDM elements is applied by employing the tool. If one of the code elements represents quantum information, such as qubit declarations or quantum gates, the respective stereotype attribute is added to the respective KDM element. These stereotypes indicate the extension family previously detailed and are necessary for the future extension of classical information systems towards hybrid ones.

5 Case study

This section presents a detailed case study, which demonstrates that it is, by means of reverse engineering, feasible to transform programs developed in the quantum programming language Q#, into KDM models. The creation of such abstract models of quantum software helps to evolve classical systems toward hybrid ones, which combine both quantum and classical computing paradigms.

5.1 Case study design

The case study was designed and conducted by following the protocol proposed by Runeson et al. (Runeson et al., 2012). The following subsections are structured according to that protocol. The rationale and objective of the study are first explained in Sect. 5.1.1, after which the theoretical framework with the overall context of the study is discussed in Sect. 5.1.2 and research questions addressed by the case study are then presented in Sect. 5.1.3. Sect. 5.1.4 defines the concepts and relevant measures used in the case study, after which the criteria employed to select certain quantum programs are explained and the cases eventually selected are depicted in Sect. 0. Finally, the case study protocol and data storage are shown in Sect. 5.1.6.

5.1.1 Rationale and objective of the study

As explained previously, companies will need to adapt their classical information systems in order to take advantage of the new quantum computing paradigm while critical business knowledge is preserved. In this context, reverse engineering is one important part of the whole quantum reengineering process. As a result, the rationale of this case study is to determine whether abstract models representing quantum software can be built by inspecting the quantum source code.

The specific objective is to determine whether QRev is able to generate valid and useful KDM models by analysing Q# code.

5.1.2 Theoretical framework

The theoretical framework of the study consists of the analysis presented in Sect. 3, which shows the previous studies concerning reverse engineering techniques and tools and how reengineering has contributed to the migration of legacy information systems alongside new quantum algorithms. As the literature review states, the studies combining reverse engineering and quantum software are currently limited. Moreover, owing to the current volatility of quantum programming languages and related frameworks, a broader theoretical generalisation is difficult.

5.1.3 Research questions

This study defines two research questions, RQ1 and RQ2. RQ1 concerns the KDM models generated by the tool, and attempts to evaluate whether the models comply with the standard, and whether those models correctly define all the quantum code elements. The objective of RQ2 is, meanwhile, to assess the scalability of the tool. RQ1 is, therefore, more related to effectiveness, while RQ2 is associated with the efficiency of QRev.

RQ1. Is QRev able to generate accurate and complete KDM models?

RQ2. Is QRev able to generate KDM models in a scalable manner?

5.1.4 Concepts and measures

In this case study, several measures have been considered in order to answer the research questions and reach the correct conclusions at the analysis stage. Table 2 provides a summary of the study variables, indicating for each one: (i) the research question that it will help to answer, (ii) the concept that will be measured, (iii) a short definition (or formula) of the variable, (iv) the scale type, and (v) the range definition for all the possible values.

In order to answer RQ1, which is related to the concept of effectiveness, the proposed measures are defined for the effectiveness of both the Q# parser and the KDM generator. It is not simple to measure the effectiveness of the parser, since the outgoing result is an AST. It, therefore, simply measures whether or not the input file is parsed. With regard to the KDM Generator, unlike the previous case, there are certain KDM elements in the output file that can be analysed in order to compute missing elements (false negatives), and irrelevant elements (false positives), signifying that precision, recall and the *F*-measure can be used. Precision represents the pertinent recuperated elements inside the set of recovered elements in the KDM model, while recall represents the degree of total relevant elements that have been retrieved. Finally, the *F*-measure is the weighted harmonic mean of precision and recall. RQ2 is, meanwhile, related to the concept of efficiency and attempts to determine whether the tool could be used for larger programs. The division previously carried out for RQ1 has been also applied in RQ2 for this purpose. It considers the time values used to parse the input Q# files and generate the KDM files. Finally, the last two measures are aggregated in the total time spent. The size of the Q# files is also measured in terms of the number of lines of source code (LOC), as an independent variable.

5.1.5 Case selection

As this case study evaluates the functioning of the QRev tool as regard generating KDM models, it is, among other concerns, necessary to select which cases are going to be used for this evaluation. Table 3 shows the different criteria used to select the different cases.

C1 originated from the scope of the study, since the QRev tool was specifically developed in order to analyse Q# syntax. Despite being a quantum programming language, Q# includes some structures from C#, a classical programming language, and it could, therefore, be the case that a Q# file does not include any quantum programming building blocks. This was the reason for considering C2, which evaluates the usage of elements such as qubits and quantum gates. C3 was established in order to limit the complexity of the cases regarding the definition of syntax rules originating from external libraries, which is not within the scope of this study. Examples of the libraries commonly used in Q# programs are Maths or Chemistry. Finally, C4 originated from the fact that there are different ways in which to implement the same algorithm in Q#, and this criterion, therefore, ensures that only the algorithms implemented and designed by Microsoft are analysed. This criterion ensures that the programme can be accessed from a reputed and open repository, and will consequently contribute to the replicability and extensibility of this case study.

After applying the selection of cases based on the criteria mentioned in the previous section, five programs from the Microsoft Quantum Development Kit were selected. For

Table 2 Definition of variables

Id	Concept	Measure	Definition	Scale	Range
RQ1	Case	Code programme	Each programme file with the Q# source code	Nominal	
RQ2					
RQ1	Efficacy of the parser	Parsed programme	Whether or not the Q# programme has been parsed	Nominal	{0-Error, 1-Parsed}
	Efficacy of the KDM Generator	Precision	$\frac{ (\text{relevantelements}) \cap (\text{recoveredelements}) }{ (\text{recoveredelements}) }$	Ratio	$x \in [0.0, 1.0]$
		Recall	$\frac{ (\text{relevantelements}) \cap (\text{recoveredelements}) }{ (\text{relevantelements}) }$	Ratio	$x \in [0.0, 1.0]$
		<i>F</i> -measure	$\frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}}$	Ratio	$x \in [0.0, 1.0]$
		Time parsing	Time spent parsing the file	Ratio	$x \in \mathbb{R}^+$
		Time generating KDM	Time spent generating the KDM file based on the abstract syntax tree	Ratio	$x \in \mathbb{R}^+$
RQ2	Efficiency	Total time	Time parsing + time generating KDM	Ratio	$x \in \mathbb{R}^+$
		Lines of code	Lines of code generated	Ratio	$x \in \mathbb{N}$

Table 3 Criteria for case selection

Id	Criterion for case selection
C1	It must be written in Q# with extension 'qs'
C2	It should use quantum elements (such as quantum gates and qubits)
C3	It must not use Chemistry or Maths libraries
C4	It must be accessed in a Microsoft repository

each programme, Table 4 shows the programme name, LOC, the source to be accessed, and a brief description of the program.

5.1.6 Case study protocol, data storage, and analysis

The study protocol defines how the selected cases were analysed and the derived data was collected and eventually analysed.

First, all the Q# files corresponding with the cases under study were accessed and stored in the analysis environment. This environment consisted of a set of automated tests (implemented with Junit) with which to execute QRev with all the cases being studied. As a result, various KDM models were generated and stored for later analysis. At this point, some of the cases could not be recognised by the Q# parser. For those KDM models generated, both the KDM elements and the respective Q# programme elements were analysed in parallel. At this point, missing elements in the KDM model were counted (i.e. those Q# elements that could not be abstracted in KDM), along with irrelevant elements (i.e. those KDM elements that did not represent any important Q# elements in the respective programme). These numbers were then employed to calculate precision, recall, and the *F*-measure. All these values were stored in the main dataset, together with the parsing and generating times. These values were directly reported by QRev. The main dataset containing the raw data of the case study can be viewed in Pérez-Castillo and Jiménez-Navajas (2020a).

In order to complete the whole dataset, the application of the tool was done in a laptop with an AMD Ryzen 5 35000U with 2.10 GHz, 8 GB of RAM and running on an SSD.

Having completed the main dataset, various analyses were performed in order to answer the research questions. The study employed descriptive statistics to answer both RQ1 and RQ2. A regression model was additionally used to analyse the scalability in RQ2. Regression models analyse relationships among variables. The linear regression model considered the two measures previously depicted as being the dependent variable and the independent variable. The correlation coefficient ρ (which is between -1 and 1) was computed by applying Pearson's rank correlation test. Pearson's rho (ρ) is the degree to which the actual values of the dependent variable are near to the predicted values. This linear regression model considered the two time-related measures previously depicted as the dependent variable, and the size as the independent variable.

5.2 Analysis and interpretation

This section presents the results of the analysis carried out for each research question, along with the main insights derived.

Table 4 Programs analysed in the case study

Case	Program name	LOC	Source	Description
P1	QRNG.qs	28	Microsoft [2020b]	A quantum random number generator
P2	TeleportationSample.qs	45	Microsoft and Quantum teleportation sample [2020c]	It implements the teleportation algorithm in order to interchange information between two qubits
P3	BernsteinVazirani.qs	46	Microsoft [2020d]	It implements the Bernstein-Vazirani algorithm
P4	Measurement.qs	66	Microsoft [2020e]	It provides basic examples of qubit measurement operations
P5	DeutschJozsa.qs	44	Microsoft [2020d]	It implements the Deutsch-Jozsa algorithm

5.2.1 RQ1. effectiveness of QRev

Table 5 summarises the results obtained for the five programs being studied. As can be seen, all the programs were properly parsed, signifying that the success of the Q# parser was 100%, and five respective KDM models were generated.

In order to compute precision, recall, and the *F*-measure (as depicted in Sect. 5.1.4), various elements were manually counted for each case being studied (see Table 5).

- The total number of lines of source code of each program.
- The number of quantum elements in the source code was counted as the expected relevant elements. For example, each qubit declaration or assignment counted as one single quantum element, and quantum gates were counted as one or more, depending on the qubits involved in the gate (e.g. CCNOT counted as two elements).
- The number of KDM elements was counted in a similar way in the output model.
- The number of missing elements in the KDM (regarding the input Q# file) was also counted. These elements are considered to be the false negatives, since these were elements that were expected to be modelled in the KDM model but were not.
- The number of irrelevant elements in the KDM models, i.e. the false positive elements that were generated in the KDM model.

Precision ranges between 0.92 (P1) and 0.98 (P2), while recall varies between 0.85 (P1) and 0.93 (P5). Both values are high, which signifies that the KDM generator is effective. If the values are analysed in detail, it will be noted that recall is slightly lower than precision. This is owing to the fact that various false negatives (missing elements) were reported, signifying that various quantum elements were not represented in the output KDM model. These missing elements were specific Q# operations that were not considered during the design of the Q# parser, and it was not, therefore, possible to transform them into the KDM model. These missing elements were the following:

- *MresetZ*, which is an operation of Q# used to measure a qubit, return its value, and reset it to $|0\rangle$. It is present in P1.
- *ApplyToEach*, which is an operation defined by Q# that accepts two parameters, a quantum gate, and a qubit array. The quantum gate is applied to every qubit in a row. It is used in P3 and P4.
- *AssertMeasurementProbability*, which is an auxiliary operation used to measure qubits in simulated environments, in which probabilities can be provided. It is used in P4.

Although these special Q# operations were not considered, the recall value for these five programs is an average of 0.89. Overall, the average *F*-measure is 0.92, and it is, therefore, possible to state that the effectiveness of the reverse engineering tool is 92%. We believe that this result allows research question RQ1 to be answered positively, and we can conclude that QRev is effective.

5.2.2 RQ2. efficiency of QRev

The objective of research question RQ2 was to assess the efficiency of the reverse engineering tool and, in particular, whether QRev is scalable to larger quantum programs.

Table 5 Effectiveness results for the programs under study

Case	parsed?	Lines of code	# Quantum elements	# KDM Elements	# Missing elements in KDM	# Irrelevant elements in KDM	Precision	Recall	F-measure
P1	✓	28	6	12	2	1	0.92	0.85	0.88
P2	✓	45	29	47	4	1	0.98	0.92	0.95
P3	✓	46	14	24	3	1	0.96	0.88	0.92
P4	✓	66	20	29	4	3	0.90	0.87	0.88
P5	✓	44	10	29	2	1	0.97	0.93	0.95
Mean									
Standard deviation									
							0.94	0.89	0.92
							0.03	0.04	0.03

Table 6 shows the overall results obtained for the time measures taken in the case study. For each system, Table 6 provides (i) the number of lines of source code; (ii) the time (in milliseconds) spent parsing the Q# file and (iii) generating the KDM model; (iv) the total time as the sum of the two previous values, along with (v) the percentage of time spent parsing and (vi) generating the KDM model. The results show that all the programs were transformed into a KDM model in less than one second, which suggests that QRev could be applied in real environments with a reasonable time. It should be noted that most of the time was spent parsing the Q# file and building the AST, while the KDM generation from the AST took only around 4% of the total time.

In order to analyse the scalability, Fig. 9 provides the scatter plot for the parsing time (which is around 96% of the total time), along with the total time. The scatter plot also provides the trend line according to the linear regression computed using the Pearson method. The correlation coefficients were, in both cases, moderate (0.39 and 0.49 respectively). The interpretation of these results suggests that the time spent for larger quantum programs will grow linearly. Although more programs should be analysed in order to strengthen this evidence, we can, therefore, preliminarily conclude that QRev will be scalable to larger quantum programs.

5.3 Evaluation of validity

This section discusses threats to the validity of the case study and possible actions by which to mitigate them.

5.3.1 Construct validity

Construct validity attempts to ensure that the measures used in the case study are appropriate in order to draw conclusions that are useful to answer research questions. In this study, we have employed precision, recall, and the *F*-measure, which have been widely used to evaluate many information retrieval approaches. In this respect, the only threat is the way in which these metrics were calculated. The researchers involved in this study manually inspected the input programs and output models to determine missing and irrelevant elements as the basis on which to compute precision and recall. The researchers' subjective opinion could have entailed a bias. In order to mitigate this threat, we established certain criteria with which to count missing and irrelevant elements (cf. Sect. 5.2.1).

Moreover, various time measures were used to evaluate the efficiency of QRev. One threat to the validity of this could be that there are no benchmarks with which to compare the time

Table 6 Efficiency results for the programs studied

Case	LOC	Q# parsing time (ms)	KDM generation time (ms)	Total time (ms)	% Q# parsing time	% KDM generation time
P1	28	585	18	603	97%	3%
P2	45	555	22	577	96%	4%
P3	46	578	23	600	96%	4%
P4	66	640	28	668	96%	4%
P5	44	554	30	584	95%	5%

Fig. 9 Scatter plot for the Q# parsing and total KDM generation time

values obtained in this study. Efficiency was, therefore, evaluated in terms of the scalability by carrying out a correlation analysis of the programme size and the time-based measures.

5.3.2 Internal validity

The study included only five programs, which is definitively not a large population from which to attain statistically representative results. It is, nevertheless, possible to extract preliminary insights demonstrating the effectiveness and efficiency of QRev. It should be borne in mind that there is not, as yet, a huge code base for quantum programs, at least from reputable sources. It was, therefore, difficult to include further programs in this study. In the future, we hope to include additional programs so as to extend the case study in order to obtain stronger evidence than that reported in this study.

Another important factor that threatens the internal validity is the specific implementation of the proposed QRev tool. If QRev is implemented in a different way, the results may be different. For example, some specific Q# expressions were not included in either the Q# grammar definition or the tool, which clearly affected the recall values. If these elements were supported in another tool, these values could vary.

5.3.3 External validity

The generalisability of the study is the main focus of the external validity. Strictly speaking, according to the tool used and the case selection criteria, the results obtained in this case study can be generalised only to Q# source code that is inspected and abstracted into

KDM models. In fact, one of the selection criteria removed from the study was that of those programs that used other quantum libraries in Microsoft QDK. The usage of these libraries may be of interest in future case studies, signifying that further case studies are necessary in order to strengthen the generalisation to a greater extent.

5.3.4 Reliability

Finally, reliability concerns the extent to which data and the analysis are dependent on the researchers who conducted the study. As stated above, the researchers' subjective opinion when computing precision, recall, and the F -measure may have affected reliability. In order to mitigate this and increase reliability, this study provides the tool used in the study, the programs used as input, and the KDM models generated as output in order to ensure the replicability of the study. Another way in which to mitigate the aforementioned threat was the selection of quantum programs from reputed sources (such as the Microsoft repository) rather than using Q# examples created by the researchers.

6 Conclusions and future work

This paper presents QRev, a reverse engineering tool with which to abstract KDM models from the inspection of Q# programs. This tool has been empirically validated through the use of a multi case study involving five quantum programs coded in Q#. The preliminary insights derived from the case study are that QRev is effective and scalable to larger quantum programs. The case study, therefore, demonstrates the suitability of QRev and contributes to its applicability in industry.

The main achievement of QRev is that abstract models representing information systems artefacts (such as source code) can be modelled using KDM. KDM makes it possible to represent elements originating from both classical and quantum information systems, along with the relationships among some of those elements. Since all this knowledge is represented in an abstract manner, KDM is independent of the different classical or quantum technology and programming languages, signifying that other quantum programs coded in different programming languages could be integrated. We consider that this could be a major milestone for the quantum software engineering community, because thanks to the presented tool we have opened a new line of research for the evolution of traditional information systems towards hybrid information systems through reengineering.

The benefit of KDM models lies in the fact that they can be used in the different stages of software modernisation (or reengineering) processes, thus, for example, facilitating the refactoring of a system or the introduction of new functionalities to be coded as quantum software. QRev, therefore, considerably contributes to the challenge of investigating how reengineering and software modernisation can help to merging both classical and quantum information systems towards hybrid ones.

QRev currently has room for improvement. As stated previously, we have already detected some possible enhancements. For example, we intend to provide support as regard recognising specific building blocks of Q# (such as *MresetZ*, *ApplyToEach*, *AssertMeasurementProbability*). Moreover, case studies should be conducted with further quantum programs or random circuits automatically generated with which to stress the tool, along with other case studies that will analyse the inspection of classical and quantum source code and their combination in KDM models.

With regard to our future research, QRev is framed in long-term research into quantum software reengineering, and the researchers, therefore, expect to work on the following reengineering stages discussed in Sect. 4.1. For instance, we expect to develop a UML extension with which to represent analysis and design models of the target hybrid information system. Some automatic model transformations between KDM and UML may also subsequently be developed. Finally, the greatest challenge we shall confront will be how to transfer the ideas and techniques supported by QRev into a real environment, such as QPath (aQuantum. QPath., 2020), a Quantum Software Development, and Lifecycle Application Platform.

We also expect to incorporate in the future more quantum programming languages to be parsed, like Open QASM or python. However, not only quantum programming languages are supposed to be analysed, but also source code developed in classical programming languages or as a combination of both. For this reason, adding new parsers and adapting the KDM generator will be accomplished in the future for considering the integration of quantum and classical elements in the KDM models. This is related to other future research line about how to introduce quantum routines into existing classical information systems.

Acknowledgements This study is part of the SMOQUIN project (PID2019-104791RB-I00) funded by the *Ministerio de Ciencia e Innovación* (MICINN), along with the QHealth Project: Quantum Pharmacogenomics Applied to Aging,” 2020 CDTI Missions Programme (Center for the Development of Industrial Technology of the Ministry of Science and Innovation of Spain) and FEDER. We would like to thank all the aQuantum members, and particularly Guido Peterssen and Pepe Hevia, for their support.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aaronson, S. (2008). The limits of quantum. *J Scientific American*, 298(3), 62–69.
- Arute, F., et al. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779), 505–510.
- aQuantum. QPath. (2020) *The quantum software development and lifecycle application platform*. [cited 2020 26/11/2020]; Available from: <https://www.quantumpath.es/>
- Bennett, C. H., Brassard, G., Crépeau, C., Jozsa, R., Peres A., & Wootters, W. K. (1993). Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, 70(13), 1895.
- Boixo, S., Isakov, S. V., Smelyanskiy, V. N., Babbush, R., Ding, N., Jiang, Z., et al. (2018). Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6), 595–600.
- Bozzo-Rey, M., Longbottom, J., & Müller, H. A. (2019). Quantum computing: challenges and opportunities, in *Proceedings of the 29th annual international conference on computer science and software engineering*, IBM Corp. Toronto, Ontario, Canada. 393–394.
- Brunelière, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information Software Technology*, 56(8), 1012–1032.
- Canfora, G., & Penta, M. D. (2007). New frontiers of reverse engineering, in *2007 future of software engineering*. IEEE Computer Society.
- Cross, A. W., Bishop, L. S., Smolin, J. A., & Gambetta, J. M. (2017). Open quantum assembly language.
- De Lucia, A., Ferrucci, F., Tortora, G., & Tucci, M. (2007). *Emerging methods, technologies, and process management in software engineering*, 1–276.

- Einstein, A., Podolsky, B., & Rosen, N. J. (1935). Can quantum-mechanical description of physical reality be considered complete? 47(10), 777.
- Garhwal, S., Ghorani, M., & Ahmad, A. (2019). Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering*.
- Gayrard, R., Carrière, F., Darde, B., & Jaunay, J. (2019). Quantum computing: ready for the huge leap? <https://www.wavestone.com/app/uploads/2019/10/Quantum-Computing-WavestoneFrance-Digitale-EN-web-2019.pdf>
- Geller, A. (2020). *Introducing Quantum Intermediate Representation (QIR)*. 23rd September 2020 [cited 2020 23/11/2020]; Available from: <https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/>
- Horodecki, R., Horodecki, P., Horodecki, M., & Horodecki, K. (2009). Quantum entanglement. 81(2), 865.
- IBM. (2020). *IBM Q Experience*. Available from: <https://quantum-computing.ibm.com/>
- Jiménez-Navajas, L., Pérez-Castillo, R., & Piattini, M. (2020). Reverse engineering of quantum programs toward KDM models, in *13th International Conference on the Quality of Information and Communications Technology (QUATIC)*. Springer International Publishing: Faro, Portugal (Online Conference), 249–262.
- Kay, A. (2020). Tutorial on the Quantikz Package. <https://arxiv.org/pdf/1809.03842.pdf>
- Kazman, R., Woods, S. G., & Carriere, S. J. (1998). Requirements for integrating software architecture and reengineering models: CORUM II. *Reverse Engineering - Working Conference Proceedings*, 154–163.
- Lamhaddab, K., Lachgar, M., & Elbaamrani, K. (2019). Porting Mobile Apps from iOS to Android: A Practical Experience. *Mobile Information Systems*, 2019, 4324871.
- Mueck, L. (2017). Quantum software. *Nature*, 549(7671), 171–171.
- Microsoft. *Quantum code samples*. (2020a). [cited 2020 23/11/2020]; Available from: <https://github.com/microsoft/Quantum/tree/main/samples>
- Microsoft QDK. *Creating random numbers with quantum computing*. (2020b). Available from: <https://docs.microsoft.com/es-es/samples/microsoft/quantum/creating-random-numbers-with-quantum-computing/>
- Microsoft QDK. *Measuring Qubits*. (2020e). Available from: <https://docs.microsoft.com/es-es/samples/microsoft/quantum/measuring-qubits/>
- Microsoft QDK. *Quantum teleportation sample*. (2020c). Available from: <https://docs.microsoft.com/es-es/samples/microsoft/quantum/quantum-teleportation/>
- Microsoft QDK. *Simple quantum algorithms sample*. (2020d). Available from: <https://docs.microsoft.com/es-es/samples/microsoft/quantum/simple-quantum-algorithms/>
- MIT Qantua Group. *Quantum circuit viewer: qasm2circ*. (2020). [cited 2020 20/11/2020]; Available from: <https://www.media.mit.edu/quanta/qasm2circ/>
- Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M. -A., Tilley, S. R., & Wong, K. (2000). Reverse engineering: a roadmap, in *Proceedings of the conference on the future of software engineering*. ACM: Limerick, Ireland.
- OMG. (2016). Architecture-driven modernization (ADM): Knowledge discovery meta-model (KDM), v1.4. <https://www.omg.org/spec/KDM/1.4/PDF>
- Panetta, K. (2019). The CIO's guide to quantum computing. <https://www.gartner.com/smarterwithgartner/the-cios-guide-to-quantum-computing/>
- Parr, T. (2014). ANTLR. [cited 2020 26/11/2020]; Available from: <https://www.antlr.org/>
- Pérez-Castillo, R., De Guzmán, I. G. R., & Piattini, M. (2011b). Knowledge discovery metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards and Interfaces*, 33, 519–532.
- Pérez-Castillo, R., de Guzmán, I. G. -R., & Piattini, M. (2011c). Business process archeology using MARBLE. *Information and Software Technology*, 53(10), 1023–1044.
- Pérez-Castillo, R., Serrano, M. A., & Piattini, M. (2021). Software modernization to embrace quantum technology. *Advances in Engineering Software*, 151, 102933.
- Pérez-Castillo, R., & Jiménez-Navajas, L. (2020a). QRev case study material. [cited 2020 25/11/2020]; Available from: <https://github.com/ricpdc/smoquin/tree/master/qkdm-api/test/casestudy>
- Pérez-Castillo, R., de Guzmán, I. G. R., & Piattini, M. (2011a). Architecture-driven modernization, in *Modern software engineering concepts and practices: Advanced Approaches*. IGI Global. 75–103.
- Pérez-Castillo, R., Weber, B., García Rodríguez de Guzmán, I., & Piattini, M. (2012). Integrating event logs into KDM repositories, in *27th Annual ACM Symposium on Applied Computing (SAC'12)*. ACM: Riva del Garda (Trento), Italy. 1095–1102.
- Pérez-Castillo, R., & Jiménez-Navajas, L. (2020b). ANTLR Q sharp grammar. Available from: <https://github.com/ricpdc/smoquin/tree/master/qkdm-api/src/es/alarcos/qkdm/parser/grammar/qsharp>
- Piattini, M., Peterssen, G., Serrano, M. A., Hevia, J. L., & Pérez-Castillo, R. (2021). Towards a quantum software engineering. *IT Professional*, 23(1), 62–66.
- Piattini, M., Peterssen, G., & Pérez-Castillo, R. (2020). Quantum computing: A new software engineering Golden Age. 45(J SIGSOFT Softw. Eng. Notes), 12–14.

- Piattini, M., et al. (2020). The Talavera manifesto for quantum software engineering and programming, in *QANSWER 2020. QuANtum SoftWare Engineering & pRogramming*, M. Piattini, et al., Editors. CEUR-WS: Talavera de la Reina. 1–5.
- Quantastica. (2019). Quantum Programming Studio. [cited 2020]; Available from: <https://quantum-circuit.com/quiskit.org>.
- Quiskit SDK. (2020). Quiskit SDK. Available from: <https://qiskit.org/>
- Rieffel, E. G., & Polak, W. H. (2011). *Quantum computing: A gentle introduction (scientific and engineering computation)*, The MIT Press.
- Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). Case study research in software engineering: Guidelines and examples. John Wiley & Sons.
- Schmidt, D. C. (2006). Developing applications using model-driven design environments. *IEEE Computer Society*, 39(2), 25–32.
- Schrödinger, E. (1935). Die gegenwärtige Situation in der Quantenmechanik. *Die Naturwissenschaften*, 23, 807–812.
- Sneed, H. M. (2005). Estimating the costs of a reengineering project. *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE Computer Society. 111–119.
- Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I., & Markov, I. L. (2006). A layered software architecture for quantum computing design tools. *Computer*, 39(1), 74–83.
- Svore, K., et al. (2018). Q#: Enabling scalable quantum computing and development with a high-level DSL, in *Proceedings of the real world domain specific languages workshop 2018*. 2018, Association for Computing Machinery: Vienna, Austria. p. Article 7.
- Ulrich, W. M. (2002). *Legacy systems: Transformation strategies*.
- Ulrich, W. M. & Newcomb, P. H. (2010). Information systems transformation. *Information Systems Transformation*.
- Vernacchia, S. (2019). Quantum Leap. Advancing a strategy for quantum computing that will inspire, support and safeguard economic growth in the Middle East. <https://www.pwc.com/m1/en/world-government-summit/documents/wgs-quantum-leap.pdf>, in *World Government Summit 2019*. PricewaterhouseCoopers (PwC) 44.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ricardo Pérez-Castillo Ricardo Pérez-Castillo holds the Ph.D. degree in Computer Science from the University of Castilla-La Mancha (Spain). He works at the IT & Social Sciences School of Talavera at University of Castilla-La Mancha. His research interests include architecture-driven modernization, model-driven development and business process archaeology. Currently member of the aQuantum scientific research team where he works on the migration of classical systems to quantum architectures and quantum software reengineering.

Luis Jiménez-Navajas Luis Jiménez-Navajas graduate in Computer Science from the University of Castilla-La Mancha (Spain). Current member of the scientific research group aQuantum, working on the software modernization of classical-quantum information systems with specific focus on software reverse engineering.

Mario Piattini Mario Piattini. MSc. (1989) and Ph.D. (1994) in Computer Science from Madrid Technical University (UPM). PMP, CISA, CISM, CGEIT and CRISC. Founder-Director of the Information Systems and Technologies of the UCLM (University of Castilla-La Mancha) and of the UCLM-INDRA Research and Development Joint Center. Between the “15 Top scholars in the field of systems and software engineering (2004-2008)” and the “Among the 15 “Most active experienced SE researchers (2010-2017)”. Full Professor of Software Engineering at UCLM, and leader of the Alarcos Research Group and scientific director of AQCLab, S.L. (first ENAC / ILAC accredited laboratory for software and data quality based on ISO 25000). Currently leader of the aQuantum scientific research team (Alarcos Group).

Authors and Affiliations

Ricardo Pérez-Castillo^{1,2} · Luis Jiménez-Navajas¹ · Mario Piattini¹

Luis Jiménez-Navajas
luis.jimeneznavajas@uclm.es

Mario Piattini
mario.piattini@uclm.es

¹ Information Technology & Systems Institute (ITSI), University of Castilla-La Mancha, Paseo de la Universidad 4 13071, Ciudad Real, Spain

² Faculty of Social Sciences & IT, Avda. Real Fábrica de Seda, University of Castilla-La Mancha, Talavera de La Reina, 45600 s/n, Spain