Computación cuántica: Pruebas de mutación
Quantum computing: Mutation testing

Director: Manuel Núñez García

Codirectora: María de las Mercedes García Merayo

Autores: Luis Aguirre Galindo & Javier Pellejero Ortega

Trabajo de fin de grado del Doble Grado en Matemáticas e Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Curso 2019-2020

I ain't no physicist but I know what matters.

– Popeye el Marino

Resumen (español)

A lo largo de la última década, la *computación cuántica* ha ido ganando protagonismo en el campo científico. Con ello, han surgido multitud de herramientas enfocadas a este nuevo área y lenguajes de programación de naturaleza cuántica. A su vez, en una sociedad tan tecnológica como la actual, la comprobación del correcto funcionamiento del software desarrollado se ha convertido en una necesidad imperiosa. De esta forma aparece todo un ámbito de la Ingeniería del Software dedicado a cubrir esta necesidad: *Testing de Software*. Así, surge la idea de tomar una de las técnicas más importantes de este campo, el *mutation testing*, y tratar de trasladarla al mundo cuántico aplicándola a dos de los lenguajes de computación cuántica más relevantes en el momento: Q# y *Qiskit*. Para ello, se ha desarrollado un sistema en Java que recibe código de estos dos lenguajes y aplica *mutation testing*. Además, si bien en la versión actual sólo se puede ejecutar código de Q# y *Qiskit*, el sistema ha sido desarrollado bajo la premisa de poder añadir otros lenguajes cuánticos en el futuro.

A lo largo de este trabajo se explicarán con detalle las nociones más importantes de la computación cuántica, así como de la disciplina del testing, haciendo especial énfasis en las pruebas de mutación. También se detalla la planificación seguida para la fase de desarrollo, así como la estructura y el funcionamiento interno del programa. Por último se muestra la ejecución del programa para uno de los algoritmos cuánticos más importantes: *el algoritmo de Deutsch-Jozsa*.

Palabras Clave: Computación cuántica, *testing* cuántico, pruebas de mutación, Microsoft Q#, IBM Qiskit, herramienta Java.

Abstract (English)

Over the last decade, Quantum Computing has been gaining prominence in the scientific field. With this, a full spectrum of tools focusing on this new area and programming languages of a quantum nature have emerged. At the same time, in a society as technological as the current one, the validation of the correct bevaviour of the developed software has become a must. In this way, a whole area of Software Engineering is devoted to covering this need: *Software Testing*. Thus, the idea arises of taking one of the most important tools of testing, *mutation testing*, and trying to transfer it to the quantum world in order to apply it to two of the most relevant quantum computing languages: Q# and *Qiskit*. To do this, a system has been developed, using Java, that allows us to take code from both languages and apply *mutation testing*. Furthermore, although in the current version it is only possible to execute code of Q# and *Qiskit*, the system has been developed under the premise of being able to add other quantum languages in the future.

Throughout this work, the most important notions of Quantum Computing will be explained in detail, as well as the discipline of testing, with special emphasis on mutation testing. The planning followed for the development phase is also detailed, as well as the structure and internal functioning of the program. Finally, the execution of the program for one of the most important quantum algorithms is shown: the *Deutsch-Jozsa algorithm*.

keywords: Quantum Computing, quantum testing, mutation testing, Microsoft Q#, IBM Qiskit, Java tool.

Prefacio

En esta memoria queremos reflejar las nociones básicas de las pruebas de mutación aplicadas a la computación cuántica. Por nuestra condición de estudiantes del doble grado de Matemáticas e Ingeniería Informática, la computación y teoría de la información cuántica representa un área de confluencia de las Matemáticas, la Física y la Informática muy apropiada para abordar en un TFG. En total se han desarrollado 3 Trabajos de Fin de Grado en relación a este tema: dos realizados para la Facultad de Matemáticas, por cada uno de los integrantes de este grupo, y este mismo. Haremos referencia a este hecho durante la memoria pues, debido al tema común de los trabajos, tareas como la investigación y la planificación se elaboran de manera conjunta. Queremos añadir que creemos que este proyecto sería más completo y *redondo* si se hubiera combinado en un único TFG tal y como se venía haciendo hasta el curso pasado. Tanto los conocimientos reflejados como la extensión de esta posible combinación deberían ser suficientes para satisfacer los requisitos que proponen cualquiera de las dos facultades. Con el modo actual, cada investigación por separado pierde valor y se hace más complicado reflejar todos los contenidos abarcados y lo más importante, la conexión entre ellos. Nos hemos esforzado para que esta memoria sea autocontenida y no requiera de otros documentos, pero si el lector tiene interés, recomendamos la lectura de nuestras memorias presentadas como TFGs en la Facultad de Matemáticas porque conforman un pilar sólido para obtener una introducción a la computación cuántica.

Consideramos que los prerrequisitos para leer esta memoria se restringen a nociones realmente básicas de álgebra lineal y computación clásica. Presentamos los elementos necesarios, a un nivel fundamental, en las tres grandes áreas que se consideran en este TFG (computación cuántica, *mutation testing* y *mutation testing* aplicado a la computación cuántica), no sin antes realizar una introducción breve sobre los antecedentes de estas materias y algunas referencias actuales.

Una parte vital de la investigación que presentamos en esta memoria es el proyecto de desarrollo de *software* que hemos llamado *Mutation Testing for Quantum Computing* (MTQC). Destinamos un capítulo a narrar su gestión, planificación, desarrollo y funciona-

miento. El programa principal en el que se apoya esta TFG está escrito en *Java* aunque también se utilizan otros lenguajes como *Python*, *Qiskit* y `Q#`. Hemos procurado que su desarrollo sea lo más modular posible para que pueda tener continuidad como proyecto al ser relativamente fácil implementar nuevos lenguajes y funcionalidades. En la parte final de esta memoria realizamos pruebas de mutación, utilizando MTQC sobre programas cuánticos escritos en *Qiskit* y `Q#`, y establecemos algunas conclusiones sobre la investigación y el proyecto, además de aportar algunas ideas que mejorarían MTQC y que podrían ser desarrolladas en un futuro cercano.

Queríamos agradecer a nuestro tutores Mercedes G. Merayo y Manuel Núñez la proposición de un tema tan interesante y completo para elaborar nuestro TFG. Si bien nos ha hecho invertir mucho tiempo y tal vez otro tema hubiera resultado más sencillo y breve, nos gustaría pensar que estos conocimientos puedan ser una punta de lanza, en el mundo académico y laboral, en un futuro cercano si la tecnología permite el desarrollo de computadores cuánticos cada vez más potentes. Por último, nos gustaría agradecer a nuestros padres la inversión y el apoyo para poder realizar nuestra formación académica universitaria y, en general, por aguantarnos.

Índice general

1. Introducción histórica **1**
1.1. Antecedentes históricos y actualidad (español) 1
1.2. Historical and current background (English) 2

2. Introducción a la computación cuántica **5**
2.1. Notación de Dirac 5
2.2. Qubit 6
2.2.1. Medición de un qubit 6
2.2.2. Experimento: Distribución de Clave Cuántica 7
2.3. Múltiples Qubits 9
2.3.1. Producto Tensorial 9
2.4. Puertas cuánticas 13
2.4.1. Puerta cuánticas de un qubit 13
2.4.2. Puertas cuánticas de más de un qubit 15
2.4.3. Teorema de no clonación 17

3. Introducción al Testing **19**
3.1. ¿Qué es el Testing? 19
3.2. Mutation Testing 21

4. Mutation Testing en computación cuántica **25**
4.1. Diseño de operadores de mutación 25
4.2. Especificación de inputs cuánticos 26
4.3. Decisión de la muerte de un mutante 28
4.3.1. Valoración estadística de los outputs 28
4.3.2. Estado interno del simulador 29
4.4. Mutation Testing en *Qiskit* y `Q#` 30

5. Gestión, planificación y desarrollo del software **33**

5.1. Gestión del proyecto 33
5.1.1. Gestión de equipo 33
5.1.2. Contribución al proyecto de Luis Aguirre 34
5.1.3. Contribución al proyecto de Javier Pellejero 36
5.1.4. Gestión de configuración 38
5.2. Planificación 39
5.2.1. Planificación temporal 39
5.2.2. Modelo de proceso 41
5.3. MTQC: Mutation Testing for Quantum Computing 42
5.3.1. Principales funcionalidades 42
5.3.2. Diseño 43

6. Ejemplo de uso de MTQC **55**
6.1. Pruebas de mutación sobre Deutsch-Jozsa 55
6.1.1. Algoritmo Deutsch-Jozsa 55
6.1.2. Deutsch-Jozsa en MTQC 58

7. Conclusiones **63**
7.1. Conclusiones (español) 63
7.2. Conclusions (English) 64
7.3. Implementaciones futuras 64

Bibliografía **66**

Capítulo 1

Introducción histórica

1.1. Antecedentes históricos y actualidad (español)

A principios del siglo XX se produjo un giro de guión en el mundo de la Física con la aparición de la teoría de la mecánica cuántica. Más adelante, hacia mediados de siglo, aparecen los primeros artículos sobre computación. No sería hasta 1982 cuando Richard Feynman trató de unificar el mundo de la computación con el mundo cuántico, planteando cómo se podrían representar sistemas físicos a través de computadores [Feynman 1982]. Unos años más tarde, en 1985, David Deutsch propone el concepto de *universal quantum computing*, una máquina con una serie de propiedades no reproducibles por las máquinas de Turing clásicas [Deutsch 1985]. Sin embargo, todavía no era claro que este tipo de computador cuántico pudiese competir, a nivel de rendimiento, con un computador clásico. Hubo que esperar hasta 1992 para que David Deutsch y Richard Jozsa propusieran un problema muy particular cuya solución clásica se ve mejorada exponencialmente mediante el uso de un algoritmo cuántico [Deutsch y Jozsa 1992]. Aunque esta solución se diese desde un marco teórico, se comenzaba a entrever ya el potencial de la computación cuántica. A pesar de ello, todavía no había un algoritmo cuántico que mejorase la solución a un problema real, pues el problema propuesto por David Deutsch y Richard Jozsa se desarrollaba en un escenario muy particular. Fue en 1994 cuando Peter Shor [Shor 1994] mostró un algoritmo cuántico capaz de factorizar enteros de manera eficiente, poniendo en jaque al sistema criptográfico de clave pública RSA.

A raíz del algoritmo propuesto por Peter Shor, el interés en la computación cuántica se ha ido incrementando con el paso de los años. Se realizan las primeras demostraciones experimentales de algoritmos cuánticos sobre una máquina física [Jones, Mosca y Hansen 1998] y aparecen los primeros computadores potencialmente escalables formados por *qubits*

[Häffner y col. 2005]. En los últimos años grandes empresas tecnológicas como IBM, Microsoft o Google han entrando en juego. Sería esta última la que en 2019 anuncia haber alcanzado la *supremacía cuántica* [Arute y col. 2019], es decir, por primera vez un ordenador cuántico mejora sustancialmente el rendimiento frente a un computador clásico de manera empírica.

La computación clásica no se vio frenada por los avances en el mundo cuántico. De hecho, el continuo desarrollo de programas cada vez más complejos incrementó la necesidad de comprobar el correcto funcionamiento de los mismos. Así surgió una rama de la Ingeniería del Software, el *testing*, que ha ido evolucionando a lo largo de los años con diversas técnicas tales como la que consideramos en esta memoria: *mutation testing*. Recientemente ha aparecido un interés en la aplicación de testing a programas cuánticos [Polo Usaola 2020] aunque nos encontramos en una fase muy preliminar, más enfocada en identificar los campos de aplicación que en desarrollar nuevas teorías. En esta línea, se está estudiando como adaptar las técnicas de testing ya desarrolladas para programación clásica al mundo de la programación cuántica, respetando las particularidades de este último. Sin embargo, no conocemos ninguna herramienta de testing para validar el comportamiento de programas cuánticos.

De esta forma, surge la idea de desarrollar una herramienta que permita aplicar una técnica de testing, *mutation testing*, sobre dos de los lenguajes de programación cuánticos más relevantes en el momento: el lenguaje de IBM *Qiskit* y el lenguaje de Microsoft `Q#`.

1.2. Historical and current background (English)

In the beginning of the XX century the physics field suffered a sudden change with the outbreak of quantum mechanics. Few decades after, the world witnesses the first computation articles. It was not until 1982 when Richard Feynman tried to unify the computation world with the physics one, raising how physics systems could be implemented via computers [Feynman 1982]. A few years after, in 1985, David Deutsch came up with the idea of *universal quantum computing*, a machine with some special properties that could not be replicated by classic Turing machines [Deutsch 1985]. However, it was not obvious that a quantum computer could face performance-wise a classic machine. We would had to wait until 1992 when David Deutsch and Richard Jozsa suggest a very particular problem for which solution could exponentially benefit from applying a quantum algorithm [Deutsch and Jozsa 1992]. Even though this solution was just proposed in a theoretical way, the potential of quantum programming started to be notable. Despite this progress, there was not a quantum algorithm which improved the solution for a real problem, because the Deutsch and Jozsa problem was too concrete. It was in 1994 when Peter Shor [Shor 1994] came up with a quantum algorithm capable of factorizing prime whole numbers, challenging the

security of the RSA public-key cryptosystem.

As a result of the algorithm proposed by Peter Shor, interest in quantum computing has increased over the years. The first experimental demonstrations of quantum algorithms on a physical machine are carried out [Jones, Mosca and Hansen 1998] and the first potentially scalable computers formed by qubits appear [Häffner et al. 2005]. In recent years large technology companies such as IBM, Microsoft or Google have come into play. It is the latter that in 2019 announces having achieved *quantum supremacy* [Arute et al. 2019], i.e. for the first time a quantum computer substantially improves performance compared to a classic computer in an empirical way.

On another front, classic computation was not slowed by the outbreaks on the quantum field, and with the continuous software developement, the need of verifying these programs suffered a boost. That is how *Testing* appeared as a new area in Software Engineering. Testing has evolved over the years with new techniques, such as the one considered in this report:*mutation testing*. Recently there has been a small increase in interest in the application of testing to quantum programs [Polo Usaola 2020] athough we are in a very preliminary phase, more focused on identifying fields of application than on developing new theories. In this line, we are studying how to extrapolate the testing techniques already developed for classical programming to the world of quantum programming, taking into account the particularities of the latter. However, we do not know any testing tools to validate the behavior of quantum programs.

In this way, the idea arises to develop a tool that allows the application of one of these techniques, *mutation testing*, on two of the most relevant quantum programming languages at the moment: IBM *Qiskit* language and Microsoft `Q#` language.

Capítulo 2

Introducción a la computación cuántica

En este capítulo vamos a introducir al lector en el mundo de la computación cuántica. No nos adentraremos en la teoría de la mecánica cuántica y usaremos nociones básicas de matemáticas, concretamente del álgebra lineal. Existen multitud de fuentes que ahondan en el conocimiento aportado por las Matemáticas, la Física y las Ciencias de la Computación para cimentar la teoría que vamos a presentar a continuación. Remitimos al lector a ellas si existe el deseo de conocer más sobre esta rama de la ciencia. En particular, los dos TFGs de los miembros de este grupo presentados para el grado de Matemáticas contienen más explicaciones sobre los distintos conceptos teóricos que subyacen a la computación cuántica. En primer lugar, presentaremos una notación muy usada en mecánica cuántica y que ese utiliza a lo largo de este trabajo.

2.1. Notación de Dirac

La notación $|\psi\rangle$ denominada *ket* pertenece a la notación de *Dirac* y representa al vector ψ de cierto espacio vectorial complejo como columna, mientas que $\langle\psi|$, denominado *bra*, representa al conjugado de ψ como una fila. Por tanto, $\langle\psi'|\,|\psi\rangle$ o $\langle\psi'|\psi\rangle$ denota el *producto escalar complejo* dado por

$$\langle\psi, \psi'\rangle = \sum_{i=1}^{n} \psi_i \overline{\psi'_i}$$

Denotamos por $|\psi\rangle\langle\psi|$ al *producto exterior*. Por la definición anterior dadas para *bra* y *ket*, se deduce que el resultado es una matriz de dimensiones $n \times n$, donde n es la dimensión del

espacio complejo donde habita ψ. Al tratarse de una matriz cuadrada, podemos identificarla como la matriz asociada con un isomorfismo lineal $\mathbb{C}^n \to \mathbb{C}^n$.

2.2. Qubit

El *qubit* o *cúbit* es el sistema de información más básico de la computación cuántica. Se trata de un vector unitario del espacio vectorial $\mathbb{C}^2$ con una base ortonormal prefijada que denotamos por $\{|0\rangle, |1\rangle\}$. A menudo, a estos vectores ortonormales se les identifica con dos vectores de $\mathbb{C}^2$, habitualmente $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ y $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Esta elección de la base no se realiza de manera arbitraria. De hecho, los estados $\{|0\rangle, |1\rangle\}$ nos ayudarán a representar los valores de los bits clásicos 0 y 1. A diferencia de los bits, un qubit puede encontrarse en una *superposición* de los estados de la base, es decir, un qubit $|\psi\rangle$ se expresa como:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \text{ donde } \alpha, \beta \in \mathbb{C}.$$

A los valores α y β que aparecen en la expresión anterior se les conoce como *amplitudes* del estado $|0\rangle$ y $|1\rangle$, respectivamente. Como un qubit es un vector unitario, los valores α y β deben cumplir la siguiente propiedad (usualmente conocida como la *restricción de normalización*):

$$|\alpha|^2 + |\beta|^2 = 1$$

2.2.1. Medición de un qubit

Un concepto muy importante asociado con el tratamiento de un qubit es el proceso de obtención de información que nos puede proporcionar. Dicho proceso se conoce como *medición*. Pese a que un qubit se pueda encontrar en una superposición de estados de la base mencionada, la información que podemos extraer de este no es más que un valor de bit clásico. Esto se debe a que para obtener esta información hay que realizar los que se conoce como *medida* sobre el qubit. Cuando se mide un qubit, este colapsa a uno de los dos estados de la base $\{|0\rangle, |1\rangle\}$ y, por lo tanto, al igual que con los bits clásicos, solo hay dos posibles resultados.

Dado un qubit en el estado $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$, la probabilidad de obtener el estado $|0\rangle$ o $|1\rangle$ viene determinada por el cuadrado de las amplitudes de ambos. De esta forma, se tiene que $|\alpha|^2$ representa la probabilidad de obtener el estado $|0\rangle$, mientras que $|\beta|^2$ es la probabilidad de obtener el estado $|1\rangle$ al realizar una medición. Tras la medición, el estado actual es el obtenido por la medición. Este proceso es irreversible: una vez realizada la medición, no se puede recuperar el estado original del qubit.

Cabe destacar que la elección de la base en la que medimos no es única. Una base alternativa relevante es $\{|+\rangle, |-\rangle\}$, donde $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ y $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Se verifica que $|+\rangle$ y $|-\rangle$ son ortonormales entre sí.

Ejemplifiquemos todo esto suponiendo que tenemos el estado $|\psi\rangle = |+\rangle$ al que vamos a aplicar una medición. Si lo hacemos respecto de la base $\{|0\rangle, |1\rangle\}$ entonces obtendremos con una probabilidad idéntica del 50 % cada uno de los dos elementos de la base: $|0\rangle$ o $|1\rangle$. Sin embargo, si medimos respecto de la base $\{|+\rangle, |-\rangle\}$ obtendremos un 100 % de las veces el resultado $|+\rangle$.

Llegados a este punto, el lector puede preguntarse cuales son las ventajas de utilizar qubits frente a bits clásicos si la información que podemos obtener de ellos sigue siendo binaria. Por ello, vamos a mostrar algunos ejemplos muy relevantes a lo largo de este capítulo que muestran el potencial de los qubits.

2.2.2. Experimento: Distribución de Clave Cuántica

Hemos hablado en el capítulo anterior del logro que supone el algoritmo de factorización de enteros de Shor. Este algoritmo podría poner en entredicho la seguridad de la computación clásica actual basada principalmente en el algoritmo de *RSA* que sustenta su confianza en la complejidad del problema de factorizar grandes números. Surge así el interés por descubrir nuevos algoritmos criptográficos que, mediante el uso de la computación cuántica, resuelvan este problema. Vamos a hablar sobre el primer algoritmo de clave cuántica, *BB84*, cuyo esquema fue planteado por primera vez en 1984 por Bennett y Brassard [Bennett y Brassard 1987].

Supongamos que Alice y Bob quieren comunicarse de manera privada empleando para ello una clave secreta. Disponen de un canal clásico bidireccional y de otro cuántico unidireccional que parte de Alice hacia Bob. El problema es que una tercera persona, Eve, tiene acceso a ambos canales sin que ellos lo sepan (ver figura 2.1). Además, Eve puede no sólo observar el canal cuántico, sino que también puede tomar las partículas que pasen por él, medirlas y reenviarlas a Bob.

El primer paso del algoritmo es la elección de dos bases, como $\{|0\rangle, |1\rangle\}$ y $\{|+\rangle, |-\rangle\}$ y Alice procede a preparar una secuencia de bits. Para cada bit se elige aleatoriamente una de las bases para codificarlos, de manera que dicha codificación queda

$$\text{bien } \begin{matrix} 0 \to |0\rangle \\ 1 \to |1\rangle \end{matrix}, \text{ o bien } \begin{matrix} 0 \to |+\rangle \\ 1 \to |-\rangle \end{matrix}$$

en función de la base elegida. Tras la codificación, Alice envía a Bob la secuencia de qubits generada. Bob procede ahora a medir eligiendo, para cada qubit, una de las dos bases

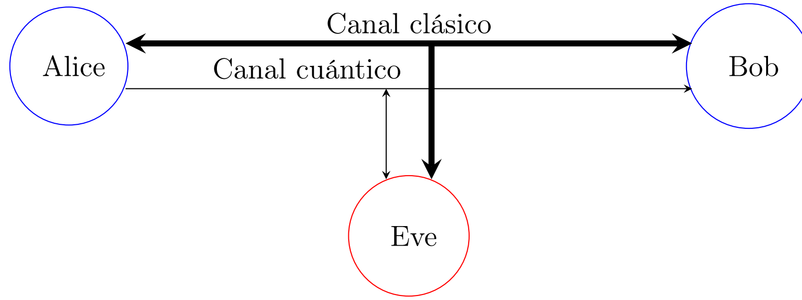


Figura 2.1: Esquema de distribución de clave cuántica.

anteriormente citadas. Tras esta medición, tanto Alice como Bob hacen públicas las bases en las que la primera codificó y el segundo midió. Alrededor del 50 % de cada elección de estas bases coincidirá y los resultados asignados a estas elecciones se validan, mientras que el resto se desechan.

Número del qubit	Base en la que codificó Alice	Base en la que midió Bob
1	$\|0\rangle, \|1\rangle$	$\|+\rangle, \|-\rangle$
2	$\|0\rangle, \|1\rangle$	$\|0\rangle, \|1\rangle$
3	$\|+\rangle, \|-\rangle$	$\|+\rangle, \|-\rangle$
4	$\|0\rangle, \|1\rangle$	$\|+\rangle, \|-\rangle$
5	$\|+\rangle, \|-\rangle$	$\|+\rangle, \|-\rangle$
6	$\|+\rangle, \|-\rangle$	$\|0\rangle, \|1\rangle$
7	$\|+\rangle, \|-\rangle$	$\|0\rangle, \|1\rangle$
8	$\|0\rangle, \|1\rangle$	$\|0\rangle, \|1\rangle$
9	$\|0\rangle, \|1\rangle$	$\|0\rangle, \|1\rangle$
10	$\|0\rangle, \|1\rangle$	$\|+\rangle, \|-\rangle$
...	...	...

Cuadro 2.1: Ejemplo de bases tomadas por Alice y Bob.

En el cuadro 2.1 se muestra un ejemplo de esta validación para los 10 primeros qubits de una secuencia. En verde aparecen los qubits para los que la base elegida por ambos coincidió y por tanto la medición de Bob arrojó el mismo resultado que la codificación de Alice. En rojo, por el contrario, aparecen aquellos cuyas bases no coincidieron y se descartan por no aportar información.

Ahora Alice y Bob pueden revelar una pequeña cantidad de los valores obtenidos por cada uno, por ejemplo los n primeros. Si todos esos valores coinciden, el canal es seguro y pueden utilizar el resto de los valores medidos no desvelados como clave.

¿Qué hubiera pasado si Eve hubiera intervenido el canal y hubiera tratado de medir los qubits antes de que lo hiciera Bob? Al igual que Bob, Eve desconoce la base elegida por Alice para cada qubit, luego tendría que elegir una aleatoriamente y realizar una medición. En el caso de escoger la misma que Alice (aunque Eve no tendría la certeza hasta después de haber acertado), no sólo podrá conocer el valor que codificó sino que además no está cambiando el estado del qubit y podrá reenviarlo a Bob inalterado. Esto ocurrirá en el 50 % de los casos; sin embargo, en la otra mitad habrá elegido la base incorrecta modificando el estado del qubit y si Bob sí acierta con la base de Alice sólo tendrá un 50 % de posibilidades de medir el mismo valor que Alice codificó cuando debería serlo del 100 %.

Es así que Eve está introduciendo una variación en los estados en el 50 % de los qubits lo que supone que los valores de Alice y Bob diferirán en un 25 %. Si la cantidad de mediciones reveladas por ambos es de $n = 20$ tras el intento de Eve de interceptar la comunicación, las probabilidades de no darse cuenta de la interferencia (no detectando ningún error) es de $0,75^{20} \approx 0,32\,\%$. Tomando $n = 30$ la probabilidad de pillar a Eve asciende a aproximadamente un 99,98 %, con lo que la certeza de garantizar la seguridad del canal es prácticamente absoluta con un tamaño de n no excesivamente grande.

2.3. Múltiples Qubits

A lo largo de esta sección veremos cual es el comportamiento de un sistema cuántico cuando tenemos más de un qubit. Es aquí donde realmente podremos atisbar la capacidad de cómputo que tienen este tipo de máquinas. En la física tradicional, si tenemos un sistema con n partículas, cada una de ellas representada por un vector de dimensión 2, el espacio vectorial que se genera es de dimensión $2 \cdot n$. Esto se debe a que cada partícula se comporta de manera independiente y por tanto con ser capaces de modelizar el comportamiento individual de cada partícula tendremos representado todo el sistema. Es por ello que las dimensiones aumentan de manera lineal. Sin embargo, esto no ocurre en mecánica cuántica. Las partículas ya no se comportan de manera independiente, sino que hay una dependencia entre ellas. Este efecto se llama *entrelazamiento* y entraremos a explicarlo en detalle posteriormente.

Es por este motivo por el que no podemos modelizar los sistemas cuánticos con el comportamiento individual de cada partícula, si no que necesitaremos de una herramienta más potente: el *producto tensorial*.

2.3.1. Producto Tensorial

Supongamos que tenemos dos espacios vectoriales $\mathcal{V}$ y $\mathcal{W}$ de dimensiones n y m, respectivamente. Entonces, $\mathcal{V} \otimes \mathcal{W}$ es un espacio vectorial de dimensión $n \times m$ y representa el producto tensorial de ambos espacios. Además, supongamos que tenemos bases ortogonales

para $\mathcal{V}$ y $\mathcal{W}$ dadas por:

$$\begin{aligned} \mathcal{B}_v &= \{|v_i\rangle \mid 1 \leq i \leq n\} \\ \mathcal{B}_w &= \{|w_j\rangle \mid 1 \leq j \leq m\} \end{aligned}$$

Entonces una base $\mathcal{B}$ del espacio vectorial $\mathcal{V}\otimes\mathcal{W}$ se obtiene mediante el producto tensorial de los vectores de $\mathcal{B}_v$ y $\mathcal{B}_v$, es decir:

$$\mathcal{B} = \{|v\rangle \otimes |w\rangle \mid |v\rangle \in \mathcal{B}_v, |w\rangle \in \mathcal{B}_w\}$$

Normalmente se aligera la notación y denotaremos $|\phi\rangle \otimes |\psi\rangle$ simplemente por $|\phi\rangle\, |\psi\rangle$ o incluso por $|\phi\psi\rangle$.

Por definición, el producto tensorial cumple las siguientes tres propiedades [Nielsen y Chuang 2001].

1. Dado un escalar z y dos vectores $|v\rangle \in \mathcal{V}$ y $|w\rangle \in \mathcal{W}$, se cumple:

$$z(|v\rangle \otimes |w\rangle) = (z\,|v\rangle) \otimes |w\rangle = |v\rangle \otimes (z\,|w\rangle)$$

2. Dados dos vectores $|v_1\rangle, |v_2\rangle \in \mathcal{V}$ y un vector $|w\rangle \in \mathcal{W}$, se cumple:

$$(|v_1\rangle + |v_2\rangle) \otimes |w\rangle = |v_1\rangle \otimes |w\rangle + |v_2\rangle \otimes |w\rangle$$

3. Dado un vector $|v\rangle \in \mathcal{V}$ y dos vectores $|w_1\rangle, |w_2\rangle \in \mathcal{W}$, se cumple:

$$|v\rangle \otimes (|w_1\rangle + |w_2\rangle) = |v\rangle \otimes |w_1\rangle + |v\rangle \otimes |w_2\rangle$$

Hemos visto el producto tensorial aplicado a espacios vectoriales y hemos definido las propiedades que debe verificar un operador para ser considerado tensorial. Veamos cómo aplicaremos este operador en el caso de vectores, aunque lo haremos de una manera más general, definiendo el producto tensorial entre matrices. No olvidemos que un vector tiene una representación matricial como columna.

Sean A y B matrices con coeficientes complejos de dimensiones $m \times n$ y $p \times q$ respectivamente. Se verifica

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix}$$

Por tanto, la matriz resultante $A\otimes B$ tiene dimensiones $(m\cdot p)\cdot(n\cdot q)$. Así, sean dos qubits cuyos espacios vectoriales están representados respectivamente por la base ortonormal estándar $\{|0\rangle, |1\rangle\}$. Entonces, el nuevo espacio vectorial generado por ambos qubits tiene como

base los vectores $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, es decir, un espacio de dimensión 4. Sin embargo, este ejemplo no es demasiado ilustrativo (pues $2^n = 2n$ si $n = 2$). Así que supongamos que añadimos un tercer qubit. Entonces el nuevo espacio vectorial tendrá dimensión 8 (2^3) y su base viene dada por los vectores $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$. Recordemos en este momento que un qubit tiene también una representación como vector columna. Por lo tanto podemos aprovecharnos de dicha representación para calcular productos tensoriales entre qubits. Supongamos que tenemos dos qubits $|\psi\rangle$ y $|\phi\rangle$ que cumplen:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle\,,\ \ |a|^2 + |b|^2 = 1$$
$$|\phi\rangle = c\,|0\rangle + d\,|1\rangle\,,\ \ |c|^2 + |d|^2 = 1$$

Estos qubits se pueden ver en forma de matriz como:

$$|\psi\rangle = \begin{pmatrix} a \\ b \end{pmatrix}$$
$$|\phi\rangle = \begin{pmatrix} c \\ d \end{pmatrix}$$

Finalmente, esta representación nos permite calcular el producto tensorial de los dos qubits, $|\psi\rangle \otimes |\phi\rangle$, como:

$$|\psi\rangle \otimes |\phi\rangle = \begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}$$

Recuperando la notación de Dirac, este vector sería $|\psi\rangle \otimes |\phi\rangle = ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle$. De esta forma, se ha obtenido un vector de dimensión 4 que, además, sigue cumpliendo la restricción de normalización ya que:

$$\begin{aligned} |ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 &= \\ |a|^2|c|^2 + |a|^2|d|^2 + |b|^2|c|^2 + |b|^2|d|^2 &= \\ |a|^2(|c|^2 + |d|^2) + |b|^2(|c|^2 + |d|^2) &= \\ |a|^2 + |b|^2 &= 1 \end{aligned}$$

Una vez que hemos visto como combinar qubits mediante el producto tensorial, es imprescindible mostrar que existen ciertos estados cuánticos que no se pueden expresar mediante un producto tensorial. Este tipo de estados son conocidos como *estados entrelazados*. Un ejemplo de este tipo de estados es el dado por $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Veamos que, efectivamente,

no se puede obtener dicho estado mediante el producto tensorial de dos qubits. Supongamos que dichos qubits existen. Entonces tenemos:

$$(a\,|0\rangle + b\,|1\rangle) \otimes (c\,|0\rangle + d\,|1\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Además, por lo visto anteriormente, se tiene que:

$$(a\,|0\rangle + b\,|1\rangle) \otimes (c\,|0\rangle + d\,|1\rangle) = ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle$$

Por tanto, los miembros de la derecha de ambas ecuaciones deben ser iguales. Para ello se debe cumplir que $a \cdot d = 0$ y $b \cdot c = 0$. Sin embargo, si alguno de los coeficientes $a,\ b,\ c,\ d$ es cero, anularía también la amplitud correspondiente al estado $|00\rangle$ o $|11\rangle$ según corresponda, y esto es absurdo.

Este tipo de estados no guardan similitud alguna con el mundo clásico y son los causantes de aportar el crecimiento exponencial respecto al número de partículas en el sistema.

Por último, queda ver qué ocurre con las mediciones en un sistema con múltiples qubits. Supongamos que tenemos un estado general en un sistema de 2 qubits dado por:

$$a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle\,,$$

donde $a,\ b,\ c,\ d$ son números complejos que cumplen la restricción de normalización, es decir, $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$.

Si medimos el primer qubit respecto de la base estándar $\{|0\rangle\,, |1\rangle\}$, se obtendrá $|0\rangle$ con una probabilidad $|a|^2 + |b|^2$ y $|1\rangle$ con probabilidad $|c|^2 + |d|^2$. Además si se obtiene $|0\rangle$ al medir el primer qubit, el estado colapsa al subespacio compatible con la medida, es decir, al generado por los vectores $|00\rangle$ y $|01\rangle$. De esta forma, se proyectaría el sistema al estado $a\,|00\rangle + b\,|01\rangle$ tras realizar una primera medición. Sin embargo, este estado no está normalizado, luego realmente se obtendría el estado:

$$\frac{1}{\sqrt{|a|^2 + |b|^2}}(a\,|00\rangle + b\,|01\rangle)$$

En este nuevo estado tendríamos por lo tanto que al realizar una medida sobre el segundo qubit, obtenemos con probabilidad $|a|^2$ el estado $|0\rangle$, mientras que $|1\rangle$ se obtiene con probabilidad $|b|^2$.

Las mediciones en estados entrelazados son muy particulares ya que, como se muestra a continuación, realizar una medición sobre uno de los qubits influye en el estado de los otros. Dado el estado entrelazado $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ es inmediato ver que es equiprobable obtener el estado $|0\rangle$ o el estado $|1\rangle$ al realizar una medida sobre el primer qubit. Sin embargo, si el

segundo qubit ya ha sido medido previamente y, por ejemplo, se ha obtenido el estado $|0\rangle$, entonces el sistema colapsará al estado $|00\rangle$ y en este nuevo estado se obtiene con probabilidad 1 el estado $|0\rangle$ al medir el primer qubit. Esta propiedad de los estados entrelazados es en la que se basan algunos de los experimentos más relevantes de la computación cuántica, como puede ser los experimentos de *teleportación* o *codificación densa*.

Una vez visto como actúan múltiples qubits dentro de un sistema, a lo largo de la siguiente sección se desarrollará con detalle como aplicar la principal herramienta que permite modificar el estado de los qubits: las puertas cuánticas.

2.4. Puertas cuánticas

En computación clásica, los bits son operados mediante una serie de puertas que son representadas como un circuito cableado. De igual modo tenemos puertas en el mundo cuántico. Dichas puertas tienen que ser isomorfismos lineales unitarios, es decir, si U es la matriz asociada con dicho isomorfismo, se debe verificar que

$$UU^* = I$$

donde U^* es la matriz conjugada traspuesta de U. Este tipo de matrices se llaman matrices unitarias, tienen la propiedad de preservar la norma y, por tanto, la restricción de normalización. Una consecuencia inmediata de esta igualdad es que cualquier puerta cuántica es reversible, cosa que no ocurría con las clásicas. Además, por ser un isomorfismo, el número de qubits de entrada coincidirá con el de salida.

2.4.1. Puerta cuánticas de un qubit

Vamos a proceder a hablar de algunas de las puertas cuánticas más relevantes que cuentan con un solo qubit de entrada y salida. Mostraremos la transformación realizada por cada una de ellas sobre los elementos de la base $\{|0\rangle, |1\rangle\}$, su expresión matricial y cómo se representan gráficamente sobre un circuito. En primer lugar presentamos las denominadas *puertas de Pauli*. La primera de ellas, I, se trata de la identidad que deja un estado cuántico invariable. X es la puerta negación: dado un qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, tenemos que $X(|\psi\rangle) = \beta|0\rangle + \alpha|1\rangle$. Z aplica un cambio de fase (véase la sección 4.3.2) sobre $|1\rangle$ pero no varía las amplitudes. Por ello, las probabilidades de obtener $|0\rangle$ o $|1\rangle$ aplicando una medición quedan igual. Por último, Y es la combinación de X y Z (es decir, $Y = ZX$).

$$I : \begin{array}{ccc} |0\rangle & \longrightarrow & |0\rangle \\ |1\rangle & \longrightarrow & |1\rangle \end{array} \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad -\boxed{I}-$$

$$X: \begin{array}{ccc} |0\rangle & \longrightarrow & |1\rangle \\ |1\rangle & \longrightarrow & |0\rangle \end{array} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad -\boxed{X}-$$

$$Y: \begin{array}{ccc} |0\rangle & \longrightarrow & -|1\rangle \\ |1\rangle & \longrightarrow & |0\rangle \end{array} \quad \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad -\boxed{Y}-$$

$$Z: \begin{array}{ccc} |0\rangle & \longrightarrow & |0\rangle \\ |1\rangle & \longrightarrow & -|1\rangle \end{array} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad -\boxed{Z}-$$

Existen otras puertas que se pueden utilizar para realizar un cambio de fase, de manera similar a como lo hace la puerta Z, que denotamos de manera general por:

$$R_\theta: \begin{array}{ccc} |0\rangle & \longrightarrow & |0\rangle \\ |1\rangle & \longrightarrow & e^{i\theta}|1\rangle \end{array} \quad \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \quad -\boxed{R_\theta}-$$

Nótese que Z es de hecho la puerta R_π. La puerta más relevante que veremos en esta sección es la *Puerta de Hadamard*. Esta puerta es importante puesto que nos permite conseguir un estado en superposición de manera que las amplitudes de $|0\rangle$ y $|1\rangle$ produzcan equiprobabilidad en la medición. La describimos como:

$$H: \begin{array}{ccc} |0\rangle & \longrightarrow & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |1\rangle & \longrightarrow & \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{array} \quad \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \quad -\boxed{H}-$$

Si esta puerta se aplica individualmente a cada uno de los n qubits de un sistema con estado inicial $|0\dots0\rangle$ obtenemos:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \dots \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle$$

Esta última igualdad nos da una idea del poder que puede alcanzar el cómputo cuántico, pues hemos obtenido 2^n estados distintos en superposición. Cuando aplicamos la puerta de *Hadamard* sobre n qubits la denominamos *puerta de Walsh-Hadamard*. Esta puerta se puede definir recursivamente mediante las siguientes dos ecuaciones:

$$W_1 = H, \quad W_{n+1} = H \otimes W_n$$

En la figura 2.2 podemos ver un ejemplo sencillo de la aplicación de la puerta de *Hadamard* sobre los 4 qubits de un sistema cuántico (o, equivalentemente, la aplicación de la

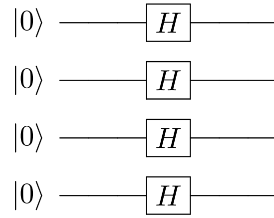


Figura 2.2: Puertas de *Hadamard* aplicadas a un sistema de 4 qubits.

puerta W_4). El estado del sistema tras la aplicación de dichas puertas es

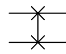
$$|\psi\rangle = \frac{1}{4}\sum_{i=0}^{15} |i\rangle$$

2.4.2. Puertas cuánticas de más de un qubit

La puerta de *Walsh-Hadamard* está definida para más de un qubit. Sin embargo, puede expresarse como el producto tensorial de varias puertas de un solo qubit. No es el caso de las puertas que veremos en esta sección. En primer lugar presentamos la puerta *SWAP*, cuya función es intercambiar los estados de dos qubits. Su representación matricial es

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

mientras que a la hora de representar gráficamente esta puerta en circuitos cuánticos usaremos la siguiente notación:



Un grupo de puertas de dos qubits muy relevante es el de las *puertas controladas*. Su funcionamiento consiste en tomar un primer qubit como control que permanecerá inalterado y un segundo al que se le aplicará una puerta U o no en función del estado del primero. Su representación matricial por bloques viene dada por

$$C(U) = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & U \end{array}\right)$$

La más importante de las puertas que pertenecen a esta familia se suele representar por $C(X)$ y es habitualmente conocida como *CNOT*. Su definición formal es:

$$\text{CNOT:}\begin{array}{ccc}|00\rangle & \longrightarrow & |00\rangle \\ |01\rangle & \longrightarrow & |01\rangle \\ |10\rangle & \longrightarrow & |11\rangle \\ |11\rangle & \longrightarrow & |10\rangle\end{array} \quad \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$$

donde $\bullet$ denota el qubit de control y $\oplus$ el qubit negado (si procede). Como vemos, la puerta X se aplica sobre el segundo qubit si y solo si el estado del primero es $|1\rangle$.

Existen puertas controladas en las que intervienen más de dos qubits como la *CCNOT* o *puerta de Toffoli*. Se trata de una puerta que toma 3 qubits. Utiliza los dos primeros de control, que permanecen inalterados, mientras que al último se le aplica una puerta X siempre y cuando los dos primeros sean $|1\rangle$. Para simplificar, mostramos su matriz por bloques y su representación en circuito:

$$T = \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & 0 & X \end{array}\right)$$

Esta puerta tiene una importancia adicional porque con ella podemos emular las puertas clásicas *NOT* y *AND* que, en particular, constituyen un conjunto funcionalmente completo que permite construir cualquier circuito clásico. Las siguientes dos ecuaciones muestran la definición de dichas puertas.

$$\begin{aligned} T\,|1,1,x\rangle &= |1,1,\neg x\rangle \\ T\,|x,y,0\rangle &= |x,y,x\wedge y\rangle \end{aligned}$$

Nótese que T permite generar cualquier circuito clásico pero no afirmamos tal cosa de los circuitos cuánticos (de hecho, podemos adelantar ya que esta afirmación no es cierta). La generación de estas puertas ayudó a probar a Deutsch que cualquier función clásica puede ser fabricada a partir de una puerta cuántica reversible [Deutsch 1985]. Incluso Bernstein y Vazirani concretaron una *máquina cuántica universal de Turing* [Bernstein y Vazirani 1997].

Según lo que acabamos de ver, dada una función clásica $f\colon \{0,1\}^m \longrightarrow \{0,1\}^k$ existe un *array* de puertas U_f que implementa f en un circuito cuántico. Se verifica que $U_f\,|x,y\rangle = |x, y\oplus f(x)\rangle$ donde x denota un elemento de m bits, y de k bits y $\oplus$ es el operador lógico XOR. Véase que $U_fU_f\,|x,y\rangle = U_f\,|x,y\oplus f(x)\rangle = |x,y\oplus f(x)\oplus f(x)\rangle = |x,y\rangle$. Luego $U_fU_f = I$ y por lo tanto verifica que es unitaria. Para calcular $f(x)$ basta con aplicar $U_f\,|x,0\ldots0\rangle$, cuyo resultado será $|x,f(x)\rangle$. Representaremos gráficamente este circuito cuántico como se muestra en la figura 2.3.

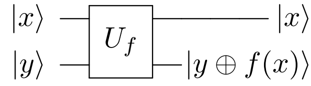


Figura 2.3: Representación en un circuito cuántico de un *array* de puertas cuántico.

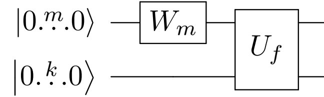


Figura 2.4: Ejemplo del poder del paralelismo cuántico aplicado a un *array* de puertas.

A primera vista, esta situación no parece suponer una gran ventaja con respecto a la computación clásica. Pero, ¿y si pudiéramos obtener todos los valores $f(x)$ para cada $x \in \{0,1\}^m$ con una sola aplicación de U_f? Esto constituye una mejora de complejidad exponencial a constante y lo conseguimos aplicando una puerta de *Hadamard* a cada uno de un total de m qubits (es decir, aplicar W_m), además de preparar otros k qubits en estado $|0\rangle$ tal y como muestra la figura 2.4. Analíticamente, por la linealidad de U_f, tenemos:

$$U_f(W_m(|0.\overset{m}{\cdot\cdot}.0\rangle) \otimes |0.\overset{k}{\cdot\cdot}.0\rangle) = U_f\left(\frac{1}{\sqrt{2^m}}\sum_{i=0}^{2^m-1}|i, 0.\overset{k}{\cdot\cdot}.0\rangle\right) =$$

$$= \frac{1}{\sqrt{2^m}}\sum_{i=0}^{2^m-1} U_f\,|i, 0.\overset{k}{\cdot\cdot}.0\rangle = \frac{1}{\sqrt{2^m}}\sum_{i=0}^{2^m-1}|i, f(i)\rangle$$

No obstante, todo esto no es tan maravilloso como parece. Recordemos que sólo podemos obtener información de un estado cuántico mediante la medición y que tras realizarla, el estado que obtengamos será el mismo al que colapsará el sistema y será único. Por tanto, perdemos este poder de *paralelismo cuántico* que acabábamos de obtener. A pesar de ello, podemos seguir aprovechándonos de este fenómeno pero, antes de medir, debemos aplicar algunas técnicas adicionales alejadas de la computación clásica, como por ejemplo aumentar las amplitudes de los valores de mayor interés. Esta técnica es utilizada en algunos algoritmos como el de Grover.

2.4.3. Teorema de no clonación

De igual forma que tenemos resultados positivos, también existen *teoremas de imposibilidad* de forma similar a lo que ocurre en computación clásica con, por ejemplo, el *problema de parada.*

Teorema 2.1. *Dado un estado cuántico desconocido* $|\phi\rangle$ *es imposible realizar una copia de dicho estado mediante puertas cuánticas unitarias.*

La demostración de este resultado clásico en computación cuántica se basa en el siguiente razonamiento. Supongamos que pudiéramos construir una maquina cuántica capaz de clonar estados cuánticos mediante transformaciones unitarias. Es decir, que existe un operador unitario U (nótese que tanto el producto tensorial como el producto usual de matrices unitarias da lugar a una matriz unitaria que denotamos en este caso por U) tal que para cualquier estado $|\phi\rangle$ se cumple:

$$U(|\phi 0\rangle) = |\phi\phi\rangle$$

Supongamos ahora que tenemos un estado ortogonal a ϕ que denotaremos por ψ. Esto nos permite definir un tercer estado como superposición de los dos anteriores. Dicho estado viene dado por:

$$|\Psi\rangle = \frac{1}{\sqrt{2}}(|\phi\rangle + |\psi\rangle)$$

Si en este momento aplicamos el operador lineal U al estado Ψ entonces se obtiene:

$$U(|\Psi 0\rangle) = \frac{1}{\sqrt{2}}(U\,|\phi 0\rangle + U\,|\psi 0\rangle) = \frac{1}{\sqrt{2}}(|\phi\phi\rangle + |\psi\psi\rangle)$$

Sin embargo, como este operador que estamos definiendo es el operador de clonación, por otro lado también se tiene:

$$U(|\Psi 0\rangle) = |\Psi\Psi\rangle = \frac{1}{2}(|\phi\phi\rangle + |\phi\psi\rangle + |\psi\phi\rangle + |\psi\psi\rangle)$$

Como los lados derechos de las dos ecuaciones anteriores no son iguales se sigue el resultado, dado que hemos llegado a una contradicción. Por ello, podemos concluir que no existe una operación unitaria que pueda clonar un estado cuántico desconocido.

Capítulo 3

Introducción al Testing

Uno de los pilares fundamentales de este trabajo es el testing, más concretamente una de las técnicas más usadas en este área: mutation testing. Por ello, a lo largo del siguiente capítulo daremos unas pinceladas muy básicas sobre en qué consiste el testing y hablaremos más en profundidad de la variante basada en mutaciones.

3.1. ¿Qué es el Testing?

El *testing* es una disciplina de la *Ingeniería del Software* orientada a aumentar la calidad y fiabilidad de un programa. Aunque es difícil dar una definición precisa en pocas líneas (el lector la puede encontrar en [Ammann y Offutt 2016]) podemos decir que su principal objetivo es encontrar errores en el software para, de esta manera, poder corregirlos. Hay que tener muy en cuenta que demostrar el correcto funcionamiento de un programa es, en general, indecidible y por tanto este no es el objetivo del testing. En palabras de Edsger Wybe Dijkstra [Dijkstra 1972], el testing permite encontrar errores pero, sin embargo, no permite demostrar la ausencia de ellos.

A la hora de encontrar un fallo en el software, hay tres términos que hay que tener muy claros para poder describir con precisión la naturaleza del fallo en cuestión. Se introducen los términos en inglés, pues su traducción puede inducir a confusiones no deseadas:

- *Fault* es el termino que se utiliza para indicar un defecto estático en el código del programa que está siendo testeado.

- *Error* se corresponde con un estado incorrecto del programa durante la ejecución del mismo.

- *Failure* se produce cuando el comportamiento externo del programa es incorrecto respecto a los requisitos establecidos para este.

La unidad u objeto fundamental del testing es el *test*. Un test se constituye principalmente de dos componentes:

- *Input*: son los valores que se necesita aportar al programa para que se ejecute la funcionalidad que está siendo testeada.
- *Output*: se corresponde con el resultado que debería producir el programa en caso de que su comportamiento fuera correcto.

Como el software puede constar de múltiples funcionalidades, va a ser necesario contar con numerosos test para poder validarlo adecuadamente. Nos referiremos a ellos como conjunto de test. Por otro lado, en la actualidad los sistemas software suelen tener una cantidad ingente de líneas de código y realizar un análisis exhaustivo es inviable o, incluso, imposible. Pensemos, por ejemplo, en un compilador de Java. Potencialmente, los inputs que puede recibir el compilador no son sólo todos los programas Java, sino cualquier cadena de caracteres. La única limitación viene impuesta por el máximo tamaño de archivo que acepta el *parser* del compilador. Por ello, aunque de forma efectiva se trata de un conjunto finito, al tener un tamaño tan grande es habitual considerar que el número de inputs es *infinito* [Ammann y Offutt 2016]. Siguiendo las ideas presentadas en este trabajo que acabamos de mencionar, la manera de solucionar este problema consiste en introducir unos criterios que permitan al testeador diseñar test eficaces, es decir, test que potencialmente sean más propensos a encontrar *faults* en el código. Estos criterios son:

- *Requisito de Test*: es un *artefacto* concreto del software que se debe cubrir durante el proceso de testing.
- *Requisitos de Cobertura*: son un conjunto de reglas que debe cumplir un conjunto de test para que se cumplan todos los requisitos de test. Existen diversos tipos de cobertura basados en grafos, lógica o dominio de *inputs*.

Una vez que hemos revisado unas nociones básicas sobre testing, podemos mover el foco hacia la rama del testing que realmente concierne a este trabajo. En testing no es solo necesario tener conjuntos de test que nos ayuden a detectar fallos en un programa, sino que los mismos deben ser *eficientes* a la hora de encontrar comportamientos incorrectos. Imaginemos que tenemos un programa con una serie de *faults* en el código. Puede ocurrir que si se diseña un conjunto de test sobre unos requisitos de test pobres, dicho conjunto de test no de lugar a ningún *failure*. Este resultado tras el proceso de testing no se debe a que el programa funcione correctamente, sino a que el conjunto de test no es bueno. *Mutation testing* tiene como uno de sus objetivos evaluar la bondad de un conjunto de test.

3.2. Mutation Testing

Las pruebas de mutación o *mutation testing* fueron propuestas por Richard Lipton en 1971 [Lipton 1971]. Esta técnica entra dentro de lo que se conoce como *testing basado en sintaxis*, y como su propio nombre indica, el principal elemento a tener en cuenta en este tipo de testing es la propia sintaxis del lenguaje.

El objetivo principal de mutation testing es comprobar la eficacia de un conjunto de test para un cierto programa. Para ello, se parte de un código que funciona correctamente y se introducen pequeñas modificaciones en este para posteriormente aplicar los test sobre estos programas modificados y de esta forma comprobar si su comportamiento coincide o no con el del programa original, es decir, si el *fault* inyectado es detectable por alguno de los test que han sido considerados.

Hay varios conceptos clave que nos permiten entender con mayor facilidad mutation testing.

- Un *operador de mutación* es una regla que especifica variaciones sintácticas para una cierta cadena en el lenguaje.
- Un *mutante* es un nuevo programa que se obtiene al aplicar un operador de mutación sobre el programa original.
- Sea m un mutante obtenido al aplicar un operador de mutación a cierto programa p. Decimos que un test t *mata* a m si p y m producen un output distinto al aplicar los inputs dados por t. De la misma forma, dado un conjunto de test $\mathcal{T}$, diremos que $\mathcal{T}$ *mata* a un mutante m si existe $t \in \mathcal{T}$ tal que t mata a m.

Cabe mencionar que esta noción de matar a un mutante se corresponde con el criterio conocido como *strong mutation testing*. Al final de esta sección se verá la diferencia con otros criterios.

Los operadores de mutación simulan errores típicos que puede cometer un programador. El primer paso a la hora de aplicar mutation testing a un programa es definir los operadores de mutación para el lenguaje en el que esté escrito dicho programa. Por ejemplo, supongamos que tenemos el algoritmo que presentamos en la parte superior de la figura 3.1. Un posible error que pueda cometer un programador es confundir los operadores de comparación. Por tanto, esta modificación sintáctica es un candidato ideal para ser considerada como un buen operador de mutación. Si aplicamos este operador al código anterior en la línea 2 se obtiene el mutante que aparece en la parte inferior de la figura 3.1. Es claro que el comportamiento de los dos algoritmos anteriores no es equivalente. Por tanto, un buen conjunto de test debe ser capaz de distinguir entre ambos, es decir, de matar al mutante. Por ejemplo, si ejecutamos el programa original con los valores $(6, 5)$ entonces se devolverá el valor 6. Sin embargo,

```
1: procedure EXAMPLE(n, m)        ▷ Devuelve n si es estrictamente más grande que m.
2:     if n>m then
3:         return n
4:     else
5:         return m
6:     end if
7: end procedure
```

```
1: procedure EXAMPLE(n, m)        ▷ Devuelve n si es estrictamente más grande que m.
2:     if n<m then
3:         return n
4:     else
5:         return m
6:     end if
7: end procedure
```

Figura 3.1: Ejemplo de programa (parte superior) y uno de sus mutantes (parte inferior).

aplicando el mismo par de valores al mutante, el programa mutado devuelve el valor 5. Por lo tanto, el test que tiene como input el par $(6, 5)$ *mata* a dicho mutante.

Pero, ¿qué ocurre si un conjunto de test no mata a un cierto mutante? Esto puede deberse a dos causas:

- El conjunto de test no contiene ningún test que mate al mutante. Por ejemplo, si nuestro conjunto consta únicamente del test $(2, 2)$, entonces obtendremos el mismo valor al aplicarlo al programa original y al mutante.
- No existe test que nos permita distinguir entre el mutante y el programa original en función de sus outputs. En ese caso diremos que se tiene un *mutante equivalente*. Por ejemplo, este sería el caso del mutante generado mediante la sustitución del operador $>$ por el operador $\geq$ sobre el algoritmo mencionado anteriormente.

Nos interesa tener una métrica que nos permita saber como de eficiente es cierto conjunto de test $\mathcal{T}$ a la hora de matar un conjunto de mutantes $\mathcal{M}$ obtenidos a partir de un programa p. Denotemos al subconjunto de mutantes que es matado por $\mathcal{T}$ como $\mathcal{M}^*$. Por tanto, parece sensato definir una métrica de eficiencia en función de la fracción de mutantes que mata nuestro conjunto de test:

$$E(\mathcal{T}, \mathcal{M}, p) = \frac{|\mathcal{M}^*|}{|\mathcal{M}|}$$

donde $|A|$ denota el cardinal del conjunto A. Sin embargo, tal y como se ha mencionado previamente, es imposible matar a un mutante equivalente. Por tanto, para mejorar la precisión de la métrica es necesario excluir el conjunto de mutantes equivalentes $\mathcal{M}_e$. Se obtiene así una nueva métrica más ajustada que la anterior:

$$E^+(\mathcal{T}, \mathcal{M}, p) = \frac{|\mathcal{M}^*|}{|\mathcal{M}\,| \setminus |\mathcal{M}_e|}$$

Si bien esta última noción sería la ideal para evaluar la bondad de un cierto conjuntos de test a la hora de enfrentarse a un conjunto de mutantes, identificar los mutantes equivalentes es un problema indecidible y, por tanto, nos tendremos que conformar con la métrica E. De hecho, evitar la generación de un número elevado de mutantes equivalentes (o, en su defecto, identificar una buena parte de ellos de forma automática) sigue siendo un problema actual en mutation testing [Kintis y col. 2018; Madeyski y col. 2014].

Otro problema importante es el elevado número de mutantes que se generan, ya que incluso para programas pequeños escritos en un lenguaje con pocos operadores de mutación, el número de mutantes obtenidos puede ser inmensamente grande. Una solución que ayuda a mitigar este problema es trabajar sobre lo que se conoce como la *hipótesis del programador competente* [DeMillo, Lipton y Sayward 1978]. Esta hipótesis tiene dos puntos clave:

- La mayor parte de los errores en el software cometidos por un programador senior son debidos a pequeños errores sintácticos.
- Un programador avanzado no suele cometer más de una vez este tipo de error. Por lo tanto, no se aplicará más de un operador de mutación a la hora de generar un mutante.

Otro de las hipótesis sobre las que se suele trabajar en mutation testing es el conocido *efecto de acoplamiento*. Esta hipótesis propone que los errores más graves en un programa suelen estar *acoplados* con errores sencillos y, por tanto, si un test es capaz de desvelar estos errores sencillos, entonces también desvelará los graves [Offutt 1992].

Por último, tal y como se ha mencionado previamente, la forma de matar un mutante que hemos mencionado consiste en comparar los outputs del programa original y el mutante. Esto se conoce como *strong mutation testing*. Sin embargo, no es el único criterio que se puede considerar a la hora de decidir si un mutante muere o no. Sea p un programa, t un test y m un mutante obtenido a partir de la aplicación de un operador de mutación sobre p. Podemos decir que t mata a m en función de los siguientes dos criterios:

- *Strong Mutation Testing*: t mata a m si los outputs obtenidos por p y m al aplicar los inputs de t son distintos.
- *Weak Mutation Testing*: t mata a m si, usando los inputs dados por t, tras ejecutar la

sentencia mutada para m y la sentencia original para p el estado interno del programa es distinto.

Véase que este último criterio, al ser menos restrictivo, ayuda a matar más mutantes. Esto se debe a que un mutante m y un programa p pueden tener estados internos distintos durante la ejecución del programa y, sin embargo, producir el mismo output.

Capítulo 4

Mutation Testing aplicado a la computación cuántica

Una vez vistos los principales conceptos e ideas detrás de mutation testing, podemos tratar de abordar el problema de aplicar esta técnica a software cuántico. En este capítulo presentaremos un marco general para mostrar cómo se podrían adaptar las nociones de mutation testing que se utilizan en el mundo de la programación clásica al mundo cuántico para, posteriormente, centrar el desarrollo del capítulo en torno a los dos lenguajes de programación cuánticos que consideramos en este trabajo: *Qiskit* y `Q#`.

4.1. Diseño de operadores de mutación

La tarea más crítica que conforma el diseño de un marco de mutation testing para un lenguaje concreto, sea este cuántico o no, consiste en diseñar los operadores de mutación. Tal y como se vió en el capítulo anterior, los operadores de mutación debe modelar errores comúnmente cometidos por los programadores. Aquí se encuentra la primera dificultad con la que nos encontramos a la hora de diseñar una serie de operadores en el entorno cuántico.

Los lenguajes de programación cuánticos se han empezado a desarrollar durante los últimos años y, aunque han ido ganado notoriedad con el paso del tiempo, todavía no se están realizando grandes proyectos con este tipo de lenguajes. Esto puede deberse a que el desarrollo de sistemas cuánticos está evolucionando a un paso lento y el mayor computador cuántico del que se tiene conocimiento público a día de hoy, desarrollado por IBM, consta de tan sólo 53 qubits. Por tanto, pese a que un desarrollador de software puede contar a nivel de simulador con el número de qubits que considere necesario, es posible que no pueda ejecutar dicho código en una máquina física y disfrutar de la capacidad de cómputo que ofrecen estos

sistemas. Si hacemos una analogía con la computación clásica, la tarea de identificar los errores más comunes cometidos por un programador es relativamente sencilla para lenguajes clásicos como *Java*, C++ o *Python*, dado que existen infinidad de repositorios de código donde poder realizar un estudio sobre qué métodos u operadores son más utilizados. Además, se cuenta con múltiples plataformas de preguntas y respuestas donde poder inspeccionar cuales son los errores más habitualmente cometidos para cada lenguaje. Sin embargo, esto no ocurre para lenguajes de computación cuántica. Si bien cada vez aparecen más repositorios con código cuántico, así como plataformas donde los programadores pueden exponer sus problemas durante el desarrollo, todavía no tienen la cantidad de información suficiente para permitir un estudio a media escala sobre el código cuántico que permita identificar aquellos errores que son lo más adecuados para ser modelados como operadores de mutación. Más adelante se detalla como se decide abordar este problema para los lenguajes *Qiskit* y Q#.

Este no es el único problema que tiene la aplicación de mutation testing en código cuántico. El otro gran inconveniente es la especificación de los test, más en concreto la especificación de las entradas.

4.2. Especificación de inputs cuánticos

Una vez que se cuenta con el diseño de los operadores de mutación, la obtención de mutantes para un programa cuántico no difiere del proceso análogo para lenguajes de programación clásicos. Dicho proceso consiste tan sólo en buscar y reemplazar una cadena del código siguiendo las reglas concretas del operador de mutación que se desea aplicar.

Tras obtener el conjunto de mutantes, el siguiente paso a realizar es aplicar el conjunto de tests a dichos mutantes. Recordemos que un test se compone de unos valores de entrada, inputs, y de los valores de salida esperados, outputs. Aunque a simple vista pueda parecer que este proceso es sencillo, cuenta con una serie de dificultades. Una de ellas es obtener los valores que provoquen que se alcance el estado del programa que se quiere testear. Estos valores son conocidos como *valores de prefijo*. De la misma forma, los valores que debe recibir el programa tras alcanzar el estado que se desea testear y que nos permiten terminar la ejecución del programa o ver los resultados, conocidos como *valores de postfijo*, también presentan una complicación a la hora de diseñar los tests. Obtener estos valores de prefijo y postfijo suponen una tarea extra para el encargado de desarrollar el conjunto de tests y es necesario llevarla a cabo tanto en programación clásica como en programación cuántica. Sin embargo, con este último tipo de programación surge un nuevo problema que no se tiene con los lenguajes de programación clásicos.

En algunos de los lenguajes de programación cuánticos, como es el caso de *Qiskit* y Q#, se tienen tipos primitivos pertenecientes al paradigma de la programación clásica como enteros

```
using (register = Qubit[2]) {

  //LLamar al metodo y guardar el output
  let(output) =  method(register);

  ResetAll(register);

  return output;
}
```

```
using (register = Qubit[2]) {

  //Inicializar los qubits al estado deseado
  X(register[0]);
  H(register[1]);

  //LLamar al metodo y guardar el output
  let(output) =  method(register);

  ResetAll(register);

  return output;
}
```

Figura 4.1: Llamada a method con inicialización de qubits al estado deseado.

o booleanos. Estos tipos no suponen ningún problema si son necesarios como entradas de un test, pues son fácilmente declarables independientemente del lenguaje. Sin embargo, con la computación cuántica aparece un nuevo tipo: el qubit. Generalmente, cuando se declara un qubit, comienza inicializado al valor $|0\rangle$. Si el encargado de diseñar los test desea comprobar el funcionamiento de cierto método con un qubit que se encuentre en un estado diferente al $|0\rangle$, debe ser él el encargado de transformar el qubit al estado deseado mediante la aplicación de puertas cuánticas. Por tanto, el testeador necesita diseñar para cada test un circuito cuántico que se encargue de llevar al estado deseado cada qubit que se vaya a utilizar como entrada de un test. Esto supone añadir un grado de complejidad a la obtención de valores de prefijo.

Para ver con más claridad el problema que acabamos de presentar, veamos un ejemplo. Supongamos que tenemos el código, en el lenguaje `Q#`, que se presenta en la parte superior de la figura 4.1. Adicionalmente, supongamos que se quiere realizar un test donde los 2 qubits del registro estén inicializados a los valores $|1\rangle$ y $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, respectivamente. Entonces se debe añadir código al código anterior para representar estas inicializaciones. El resultado de esta inclusión aparece en la parte inferior de la figura 4.1. Por tanto, el testeador debe

ser capaz de crear la secuencia de puertas que transforme cada qubit que se vaya a utilizar como input del test al estado cuántico deseado y esta secuencia se debe incluir dentro del código que se quiere testear.

Una vez que ya sabemos cómo se pueden generar los mutantes y cómo se pueden diseñar los tests, sólo falta decidir cuando hemos matado a un mutante. De nuevo aparece una dificultad añadida para esta tarea cuando trabajamos en el mundo cuántico.

4.3. Decisión de la muerte de un mutante

Tal y como se vio en el capítulo anterior, la manera más habitual de decidir si un mutante muere o no es comparando los outputs de la ejecución del mutante con los del programa original. Si intentamos llevar esta idea hacia la programación cuántica aparece de inmediato un inconveniente.

Los programas cuánticos son, en su mayor parte, probabilistas. Esto se debe a que están basados en el uso de qubits y, como ya se ha visto previamente, una medición de un qubit arroja dos posibles resultados con cierta probabilidad. Por lo tanto, en el momento en que se realice una medición, automáticamente el programa pasará a ser no determinista (dando por hecho que el resultado obtenido en la medición es usado posteriormente). Para ilustrar este problema, véase el código de la figura 4.2.

En este sencillo ejemplo simplemente transformamos un qubit que se encuentra en el estado $|0\rangle$ al estado $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ mediante una puerta *Hadamard*. En dicho estado, el qubit tiene las mismas probabilidades de arrojar como resultado $|0\rangle$ o $|1\rangle$ al realizar una medición sobre él. La variable de salida *output* depende del resultado de esta medición. Es obvio que distintas ejecuciones de este programa darán resultados diferentes. Este no-determinismo supone un inconveniente para realizar testing, pues un mismo test puede ser correcto para algunas ejecuciones de un programa y erróneo para otras. En particular, para mutation testing, no va a ser suficiente con ejecutar una vez el programa original y el mutante y comparar las salidas. En el marco que presentamos en esta memoria, se han tenido en cuenta dos alternativas que solventan este problema: realizar múltiples ejecuciones del programa o aprovechar las ventajas de trabajar con un simulador. Vamos a ver con detalle estas dos soluciones.

4.3.1. Valoración estadística de los outputs

Una primera manera de abordar este problema consiste en realizar un estudio estadístico de los resultados. Para ello, se deberá ejecutar múltiples veces el programa original y, de igual manera, el mutante. El número de veces que se ejecutan estos programas es decidido

```
using (register = Qubit[1]) {

  mutable output = 0;

  H(register[0]);
  let res = M(q0);

  if(res = One) {
    output = 1:
  }

  ResetAll(register);

  return output;
}
```

Figura 4.2: Ejemplo de superposición de estados.

por el encargado de testear, sabiendo que a medida que se aumenta el número de ejecuciones aumenta la fiabilidad del resultado.

También es tarea del testeador decidir como comparar las salidas. Supongamos que un programa da como salida una tupla. Podemos entender dicho output como un *todo* y por tanto un mutante moriría si la tupla de salida no coincide íntegramente con la tupla retornada por el programa original. Pero también podemos entender dicha tupla como valores independientes. En ese caso, el el testeador debe distinguir cuantos elementos de las tuplas devueltas por el mutante y el programa original deben coincidir para matar o no a dicho mutante. Una vez que se decide como comparar las salidas y se ejecutan los programas correspondientes numerosas veces, hay que decidir una métrica estadística que nos permita dictaminar que mutantes mueren con un cierto valor de confianza. La métrica decidida en este trabajo se detalla en el siguiente capítulo.

4.3.2. Estado interno del simulador

Una segunda forma que nos permite solucionar el inconveniente del no determinismo en la computación cuántica es aprovechar el uso del simulador. Hoy en día, el acceso a un máquina cuántica es realmente limitado. Por ello, la mayor parte de los lenguajes de programación cuánticos han sido diseñados para ser ejecutados en una máquina tradicional. Esto se traduce en que los programas cuánticos corren sobre un simulador y, por lo tanto, los estados de los qubits están siendo simulados y almacenados en bits corrientes. Es por ello que, al menos en el caso de *Qiskit* y Q#, se pone a disposición del programador herramientas que permitan interaccionar con el simulador. Una de estas herramientas posibilita mirar las

amplitudes de los qubits en el simulador sin alterar el estado de estos. Esto en el mundo cuántico real no es posible, pues al observar el estado de un qubit, como ya bien sabemos, colapsa. Sin embargo, en simulación podemos hacer uso de esta herramienta para comparar las salidas de nuestros test de una manera muy eficiente. Decimos que un mutante muere si el estado interno de los qubits tras la ejecución es diferente al estado interno del programa original. Nos gustaría recalcar que somos conscientes de que esta aproximación se puede ver, legítimamente, como *hacer trampas*. Sin embargo, consideramos que es adecuada para analizar programas cuánticos a la hora de detectar *faults* en los mismos, incluso teniendo en cuenta que, finalmente, se ejecutarán en un ordenador cuántico donde no será posible hacer esta trampa.

Como no podría ser de otra forma en el mundo cuántico, hay una complicación adicional. Si nos limitamos a comprobar la igualdad entre dos qubits estaremos sopbrepasando el poder de distinción real que tenemos en el mundo cuántico. Por ello, debemos plantearnos la pregunta: ¿Cuando dos qubits son iguales?

Por un lado, en mecánica cuántica no se tienen en cuenta los cambios de *fase globales* y consideramos que dos qubits son iguales módulo un cambio de fase, es decir, $|\phi\rangle = e^{i\theta}|\phi\rangle$. Esto se debe a que el operador de proyección es el mismo. Por tanto, las probabilidades que arrojan a la hora de medir son las mismas.

Además, los qubits que difieren por un cambio de *fase relativo* de la forma: $|\phi\rangle = \alpha|0\rangle + e^{i\theta}\beta|1\rangle$ tienen las mismas probabilidades de medir $|0\rangle$ o $|1\rangle$ independientemente de la fase θ elegida (recordemos que $e^{i\theta}$ tiene modulo unidad).

Por lo tanto, a nivel de simulación, debemos considerar las probabilidades de medir $|0\rangle$ o $|1\rangle$ como factor para determinar si dos qubits son o no iguales.

4.4. Mutation Testing en *Qiskit* y Q#

En este capítulo hemos analizado los principales inconvenientes que hemos encontrado a la hora de trasladar las ideas del mutation testing al mundo cuántico. Veamos ahora como se han desarrollado estos conceptos para los dos lenguajes que se consideran en este trabajo.

En esta última sección del capítulo vamos a centrarnos principalmente en el proceso llevado a cabo para el diseño de los operadores de mutación. Tanto la especificación de los inputs para los test, como la resolución de las muertes de los mutantes se detallarán en el siguiente capítulo, pues incide de manera más directa en el diseño del sistema desarrollado.

La primera decisión importante que se tuvo que realizar es decir qué elementos del lenguaje eran candidatos a ser mutados. Tanto *Qiskit* como Q# son lenguajes que tienen una serie de métodos cuánticos, pero también están dotados de elementos clásicos como pueden

ser bucles, sentencias condicionales o tipos primitivos básicos. Como mutation testing ya ha sido aplicado a este último tipo de elementos, se decidió orientar el trabajo al mundo cuántico y, por tanto, sólo se han tenido en cuenta como candidatos a operadores de mutación aquellos elementos puramente cuánticos.

Para el lenguaje *Qiskit* se ha tenido en cuenta principalmente el intercambio de puertas cuánticas. De hecho, se consideró que era altamente probable que un programador confundiese dos puertas cuánticas y cometiese este tipo de error.

En el caso de Q# también se consideró este tipo de operador de mutación pero, además, se añadió un nuevo tipo de operador. En este lenguaje hay una serie de constantes que son de naturaleza puramente cuántica. Por un lado, tenemos los resultados al realizar una medición, que vienen dados por las constantes *Zero* y *One* y, por otro lado, las constantes relativas a los operadores de Pauli: *PauliX*, *PauliY* y *PauliZ*. Estas constantes se utilizan a la hora de especificar sobre qué eje se desea medir o a la hora de aplicar una rotación, entre otros. Se ha considerado como operador de mutación el intercambio de *Zero* por *One* y viceversa, así como cualquier intercambio dentro de los operadores de Pauli.

Por último, a muchos de los métodos de Q# se les puede aplicar el functor *Adjoint*, que proporciona la versión adjunta del método. Se ha considerado también como mutación que al programador se le olvide aplicar dicho functor.

Capítulo 5

Gestión, planificación y desarrollo del proyecto de software

En los capítulos anteriores hemos presentado las bases del testing, de la computación cuántica, de mutation testing y de los principales problemas que aparecen a la hora de adaptar este proceso al mundo cuántico. En este capítulo presentamos el sistema *software* desarrollado donde se implementan todas estas ideas. En particular, hablaremos de gestión, planificación y desarrollo de funcionalidades. La estructura y muchos de los conceptos tratados en este capítulo se sustentan en los conocimientos adquiridos en distintas asignaturas de la carrera, especialmente en *Ingeniería del Software*.

5.1. Gestión del proyecto

En primer lugar, vamos a presentar todo el proceso de gestión del proyecto, desde la organización como equipo, hasta herramientas y *software* utilizado para desarrollar el código y la memoria.

5.1.1. Gestión de equipo

La gestión del equipo es sencilla. Los dos miembros del grupo contamos con el mismo poder para la toma de decisiones y, por tanto, dichas decisiones han de tomarse de manera consensuada. Si bien es cierto que, según la actividad, alguno de los miembros puede estar más involucrado en ella y es razonable que sus argumentos tenga un mayor peso.

Además, en caso de decisiones clave siempre teníamos una segunda opinión: la correspondiente a nuestros tutores. Por tanto, podemos decir que ellos han formado parte también de

la toma de decisiones. Hemos tratado de involucrarnos por igual en todas las fases y tareas del TFG con, tal vez, algunas excepciones. Veremos a continuación de forma detallada la contribución de cada uno de los miembros al proyecto.

5.1.2. Contribución al proyecto de Luis Aguirre

La idea de realizar este proyecto comienza a finales de junio de 2019. Nos pusimos en contacto con los directores de este trabajo y se realiza una primera reunión. Hay que mencionar que las dos disciplinas principales sobre las que se ha construido este trabajo, mutation testing y computación cuántica, son áreas sobre las que no habíamos visto ni siquiera una pequeña introducción en alguna de las asignaturas de la carrera de Informática o Matemáticas cursadas hasta la fecha.

Es por ello que a raíz de esta reunión inicial surge una primera tarea a realizar: investigar y adquirir conocimiento sobre mutation testing y computación cuántica. Para ello utilicé como literatura, principalmente, *Introducción a la computación cuántica para no-físicos* [Rieffel y Polak 2000]. De este texto obtuve las primeras nociones importantes sobre computación cuántica que se han ido mencionando a lo largo del texto: qubit, superposición de estados, entrelazamiento, paralelismo y un largo etcétera. Además, también se realizó una lectura exhaustiva de un recurso online [Matuschak y Nielsen 2019] desarrollado por uno de los promotores de la computación cuántica más relevantes en la actualidad, Michael A.Nielsen, con la colaboración de Andy Matuschak. Este recurso me pareció adecuado pues la interfaz y el método de lectura están construidas sobre potentes ideas del campo de la ciencia cognitiva y su principal objetivo es hacer que el lector recuerde los nuevos conceptos aprendidos, que en el caso de la computación cuántica suelen tratarse de conceptos y notación que no es familiar.

A su vez, para aprender las primeras ideas del mutation testing se realizó una lectura del artículo *Mutation Testing* [Hierons, Merayo y Núñez 2010], si bien más adelante, durante el curso 2019/2020, decidí cursar la asignatura *Testing de Software* con la idea de ampliar mi conocimiento sobre testing y de esta forma poder aplicarlo a este proyecto.

Una vez que se tenían los conceptos principales sobre computación cuántica y mutation testing, se realiza una nueva reunión donde se decide en qué lenguaje cuántico se especializará cada alumno. En mi caso decidí estudiar el lenguaje cuántico de Microsoft `Q#`. Esta tarea implicó aprender a instalar el entorno de desarrollo y las distintas alternativas de las que se disponía, familiarizarse con la sintaxis del lenguaje, así como con el sistema de tipado, o examinar ejemplos de código desarrollado por Microsoft. Esta última tarea fue especialmente importante pues me permitió reconocer qué operadores eran más utilizados, siempre con la idea en mente de poder definir un operador de mutación a partir de ellos.

Todo este trabajo es previo a empezar a desarrollar el sistema y ha sido, posiblemente, la etapa que más tiempo ha consumido. Esto se debe al hecho de haber realizado un trabajo sobre un campo que era prácticamente desconocido para mi, y por ello decidí que era imprescindible dedicar una importante porción del trabajo a la formación y estudio de este área.

Tras haber realizado el estudio previo, tarea que se alarga hasta principio de 2020, se comienza con el desarrollo del sistema. Este proceso se extenderá a lo largo de tres meses, aproximadamente. La realidad es que ambos componentes del grupo nos hemos involucrado de manera similar en el diseño y creación del programa.

Acostumbrados ya a una dinámica previa de trabajo conjunto, se fueron designando pequeñas tareas que debían ser llevadas a cabo. Este ha sido el proceso que se ha seguido durante todo el desarrollo del sistema y, por tanto, ninguna de las componentes principales del programa ha sido desarrollada única y exclusivamente por alguno de los miembros del grupo. Sin embargo, la responsabilidad de desarrollar alguna de estas pequeñas tareas mencionadas previamente ha podido recaer en uno de los miembros del grupo, habiendo acordado previamente como debía ser desarrollado, y haciendo una comprobación posterior por parte del otro integrante. Algunas de las tareas, entre otras, que han recaído en mi persona han sido el desarrollo de la lógica encargada del análisis sintáctico del código y la aplicación de los operadores de mutación, el *parsing* del código para la obtención de la declaración de todos los métodos involucrados en el programa sobre los que se quieren aplicar las mutaciones y una parte relevante de la documentación del código.

Además, como ya se ha comentado previamente, cada uno de los integrantes del grupo se especializó en un lenguaje de programación concreto, lo que se tradujo en que ciertas clases del sistema fueran desarrolladas de manera exclusiva por cada miembro del grupo. Estas clases son las relacionadas de manera directa con cada lenguaje de programación. En particular, encontramos las clases usadas para definir cada operador de mutación. Además, en el caso de Q#, tuve que definir ciertas expresiones regulares que me permitían analizar el código en busca de cadenas de caracteres de una manera más precisa. También recayó sobre mi la especificación de la entrada por parte del usuario para el lenguaje Q#, así como la generación automática del script de Python encargado de llamar a las subrutinas de Q# a las que se deseaba aplicar las mutaciones.

Por último, en lo relativo a la elaboración de esta memoria, sí se ha realizado un reparto de tareas más amplio. Esto se debe a que, tras realizar un reunión previa donde se decidió la estructura de la memoria, se dio con una distribución de capítulos que eran prácticamente independientes unos de otros. La división del trabajo individual por capítulos nos iba a permitir trabajar de manera más rápida.

El capítulo dedicado a la introducción del testing y mutation testing se decidió que fuese mayormente desarrollado por mí, ya que contaba con un poco más de experiencia en dicho ámbito tras haber cursado la ya mencionada asignatura *Testing de Software* a lo largo de este curso. Además, al manejar con mayor soltura el idioma inglés que mi compañero, he sido el encargado de traducir los capítulos que debían ser añadidos a esta memoria también en inglés.

Así, si bien ciertas tareas se han realizado de manera individual a lo largo de este proyecto, siempre ha habido previamente una puesta en común de ideas, así como una comprobación posterior por parte de ambos integrantes para de esta forma asegurar que, tanto Javier como yo, quedáramos satisfechos, bajo nuestros estándares individuales, del trabajo realizado.

5.1.3. Contribución al proyecto de Javier Pellejero

En primer lugar, y como detallaremos en las secciones venideras, cabe destacar la dedicación empleada a adquirir los conocimientos necesarios para poder realizar este TFG. A la hora de hablar de temas como planificación o el estudio de los temas tratados, es difícil separar el trabajo realizado para los TFG de las facultades de Matemáticas e Informática.

Así, nuestra base sobre mutation testing y computación cuántica eran mínimas. Cabe destacar que esta memoria puede dar una idea equivocada de los conocimientos adquiridos durante el periodo de aprendizaje para llevar a cabo este proyecto dado que se omiten muchas cuestiones profundas, relativas al mundo de la mecánica cuántica y sus pilares matemáticos, que sí he creído convenientes incluir en el TFG del grado de Matemáticas. De este modo, quiero destacar que la preparación para poder realizar ambos trabajos ha sido laboriosa, ha ocupado una parte muy significativa del tiempo total empleado y que no siempre se manifiesta este esfuerzo fácilmente en las memorias realizadas.

La primera toma de contacto con el mundo de la computación cuántica fue mediante *An Introduction to Quantum Computing for Non-Physicists* [Rieffel y Polak 2000] recomendado por nuestros tutores. Desde mi punto de vista es un contenido muy completo, con multitud de referencias y ejemplos y útil para adentrarse en los sistemas de información cuánticos. Sin embargo, su título refleja la ausencia de contenido referente a las bases en las que se sustenta todo este marco teórico y si se quiere profundizar sobre ellas se debe acudir a otras fuentes más completas.

La solución a este problema tiene por nombre *Quantum Computation and Quantum Information* [Nielsen y Chuang 2001]. Ha sido mi principal referencia a la hora de ahondar en la experiencia cuántica. Dedica muchas páginas a construir desde cero todo el marco matemático que sustenta este tipo de computación, desde nociones tan básicas como espacios vectoriales a otras algo más elaboradas como ciertas propiedades de los espacios de *Hilbert*.

Aún así, aunque también dedica numerosas páginas a entender los entresijos de los postulados cuánticos, no es todo lo completo que debería ser.

En cuanto a mutation testing, el artículo de nuestros tutores con otro colaborador [Hierons, Merayo y Núñez 2010] fue nuestra iniciación en esta modalidad de testing. La adaptación cuántica fue un proceso que se desarrolló principalmente bajo las indicaciones de nuestros tutores, alguna idea esbozada en un trabajo reciente [Polo Usaola 2020] y la intervención de nuestras propias ideas.

Mientras adquiría estos conocimientos me iba familiarizando con el lenguaje en el que profundizo en el TFG de Matemáticas; *Qiskit* de IBM. Aunque su documentación, en general, es extensa, existen algunas herramientas y funcionalidades del lenguaje que carecen de dicha documentación y debe ser ampliada en el futuro. En cuanto a la sintaxis, es fácil de aprender y legible. Además, pese a que el hecho de pensar en puertas cuánticas podría convertirlo en un lenguaje de nivel inferior a uno ensamblador clásico, el hecho de estar combinado con un lenguaje de alto nivel multiparadigma como *Python* abre un amplio abanico de posibilidades, muchas de ellas aún por explotar.

El número de ejemplos de código en *Qiskit* presente en Internet no es abundante, pero permite hacerse a la idea de qué instrucciones son las más usadas y, por tanto, las más propensas a introducir errores por parte del programador.

En cuanto al desarrollo del sistema, la involucración de ambos miembros del grupo trató de ser lo más pareja posible. Sin embargo, debido a mi conocimiento sobre *Qiskit*, me encargué de elaborar todos sus operadores de mutación, la estructura de entrada de los datos de ejemplo que será comentada más adelante en este capítulo y todo el proceso referente a las llamadas a *Python* que ejecutan mutation testing sobre *Qiskit*.

Además, tuve cierto peso adicional a la hora de establecer la estructura del código *Java* constituida por todas las clases, paquetes o patrones de diseño que acaban marcando las funcionalidades del programa. También estuve centrado en la estructura de la mayoría de las vistas y las herramientas visuales creadas para ellas y el desarrollo de ciertos paquetes de la lógica, pero siempre con la supervisión de mi compañero Luis.

Podemos asegurar que no hay parte del código en la que no hayamos estado ambos involucrados y, por tanto, que desconozcamos su funcionamiento por mucho que dichas líneas estuvieran escritas por el otro miembro del grupo.

En cuanto a esta memoria, se ha planificado de forma más especializada. A nivel de documento no solamente es complicado, sino que carece de sentido estar trabajando sobre las mismas líneas del archivo. Pese a ello, el capítulo 2 referente a la introducción a la computación cuántica y los dos últimos, ejemplos y conclusiones, han sido elaborados de

manera conjunta, dividiendo el trabajo por secciones y subsecciones.

En mi caso, pero siempre de manera planificada y supervisada por mi compañero, me he encargado del capítulo en el que nos encontramos debido a mi ligero mayor peso y toma de decisiones en la estructura del sistema que hemos desarrollado. Como veremos a continuación, explico nuestra planificación, modelo de proceso y todos los detalles referentes a las funcionalidades del programa, la estructura del código, que partes del mismo influyen en cada uso del programa, etcétera.

En conclusión, quiero destacar el esfuerzo de ambos componentes del equipo para desarrollar este proyecto, el cual espero que tenga continuidad. Creo que las herramientas que hemos proporcionado pueden ser útiles para algunos investigadores que están tratando de dar cada vez más importancia al testing en la computación cuántica y pueden ser mejoradas y desarrolladas por otros colaboradores y estudiantes en futuros proyectos.

5.1.4. Gestión de configuración

En este apartado de la memoria describiremos las herramientas y elementos de *software* utilizados en el proyecto. Empezando por esta memoria, al estar realizada en LaTeX, hemos necesitado programas para su edición y compilación. Ambos miembros del grupo hemos utilizado como distribución *MiKTeX*, mientras que como editor hemos usado *Texmaker*.

En cuanto a nuestro sistema, el grueso del programa está realizado en *Java* y se ha utilizado como gestor y editor del mismo la plataforma *Eclipse*, usando como herramienta de desarrollo la versión 8 de *Java SE Development Kit* (JDK) de *Oracle*.

El programa principal debe ejecutar una serie de test sobre los lenguajes de computación cuántica `Q#` y *Qiskit*. En el caso del primero, permite ser llamado desde `C#` y Python, siendo más común utilizar el primero. En el caso del segundo, más que un lenguaje en sí mismo, es un marco de trabajo que engloba varias librerías que se ejecutan sobre *Python*. Se opta por dejar `C#` de lado, puesto que *Python* es el lenguaje en común de ambos y que nuestro programa principal, mediante una llamada al sistema, ejecute un programa en dicho lenguaje. Dicho programa *Python*, que nos sirve para enlazar con *Qiskit* y `Q#`, cambia en cada testing realizado y es el programa *Java* principal el que debe reescribirlo en tiempo de ejecución para, a continuación, llamarlo. Además, se han de escribir archivos adicionales en *Python* que contienen funciones útiles y siempre necesarias. Para ello se emplea un programa de edición sencillo como *Notepad++*, además de *Jupyter Notebook*, usando la distribución *Anaconda*, para analizar que tanto las funciones escritas por nosotros como las generadas por el programa funcionan correctamente.

Para el correcto funcionamiento del programa en su conjunto se necesitan una serie de requisitos.

- Una máquina virtual capaz de ejecutar *Java* como *Java Runtime Environment* (JRE).
- *Python* 2 o 3. (Se recomienda *Python* 3).
- La librería de *Python func-timeout* de Tim Savannah bajo licencia LGPLv3 accesible en `https://github.com/kata198/func_timeout/blob/master/LICENSE`. Se adjunta en el repositorio del proyecto.
- `Q#` (sólo si se ejecutaran test con este lenguaje cuántico).
- *Qiskit* (sólo si se ejecutaran test con este lenguaje cuántico).

En cuanto a la organización y distribución de nuestro código, hemos elegido *Github*. Además de ser un excelente gestor de versiones, tiene el programa (para el sistema operativo *Windows*) *Github Desktop*, una interfaz sencilla para subir y gestionar el código. En el repositorio, que se puede encontrar en `https://github.com/javpelle/TFGInformatica`, existen dos carpetas principales en el directorio raíz:

- `tex`: que almacena el código LaTeX.
- `src`: que almacena el código fuente, tanto *Java* como *Python*

Además, existen otras carpetas de menor relevancia, con contenidos como ejemplos en los dos lenguajes cuánticos tratados o presentaciones. Finalmente, en la raíz principal se encuentra la *licencia MIT*.

Como veremos, nuestro sistema no sólo está destinado a usuarios que estén interesados en realizar mutation testing sobre programas cuánticos, sino que pueden integrarse una serie de funcionalidades adicionales en el mismo. Por ello, hemos optado por esta licencia, que posiblemente sea la de menor número de restricciones. Cualquier usuario interesado puede tomar el proyecto y modificarlo, incluso para uso comercial.

5.2. Planificación

A la hora de considerar la planificación del proyecto, se pueden tener en cuenta dos aspectos ortogonales: desde el punto de vista *temporal*, que incluye el proceso de investigación, desarrollo del sistema y realización de la memoria y otro desde el punto de vista del *modelo de proceso* elegido para el desarrollo del sistema.

5.2.1. Planificación temporal

Para planificar este TFG es importante entender que, como alumnos del doble grado en Ingeniería Informática y Matemáticas, debemos realizar un TFG por grado. Nuestra idea

inicial es que ambos trabajos estuvieran relacionados y ya en julio de 2019 contactamos con nuestros tutores y conocíamos el tema de los mismos.

Dejamos constancia de que en el caso del TFG de Matemáticas, dichos trabajos eran individuales aunque ambos alumnos tratamos un tema común que es el de introducirnos en la computación cuántica con una sólida base matemática y con alguna pincelada de conocimiento en mecánica cuántica. Además, cada uno introduce un lenguaje de programación cuántico: Q# en el caso de Luis Aguirre y *Qiskit* en el caso de Javier Pellejero. Por ello es fácil deducir que este trabajo tiene sus cimientos en los dos realizados por cada miembro del grupo para el grado de Matemáticas. Explicado esto, empezaremos por enumerar una serie de fases de planificación que incluye todos los trabajos.

1. *Proceso de investigación.* Puesto que se trata de una serie de conocimientos totalmente nuevos para nosotros, y con una complejidad considerable, es importante dotar a esta fase de una duración prolongada. Decidimos establecer como límite para esta fase finales de enero, coincidiendo con el fin de exámenes del primer cuatrimestre. En el segundo cuatrimestre, ninguno de los integrantes del grupo cursará asignaturas de grado, así que debería haber tiempo suficiente para desarrollar el resto de fases.

2. *Desarrollo del sistema.* Una vez familiarizados con la teoría de la computación cuántica y con el proceso de mutation testing, estamos en condiciones de desarrollar nuestro sistema. En las siguientes páginas se darán más detalles del mismo. En cuanto al tiempo, estimamos unos dos meses para realizarlo.

3. *Memorias individuales del TFG del grado de Matemáticas.* Por lo comentado anteriormente, es consecuente realizar primero estas memorias pues su contenido sirve de base para desarrollar esta misma. Se establece que un mes es apropiado para que cada miembro del equipo realice su memoria. Cabe destacar que siguiendo las estrictas indicaciones de nuestro tutor (en el caso de los TFGs de Matemáticas, tenemos como único tutor a Manuel Núñez), las memorias se realizarían de forma completamente independiente y sin interacciones para hablar de su estructura o contenidos específicos.

4. *Memoria del TFG del grado en Ingeniería Informática.* Esta es la siguiente tarea a efectuar. El marco temporal previsto para completar esta tarea es el penúltimo mes del proyecto.

5. *Revisión.* Se establece el mes de junio para revisar las memorias (en particular, por parte de nuestros tutores), validar y perfeccionar el sistema y rematar cualquier otra tarea.

Aclaramos que estas fechas son orientativas pues se trata, ciertamente, de un proyecto de planificación. Como veremos a continuación, el modelo seguido para desarrollar nuestro

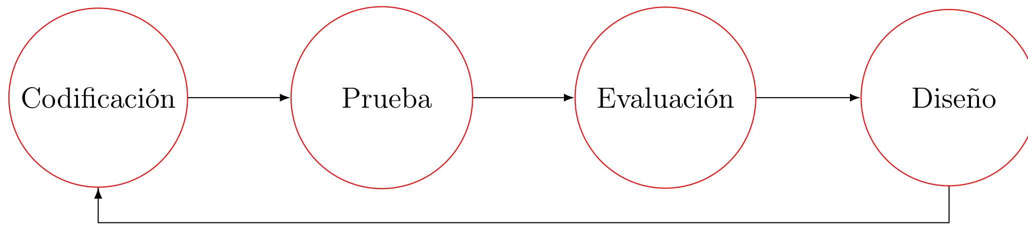


Figura 5.1: Iteración de etapas en XP.

sistema se asemeja a un *desarrollo evolutivo ágil*, concretamente al denominado *eXtreme Programming* (XP). De hecho, sus principios pueden aplicarse también a toda la planificación anterior. Puesto que el tiempo no parece un problema determinante, no seremos estrictos con las fases anteriormente mencionadas.

5.2.2. Modelo de proceso

Acabamos de comentar el modelo en el que se basa el desarrollo de nuestro sistema. Creemos poder dar algunos buenos argumentos de por qué es una buena decisión. Al contar con un grupo de tan solo dos integrantes, la comunicación entre ambos es clara, concisa y rápida, no sólo entre nosotros sino también con los tutores (a los que podemos asignar el rol de cliente), lo que ayuda a prevenir malentendidos que acaban en errores y problemas difíciles de vaticinar, aún con un proceso de desarrollo pesado con una planificación más exhaustiva como el proceso unificado. Otro factor a tener en cuenta es que, si bien los conceptos sobre los que asienta nuestro sistema no son elementales, no es excesivamente complejo el hecho de implementarlos. Además, al ser un área de la computación en el que no se ha investigado aún en exceso, puede surgir en cualquier momento la necesidad de cambios o implementación de nuevas funcionalidades. El uso de XP facilita este último hecho, pues intercala continuamente diseño y desarrollo de manera evolutiva e iterativa. En nuestro caso, era muy importante disponer de una versión ejecutable del programa desde el momento en que estuviera desarrollada la primera funcionalidad, lo cual hemos logrado. Para ello, diseñamos una vista, le damos forma mediante código y la ensamblamos con la lógica. Antes de continuar una nueva vista, nos aseguramos de que la funcionalidad implementada tiene una correcta actividad para evitar el encadenamiento de errores.

En la figura 5.1 podemos observar un diagrama de las etapas estándar de XP. Si bien la primera etapa se corresponde normalmente con la codificación, en nuestro caso siempre lo ha sido el diseño. Así, para cada funcionalidad, se diseña la vista, se implementa la misma, a continuación se desarrolla y enlaza la lógica y por último se prueba. Se itera sobre estas etapas para cada vista y funcionalidad establecidas. Se sigue así una planificación diseñada prácticamente para cada funcionalidad (planificación incremental), marcándose objetivos a

muy corto plazo pero sin perder de vista el objetivo final: completar el sistema y su correcto funcionamiento.

Para concluir, hemos tratado de hacer nuestro código lo más adaptable posible, no sólo por el tipo de proceso de desarrollo elegido, sino también para facilitar futuras extensiones del sistema. Esto, sumado a todo lo contado en los anteriores párrafos, justifica que nuestro proceso de desarrollo se ajuste a XP y nos reafirma en considerar que esta elección ha sido la más adecuada.

5.3. MTQC: Mutation Testing for Quantum Computing

MTQC son las siglas que dan nombre a nuestro sistema. Su principal funcionalidad es la de aplicar el proceso de testing de mutaciones en distintos lenguajes cuánticos, en nuestro caso `Q#` y *Qiskit*. La secuencia típica de acciones que seguirá un usuario de MTQC consistiría en generar mutantes del programa deseado, verificar los mutantes creados, aplicar al programa original y a los mutantes seleccionados dadas una serie de test y cotejar los resultados para detectar test que revelan una discrepancia.

Sobre estas cuatro fases hemos establecido toda nuestra planificación, análisis y diseño del proyecto, tomando estas etapas como estructura de cara al desarrollo del sistema.

5.3.1. Principales funcionalidades

Aunque el desarrollo dirigido por casos de uso no es propio de XP, nos inspiramos en ellos para determinar las principales funcionalidades del sistema que podemos identificarlas con las fases recientemente nombradas. Podríamos definir alguna otra adicional, como la elección un lenguaje cuántico o el reinicio del programa, pero el grueso del contenido de MTQC lo recogen estas cuatro:

1. *Generación de mutantes.* Se trata de la primera acción que el usuario debe realizar. Consta de dos entradas consistentes en dos listas: una del directorio de los archivos de código sobre los que se quiere realizar la mutación y otra de los operadores de mutación a aplicar. Arroja como salida una lista de mutantes vinculados al archivo original, el operador aplicado y la línea que sufrió la mutación. En caso de que la fabricación de un mutante provoque un error, se muestra por pantalla y se trata de generar el siguiente mutante.

2. *Visualización de mutantes.* La precondición para que este caso de uso se pueda llevar a cabo es que previamente el usuario haya generado al menos un mutante. Toma como entrada una lista de mutantes con la que el usuario puede interactuar para así poder comparar el contenido de dicho mutante y el archivo original. El objetivo es que el

usuario pueda tomar una decisión sobre si el mutante en cuestión es o no un candidato apto para ejecutar un test sobre él.

3. *Testeo de mutantes.* Se trata del caso de uso principal del programa por importancia, cantidad y calidad del código asociado al mismo. Es necesario verificar la precondición de haber generado al menos un mutante para realizar cualquier acción sobre el mismo. Enumeraremos las entradas.

    - Un archivo de código del lenguaje cuántico elegido en ese momento.
    - Una función contenida en el archivo anterior.
    - Una lista de todos los mutantes deseados para ejecución generados a partir de dicho archivo.
    - Un tipo de testing a aplicar (determinista o probabilista).
    - Un conjunto de test.

    La salida generará un conjunto de objetos que incluya los resultados obtenidos tras la ejecución de los test y sobre la muerte o no de los distintos mutantes que componen el conjunto pasado como parámetro.

4. *Visualización de los resultados de los test.* Por último, tenemos la opción de que el usuario vea los resultados que los test han producido al ser aplicados a la función original y a los mutantes, determinar cuántos de ellos han sido matados y la eficacia de dichos test. La precondición indispensable para llevar a cabo esta visualización es haber llevado a cabo las acciones mencionadas en el anterior caso de uso. Toma como entrada un conjunto de objetos que gestionan los resultados de los test y muestra la información correspondiente, previo tratamiento, por pantalla. En el caso de que el test realizado haya sido de tipo probabilista, tomará también como entrada un porcentaje de confianza.

Estas cuatro fases se han traducido en cuatro subsistemas bastante independientes. Aunque en un principio se pensó que las funcionalidades 3 y 4 conformarían un único subsistema, y estarían bajo una única vista, este planteamiento inicial cambió, principalmente, para no sobrecargar de información dicha vista y facilitar al usuario la lectura de los resultados, repartiendo las funcionalidades en diferentes subsistemas.

5.3.2. Diseño

Retomamos de nuevo las cuatro funcionalidades anteriores para componer la estructura principal del sistema. A la hora del diseño y desarrollo de MTQC se plantearon 4 subsistemas identificados cada uno de ellos con dichas funcionalidades. Para implementarlos se optó por

asignar a cada uno de ellos una pestaña visual independiente que quedan administradas bajo una misma vista conjunta. Esto se gestiona mediante el uso del patrón *Modelo-Vista-Controlador* (MVC). Pese a no ser del todo necesario, por poseer una única vista, se ha implementado el patrón *Observador* en el que la vista actúa como observador y la lógica como sujeto. Este patrón se ha usado para facilitar la adición de futuras interfaces, gráficas o no, que se pudieran desarrollar para el sistema.

En algunas partes del código se ha utilizado el patrón *Factoría*. Por ejemplo, según el tipo de testing escogido, este crea una instancia de una subclase concreta que gestiona los resultados arrojados por el test. Por otra parte, somos conscientes de que este mismo patrón es útil en combinación con *Observador* y MVC para facilitar, precisamente, implementar otras interfaces. Sin embargo, decidimos no aplicarlo como tal para facilitar el código. Pese a ello, de ser necesario, su ejecución no presentaría cambios significativos en el código, sino más bien algunas adiciones al mismo.

Detallaremos cada uno de estos subsistemas, pero antes vamos a exponer la jerarquía de paquetes que constituyen el sistema y unas breves indicaciones sobre su contenido y funcionalidad.

Estructuración del código por paquetes

- `model`. Contiene toda la lógica del programa. Además de todos los paquetes que aparecen a continuación, contiene las clases que conforman el patrón *Observador* (*Observer* y *Observable*) y la clase *Model* que gestiona el grueso de la lógica.
    - `mutantoperator`. Recoge la clase abstracta *MutantOperator* que gestiona un operador de mutación.
        - `qiskit` Contiene todas las implementaciones de operadores de mutación creados para *Qiskit*.
        - `qsharp` Contiene todas las implementaciones de operadores de mutación creados para `Q#`.
    - `mutant`. Contiene la clase *Mutant* que gestiona un mutante tras su creación. Guarda ruta al archivo mutante, ruta al archivo original y línea de código que sufrió la mutación.
    - `testing`. Contiene la clase abstracta *Testing* que gestiona las características del tipo de testing a realizar, por ejemplo, si es o no determinista. También contiene sus implementaciones.
    - `language`. Recoge clases que gestionan mutation testing y los lenguajes. Crea

archivos ejecutables a partir de un mutante y una entrada de datos y los ejecuta.

- `files`. Contiene la clase *TestFile* que gestiona los archivos finales creados por las clases del paquete `language`.
- `testresult`. Contiene la clase abstracta *TestResult* que gestiona los resultados de la ejecución de los test y también sus implementaciones, una por cada implementación de *Testing*.

- `view`. Está encargado de toda la vista del sistema.
  - `mutantgeneartorview`. Contiene la vista correspondiente al primer subsistema.
  - `mutantsviewer`. Contiene la vista correspondiente al segundo subsistema.
  - `testcaserunnerview`. Contiene la vista correspondiente al tercer subsistema.
  - `testresultview`. Contiene la vista correspondiente al cuarto subsistema.
  - `tools`. Contiene algunas clases auxiliares utilizadas en la vista.
- `control`. Contiene la clase *Control* que actúa como controlador del patrón *MVC*.
- `exception`. Recoge algunas excepciones del programa.
- `main`. Recoge la clase *Main* que contiene el método que inicia la ejecución del programa.

Estamos en disposición de hablar de cada subsistema con detenimiento y a esta tarea dedicaremos el resto de este capítulo. Mencionaremos los paquetes involucrados en cada uno de ellos, detalles de implementación, así como la aparición de problemas relevantes y la solución adoptada para ellos.

Subsistema I: Generador de mutantes

La generación de mutantes es el primer paso para realizar mutation testing. En la figura 5.2 se presenta el aspecto de la vista asociada. La interfaz en su conjunto y cada una de las pestañas tiene un diseño simple, tratando de minimizar la cantidad de información presentada, para facilitar el uso y entendimiento por parte del cliente.

Los componentes de esta vista se encuentran en el paquete `view.mutantgeneratorview`, aunque además hace uso de clases del paquete `view.tools`. En concreto, *JTableCheck*, que gestiona una tabla de objetos de una determinada instancia junto a casillas verificadoras asociadas a booleanos y *LogArea* que imprime información para el usuario. En cuanto a la lógica, los operadores se encuentran definidos en el paquete `model.mutantoperator`, y los mutantes generados se gestionan para su uso en el resto del programa mediante la clase *Mutant* del paquete `model.mutant`. La vista ofrece al usuario una lista de archivos en un

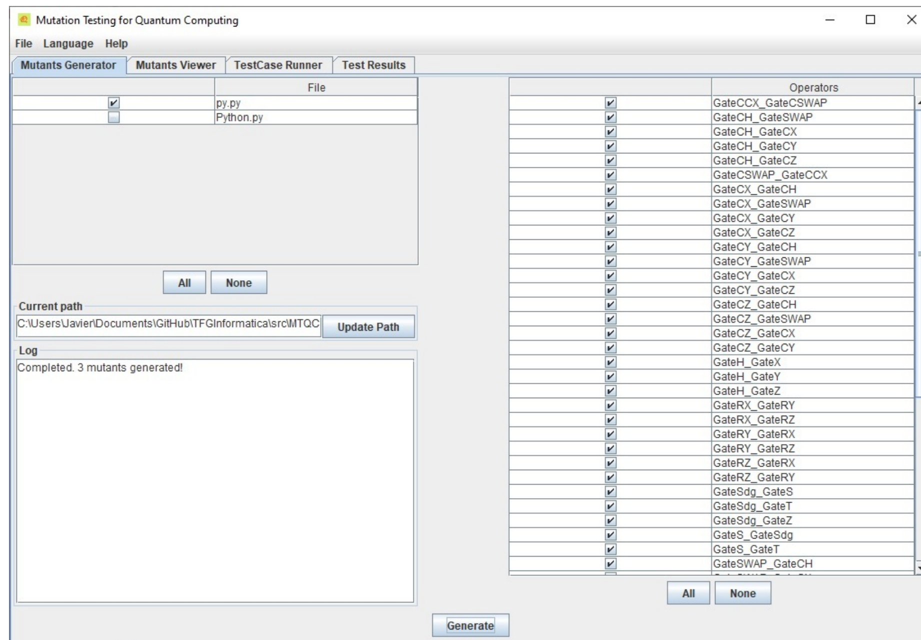


Figura 5.2: Vista del subsistema generador de mutantes.

determinado directorio que puede modificarse por el usuario. También aparece una lista de operadores de mutación acorde con el lenguaje cuántico seleccionado en ese momento. Tras la selección de los archivos y operadores se procede a llamar al controlador para que ordene a la lógica la creación de los mutantes. Para la generación de mutantes se ha implementado la técnica del *programador hábil*: sólo se aplica una mutación a cada archivo generado para simular que el programador ha cometido un único error. El proceso que sigue la lógica es sencillo: un operador contiene una secuencia de caracteres a ser buscada y otra con la que es reemplazada. Así, por cada archivo y operador, se generan tantos mutantes como veces se encontró dicha cadena. Todas estas acciones se realizan desde la clase *Model*.

El mayor reto en esta fase del desarrollo consistía en el hecho de establecer cuando hemos encontrado una cadena de caracteres coincidente con la mutación que queremos incurrir. En el caso de *Qiskit* esto es sencillo porque todas las instrucciones cuánticas son métodos que son llamados mediante un objeto perteneciente a la clase *QuantumCircuit*. Así, todas las instrucciones siguen el formato

```
objeto.instrucción(...)
```

y por tanto podemos buscar la cadena `.instrucción(` y sustituirla por `.mutación(` donde las palabras instrucción y mutación representan la conversión requerida. En el caso de `Q#` no es tan sencillo, ya que las instrucciones tienen el formato

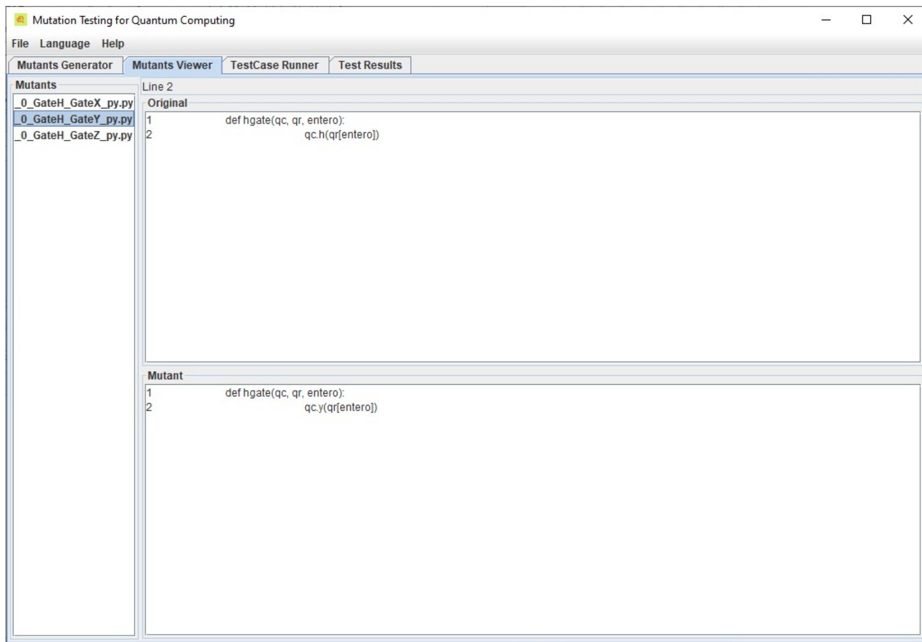


Figura 5.3: Vista del subsistema visualizador de mutantes.

```
instrucción(...)
```

así que no basta con sustituir `instrucción(` por `mutación(`. Supongamos que queremos realizar mutantes cambiando el operador de la puerta de *Hadamard*, representado por la instrucción `H(...)`, por la puerta X. Si durante el proceso de reemplazo encontrásemos un método (por raro que fuera) acabado en H, por ejemplo `applyH(...)`, estaríamos generando un mutante en el que esa instrucción sería cambiada por `applyX(...)` que no es el efecto que deseamos. La solución es comprobar el carácter anterior y verificar que se trata de un salto de línea, un espacio o una tabulación.

Otro problema similar aparece con los operadores de mutación aplicados a constantes de `Q#` como `One` y `Zero`. La solución es, de nuevo, la comentada en el párrafo anterior pero aplicando dicha comprobación también al carácter posterior.

Subsistema II: Visualizador de mutantes

El segundo subsistema es el más simple de los cuatro. Se trata de una vista sencilla (figura 5.3) que muestra la lista de mutantes generados y dos áreas de texto, que actúan como *display* de los ficheros original y mutante. Si no se han generado mutantes previamente, la lista aparecerá vacía.

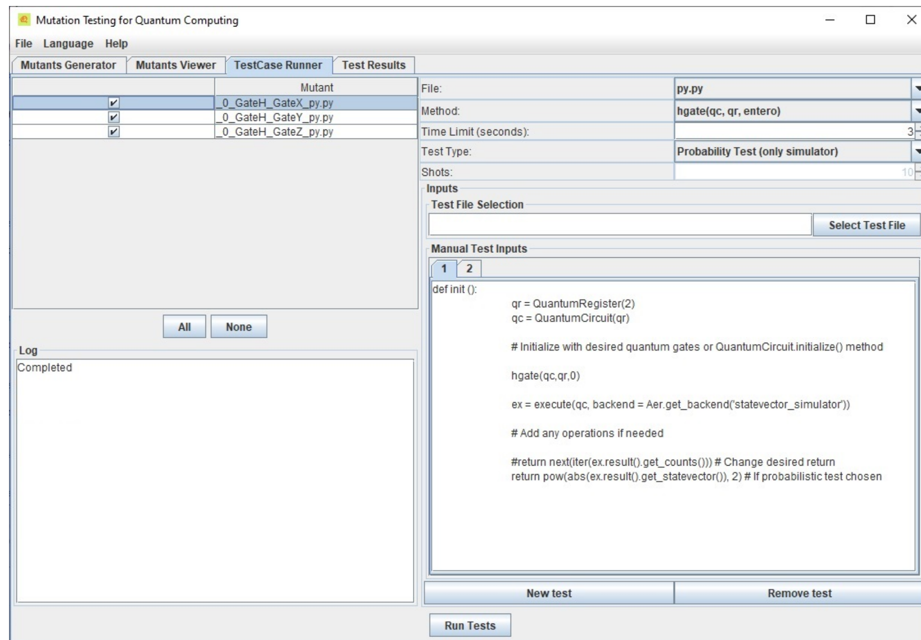


Figura 5.4: Vista del subsistema mutation testing y creación de test.

La lista de mutantes se gestiona mediante la clase *JMutantList* en el paquete `view.tools`. El resto de componentes se gestionan desde las clases *FileArea* y *MutantsViewer* del paquete `view.mutantsviewer`. Además, la lista antes mencionada contiene no sólo el identificador del mutante, sino que también incluye la instancia al completo.

El funcionamiento de este subsistema es sencillo. El usuario debe pinchar sobre el mutante que quiera observar y la vista mostrará el contenido de las rutas a los archivos original y mutado sin necesidad de hacer llamada al controlador y, por tanto, tampoco al modelo. Tal vez en este caso no se estén siguiendo las directrices del patrón *MVC* estrictamente, pero por su simplicidad optamos por este camino.

Subsistema III: mutation testing y creación de test

Es sin duda el componente más complejo del sistema. En él interviene gran parte de la lógica realizada para el proyecto. La funcionalidad se resume en ejecutar una elección de mutantes con unos test determinados para decretar qué porcentaje de ellos ha sido matado y la eficacia de los test utilizados.

En la figura 5.4 tenemos la vista que constituye el subsistema cuyos componentes se engloban dentro del paquete `view.testcaserunnerview` y que hace uso de las clases del paquete `view.tool`, *TabbedTextArea*, que gestiona las entradas para los test que el usuario

puede introducir mediante un sistema de pestañas; *JTableCheck*, tabla que gestiona la selección de mutantes; *TextField*, un simple campo de texto y *LogArea*, que imprime información de la ejecución del programa al usuario.

Aunque el usuario podía en el primer subsistema seleccionar más de un archivo para aplicar mutaciones sobre él, para realizar mutation testing debemos hacerlo, en este caso, de manera individual. Además, los archivos que MTQC genera para ejecutar todos los test se almacenan en una carpeta auxiliar. Por tanto, el programa que sirve de base para el proceso de testing de mutaciones no podrá usar archivos o librerías que no estén en el directorio por defecto del lenguaje correspondiente. Para solucionar esto, el usuario puede incluir la ruta a los archivos necesarios en el propio programa. Por ejemplo, en *Python* puede hacerse con el siguiente código:

```python
import sys
sys.path.insert(0, "path_to_your_package")
```

Cabe destacar que, en general, los programas cuánticos rara vez alcanzan una extensión suficiente para ser modulados en varios archivos y esta *complicación* no debería ser un problema en la inmensa mayoría de los programas con los que se trabaje.

Así, la primera decisión que ha de tomar el usuario es la de elegir el archivo deseado. En la vista se muestran los correspondientes a la ruta seleccionada en el primer subsistema. Tras la elección, se llama al controlador para que la lógica realice dos tareas. La primera es obtener todos los mutantes generados a partir del archivo seleccionado; la segunda es devolver todas las funciones encontradas en el archivo.

El usuario debe ahora realizar el resto de la configuración: elegir los mutantes a ejecutar, la función de llamada, el tiempo límite para la ejecución de cada test (para evitar ejecuciones infinitas debido a que nos hayamos quedado en un bucle a causa de la mutación), el tipo de testing, el número de ejecuciones, si este último es probabilista y, por último, las entradas o test.

En cuanto al tipo de testing, nuestra herramienta incluye dos opciones por defecto: uno determinista, que hemos denominado *QStateTesting*, y otro probabilista, *ProbabilisticTesting*. El primero compara las probabilidades arrojadas por los estados finales del sistema cuántico. Supongamos que una función sobre un sistema cuántico de un qubit devuelve $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ mientras que el mutante devuelve $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$, este test evalúa las probabilidades de cada estado, que en ambos casos son de $\frac{1}{2}$ para $|0\rangle$ y un $\frac{1}{2}$ para $|1\rangle$. Por tanto, no se mataría al mutante pese a que los estados cuánticos no son idénticos. Huelga decir que este primer tipo de testing solo es posible si se ejecuta sobre un simulador en un computador clásico. Para su implementación, en el caso de `Q#` hemos hecho uso de la instrucción `DumpMachine()` que imprime en un fichero o pantalla el estado cuántico en ese momento,

```
1 from qiskit import *
2 import numpy as np
3
4 qr = QuantumRegister(3)
5 qc = QuantumCircuit(qr)
6 init = [complex(0, 1/np.sqrt(2)), 0, 0, complex(1/np.sqrt(2), 0), 0, 0, 0,
    0]
7 qc.initialize(init, qr)
```

Figura 5.5: Inicialización de un circuito cuántico mediante un vector complejo en *Qiskit*.

mientras que el simulador *StatevectorSimulator* de *Qiskit* permite de igual modo acceder a dicha información.

El segundo tipo de testing, *ProbabilisticTesting*, compara la salida final de la función que, generalmente, será la medición de uno o más qubits. Por tanto, esta salida puede ser probabilista y su ejecución debe ser realizada en repetidas ocasiones con el fin de conseguir una certeza estadística sobre la muerte o no del mutante. Dicho testing puede ser aplicado en computadoras cuánticas usando *Qiskit*, basta con seleccionar una máquina de *IBM* disponible para la ejecución escribiendo el código correspondiente en la entrada. Sin embargo, hay que tener en cuenta que la disponibilidad de estas computadoras es muy limitada y las ejecuciones mensuales también lo son, así que se podría demorar bastante la obtención del resultado.

Como vimos en el capítulo anterior, la inicialización del estado cuántico deseado supone un problema añadido respecto de la computación clásica. El usuario debe dar una secuencia de puertas para conseguir el estado de entrada deseado. En el caso de *Qiskit*, existe una función del circuito cuántico (*QuantumCircuit*), `initialize()`, que permite inicializar el circuito en el estado deseado, siempre que el simulador utilizado sea *StatevectorSimulator*.

El código de la figura 5.5 inicializa el circuito cuántico de 3 qubits en el estado $\frac{i}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|011\rangle$. Nótese que el vector dado como ejemplo tiene norma uno; en otro caso la ejecución arrojaría un error.

En cualquier caso, la preparación de la entrada requiere un tratamiento previo. Para tratar esta entrada hemos facilitado dos métodos. El primero es mediante la carga de un archivo de texto en el que cada caso es separado por una línea con la secuencia de caracteres `***` y cada caso tiene que tener la estructura del segundo método que mencionamos a continuación. El segundo es un sistema de pestaña. Por defecto aparece una pestaña con un ejemplo estructural a seguir y en el caso de de añadir una nueva, aparece con una copia del contenido de la seleccionada previamente. El código dado por defecto para *Qiskit* está

```
1 def init ():
2   cr = ClassicalRegister(1)
3   qr = QuantumRegister(1)
4   qc = QuantumCircuit(qr, cr)
5
6   # Initialize with desired quantum gates or QuantumCircuit.initialize()
      method
7
8   # Call your method
9
10  ex = execute(qc, backend = Aer.get_backend('statevector_simulator'))
11
12  # Add any operations if needed
13
14  return next(iter(ex.result().get_counts())) # Change desired return
15  #return pow(abs(ex.result().get_statevector()), 2) # If probabilistic test
      chosen
```

Figura 5.6: Código dado por defecto en MTQC para inicializar un test en *Qiskit*.

reflejado en la figura 5.6.

Según lo deseado, podemos modificar el número de registro clásicos (*ClassicalRegister*) y cuánticos (*QuantumRegister*), añadir la inicialización deseada en cada caso y debemos llamar al método elegido. Por último, habrá que comentar uno de los dos `return` en función del tipo de testing elegido, según se indica, o incluso puede ser cambiado para devolver otro tipo de datos si se desea. Es importante recalcar que *Qiskit* construye un circuito con el uso de métodos de aplicación de puertas, pero realmente ese circuito no se ejecuta hasta que no se llama a la instrucción `execute(...)`. Si se llama a esta función en el método que se está utilizando como base del proceso de testing, basta con comentar o borrar la línea de código correspondiente de la figura 5.6. En cualquier caso, debemos mantener la tabulación establecida y no cambiar el nombre del método definido en la primera línea, pues se llama para poder inicializar la entrada. Si se opta por la introducción vía archivo, este método tiene que estar definido para cada caso de igual modo.

Para el caso del lenguaje `Q#` tenemos indicaciones similares (véase figura 5.7). De nuevo, debemos mantener la estructura general y modificar lo necesario siguiendo las indicaciones, ya haya sido elegido el método de entrada mediante el uso de la vista de pestañas o mediante la carga de archivo.

Tenemos así todo listo para proceder: archivo, método, mutantes, tipo de testing y conjunto de test que vamos a aplicar, entre otros. Todos estos datos se gestionan con las clases mencionadas anteriormente para cada uno de ellos. El usuario está listo para ejecutar los

```
operation MainQuantum() : // Define main function output type {

  // Select desired Qubit number to be used
  using (register = Qubit[2]) {

      // Inicialize variables and Qubits

      // Call method and save output

      // If probabilistic test chosen.
      // DumpMachine("temp.txt");

      // Reset all qubits to Zero state
      ResetAll(register);

      // Return output
  }
}
// Define any other operation if needed as input
```

Figura 5.7: Código dado por defecto en MTQC para inicializar un test en `Q#`.

test y la vista llama al controlador que preprocesa los datos antes de enviárselos a la lógica. El primer objetivo es preparar nuevos archivos que añadan las entradas establecidas a los archivos a ejecutar: mutantes y original. Dichos archivos son creados por las clases del paquete `model.language` y gestionadas por la clase *TestFile* del paquete `model.files`.

El siguiente paso consiste en generar un archivo principal en *Python* que se ejecutará mediante una llamada al sistema (por tanto, el usuario debe tener agregado *Python* en su *path*) y que llamará a todos los archivos generados en el paso anterior para obtener las salidas producidas por los test. Tras ser estos ejecutados, MTQC recoge las salidas resultantes de la ejecución asignándolas al archivo original o mutante y a uno de los test según corresponda. Esta gestión de datos se realiza mediante una de las clases del paquete `model.testresult`, según el tipo de testing elegido.

Para acabar, todos los archivos generados durante esta fase son borrados y se manda una actualización de los resultados obtenidos al último subsistema que veremos a continuación.

Subsistema IV: Visualizador de resultados

En último lugar tenemos el subsistema dedicado a visualizar los resultados. La vista está consituida por un sencillo sistema de pestañas, una para cada test, que contiene una tabla donde se comparan los resultados y se indica qué mutantes han muerto y cuáles no

Mutation Testing for Quantum Computing

File Language Help

Mutants Generator | Mutants Viewer | TestCase Runner | Test Results

Confidence Interval (%)

0 Set confindence (%)

Test 1 | Test 2

Resume

Live Mutants #	Killed Mutants #	Total Mutants #	Mutant Score %
0	3	3	100.0%

Name	Result	Killed
original_0	[0.5 0.5 0. 0.]	
_0_GateH_GateX_py.py_0	[0. 1. 0. 0.]	Yes
_0_GateH_GateY_py.py_0	[0. 1. 0. 0.]	Yes
_0_GateH_GateZ_py.py_0	[1. 0. 0. 0.]	Yes

Figura 5.8: Vista del subsistema visualizador de resultados.

(véase figura 5.8). Los componentes de la vista se encuentran íntegramente en el paquete `view.testresultview` a excepción de la clase *ResultTable*, que gestiona cada tabla, del paquete `view.tools`. Como hemos mencionado, los resultados que se muestran en esta vista se actualizan automáticamente al final de la ejecución del subsistema anterior.

Además, el usuario puede modificar la confianza utilizada para matar a un mutante en un test probabilístico. Supongamos que tenemos un total de n estados posibles $(|0\rangle, ..., |n-1\rangle)$ en cierto sistema cuántico y hemos ejecutado k veces tanto el archivo original como el mutante. Definimos las probabilidades de medición del estado cuántico $|i\rangle$ para el archivo original como

$$p_{|i\rangle,o} = \frac{f_{|i\rangle,o}}{k}$$

donde $f_{|i\rangle,o}$ denota el número de veces que la salida del programa original fue el estado $|i\rangle$. Análogamente, para cierto mutante m definimos las probabilidades de medición del estado cuántico $|i\rangle$ como

$$p_{|i\rangle,m} = \frac{f_{|i\rangle,m}}{k}$$

donde $f_{|i\rangle,m}$ denota el número de veces que la salida del mutante m fue el estado $|i\rangle$. Así un mutante m muere si se verifica

$$\text{máx}\{|p_{|i\rangle,o} - p_{|i\rangle,m}| : 0 \leq i \leq n-1\} > c$$

donde c denota el parámetro de confianza, que puede ser modificado por el usuario de nuestro sistema. Obviamente el máximo del conjunto anterior está comprendido entre 0 y 1, por estarlo las probabilidades definidas anteriormente. El parámetro de confianza en MTQC por defecto es del 1 %, pero puede modificarse para aplicar otro porcentaje que el usuario considere más adecuado. Tras esto, se contactará con la clase *Model* de la lógica que solicitará a cada instancia de la clase *TestResult* que recalcule la muerte o no del mutante en función del nuevo valor indicado y comunicará a la vista los resultados.

Capítulo 6

Ejemplo de uso de MTQC

En este capítulo mostramos un ejemplo del uso de MTQC. Implementaremos el algoritmo de *Deutsch-Jozsa* [Deutsch y Jozsa 1992] en los lenguajes *Qiskit* y `Q#` y aplicaremos pruebas de mutación sobre ambos códigos. Antes veremos que problema plantea dicho algoritmo.

6.1. Pruebas de mutación sobre Deutsch-Jozsa

Sea $f\colon \{0,1\}^n \longrightarrow \{0,1\}$ una función binaria que bien puede ser constante ($f(x) = 0$ o $f(x) = 1$ para todo $x \in \{0,1\}^n$) o bien es balanceada (la salida es 0 para la mitad de entradas y 1 para la otra mitad). El problema consiste en determinar qué tipo de función es. Desde el punto de vista clásico, como el número de entradas de f es 2^n, el caso peor se da cuando las 2^{n-1} primeras salidas devuelven el mismo valor. Por ello debemos verificar $2^{n-1}+1$ salidas para determinar si estamos en el caso constante o el balanceado. El algoritmo cuántico de *Deutsch-Jozsa* lo resuelve en una sola iteración.

6.1.1. Algoritmo Deutsch-Jozsa

Previamente a mostrar el código de nuestro ejemplo, analizamos en detalle cada paso del que consta el algoritmo de Jozsa. En primer lugar, si nuestra función f toma valores en $\{0,1\}^n$, debemos crear un circuito con $n+1$ qubits. Todos los qubits deberan tomar el estado $|0\rangle$ menos el último de ellos, que utilizaremos como auxiliar e inicializamos al estado $|1\rangle$. De esta forma, nuestro sistema se encuentra en un primer estado dado por:

$$|\phi_0\rangle = \left|0 \overset{n}{\cdots} 01\right\rangle$$

A continuación, aplicamos una puerta *Hadamard* a cada qubit, obteniendo de esta forma

el siguiente estado:

$$|\phi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{i=0}^{2^n-1} |i\rangle (|0\rangle - |1\rangle)$$

donde i se corresponde con su representación binaria. Posteriormente, hay que aplicar el operador U_f tal y como fue definido en la sección 2.4.2. Si denotamos por x a los n qubits de entrada y por y al qubit auxiliar se tiene:

$$U_f : \begin{array}{ccc} x & \longrightarrow & x \\ \mathrm{y} & \longrightarrow & y \oplus f(x) \end{array}$$

El diseño de esta U_f puede ser muy complejo, dependiendo de cómo sea la función f. En nuestro ejemplo hemos elegido una función que facilita la creación del operador U_f. Tras aplicar el operador U_f, se obtiene un tercer estado dado por:

$$|\phi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{i=0}^{2^n-1} |i\rangle (|f(i)\rangle - |1 \oplus f(i)\rangle)$$

Ahora, como $f(i)$ sólo puede tomar valores binarios, podemos simplificar la expresión anterior:

$$|\phi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{i=0}^{2^n-1} (-1)^{f(i)} |i\rangle (|0\rangle - |1\rangle) \tag{6.1}$$

A partir de este punto, el estado del último qubit $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ puede ser ignorado, pues no será relevante para el resto de cálculos. A continuación, debemos aplicar una puerta *Hadamard* a cada uno de los n qubits restantes. Previamente a realizar dicha operación, vamos a ver una representación matemática de la aplicación simultánea de puertas *Hadamard* que facilita la comprensión de la parte final del algoritmo. Cuando contamos con un solo qubit, una puerta *Hadamard* viene dada por:

$$H : \begin{array}{ccc} |0\rangle & \longrightarrow & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |1\rangle & \longrightarrow & \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{array}$$

Por tanto, podemos escribir el caso general, donde $x = 0$ o $x = 1$ como:

$$H|x\rangle = \tfrac{1}{\sqrt{2}} \sum_{z \in \{0,1\}} (-1)^{x \cdot z} |z\rangle$$

donde $x \cdot z$ denota el producto escalar bit a bit, módulo 2. Veamos ahora el caso específico en el que tenemos 2 qubits y, por tanto, aplicamos una puerta *Hadamard* a cada uno de ellos:

$$\begin{aligned} H^{\otimes 2}\left|x_1, x_2\right\rangle &= H\left|x_1\right\rangle \otimes H\left|x_2\right\rangle \\ &= \tfrac{1}{\sqrt{2}} \sum_{z_1 \in \{0,1\}} (-1)^{x_1 \cdot z_1} \left|z_1\right\rangle \otimes \tfrac{1}{\sqrt{2}} \sum_{z_2 \in \{0,1\}} (-1)^{x_2 \cdot z_2} \left|z_1\right\rangle \\ &= \tfrac{1}{\sqrt{2^2}} \sum_{z_1, z_2 \in \{0,1\}} (-1)^{x_1 \cdot z_1 + x_2 \cdot z_2} \left|z_1, z_2\right\rangle \end{aligned}$$

De esta misma forma, podemos representar la aplicación de puertas *Hadamard* a n qubits como se muestra a continuación:

$$H^{\otimes n}\left|x_1, \cdots, x_n\right\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{z_1, \cdots, z_n \in \{0,1\}} (-1)^{x_1 \cdot z_1 + \cdots + x_n \cdot z_n} \left|z_1, \cdots, z_n\right\rangle$$

Si retomamos ahora la notación para el sumatorio utilizada a lo largo de la demostración, podemos expresar el resultado anterior como:

$$H^{\otimes n}\left|x_1, \cdots, x_n\right\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{z=0}^{2^n-1} (-1)^{x \cdot z} \left|z\right\rangle$$

Por tanto, partiendo de la expresión recogida en la ecuación 6.1, si aplicamos una puerta *Hadamard* a los n primeros qubits (recordemos que el estado del último qubit ya no era relevante), se obtiene un nuevo estado ϕ_3 dado por:

$$\begin{aligned} \left|\phi_3\right\rangle &= \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} (-1)^{f(i)} \left[\frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} (-1)^{i \cdot j} \left|j\right\rangle \right] \\ &= \frac{1}{2^n} \sum_{j=0}^{2^n-1} \left[\sum_{i=0}^{2^n-1} (-1)^{f(i)} (-1)^{i \cdot j} \right] \left|j\right\rangle \end{aligned} \tag{6.2}$$

Una vez que hayamos obtenido este estado, vamos a estudiar cual es la probabilidad de obtener el estado $\left|0 \overset{n}{\cdots} 0\right\rangle$ al medir los n primeros qubits. En virtud de la ecuación 6.2, el cuadrado de la amplitud del estado $\left|0 \overset{n}{\cdots} 0\right\rangle$ viene dado por:

$$\left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} (-1)^{f(i)} (-1)^{i \cdot j} \right|^2 = \left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} (-1)^{f(i)} \right|^2$$

ya que $j = 0 \overset{n}{\cdots} 0$. Por tanto, si la función f es *balanceada*, la mitad de los términos $(-1)^{f(i)}$ evaluaran a 1, mientras que la otra mitad evaluarán a -1, otorgando por ello *probabilidad* 0

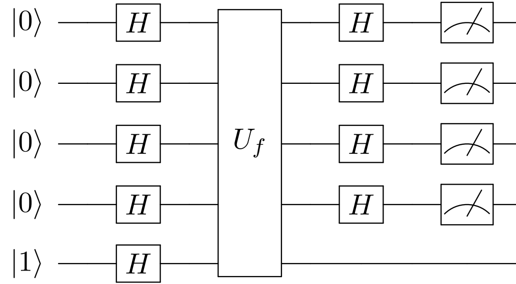


Figura 6.1: Circuito de una implementación de Deutsch-Jozsa.

de obtener el estado $\left|0\overset{n}{\cdots}0\right\rangle$. Si, por el contrario, la función f es *constante*, entonces todos los términos $(-1)^{f(i)}$ tendrán el mismo signo, no cancelándose entre ellos y, por ellos, se obtendrá *probabilidad* 1 de medir el estado $\left|0\overset{n}{\cdots}0\right\rangle$.

De esta forma, se puede discernir de manera completamente determinista si la función f es constante o balanceada. En la figura 6.1 podemos ver el circuito que resume todo lo contado anteriormente.

6.1.2. Deutsch-Jozsa en MTQC

Una vez que hemos visto cuales son los pasos a aplicar para ejecutar el algoritmo de Deutsch-Jozsa, se procede a exponer el código de dicho algoritmo para los lenguajes `Q#` y *Qiskit*, además de las funciones f utilizadas. Empecemos por estas últimas. Emplearemos 3 funciones $\{0,1\}^4 \longrightarrow \{0,1\}$ que vienen definidas por:

- $f_1(x_1, x_2, x_3, x_4) = \begin{cases} 1 \text{ si } x_1 = 1 \\ 0 \text{ si } x_1 = 0 \end{cases}$
- $f_2(x_1, x_2, x_3, x_4) = 0$.
- $f_3(x_1, x_2, x_3, x_4) = 1$.

Es inmediato observar que la primera de estas funciones está balanceada mientras que las otras dos son constantes. Las U_f respectivas son fáciles de implementar. En el primer caso basta con aplicar una puerta *CNOT* empleando como controlador el qubit correspondiente al bit x_1 y como qubit receptor el de salida. En el caso de U_{f_2} no hay que realizar ninguna operación, aunque pondremos simbólicamente la puerta identidad aplicada al qubit de salida. Por último, U_{f_3} viene determinada por una puerta X aplicada al qubit de salida. En la figura 6.2 podemos ver el código en *Qiskit* de la implementación del algoritmo de *Deutsch-Jozsa* para un sistema de 5 qubits y cada una de las funciones U_f expresadas anteriormente,

mientras que en la figura 6.3 encontramos el mismo algoritmo en Q#.

En primer lugar, tratamos de la generación de mutantes. Utilizando MTQC sobre los ficheros que contienen los algoritmos (que no incluyen las funciones U_f) obtenemos un total de 9 mutantes en el caso de *Qiksit*, mientras que en Q# obtenemos 10. Esencialmente, son los mismos 9 mutantes en uno y otro lenguaje, sólo que Q# tiene uno adicional que cambia la constante *One* por *Zero*. En cuanto a los comunes, son distintas mutaciones de las puertas H y X a otras puertas unitarias.

Ya tenemos todo listo para la ejecución de nuestro algoritmo. En este caso, el valor de los qubits de entrada debe ser $|00000\rangle$. Nótese que carece de sentido introducir valores distintos, pues el correcto funcionamiento del algoritmo depende de que se verifique dicho estado al comienzo de la ejecución. Sí podemos, sin embargo, utilizar cada una de las U_f definidas como parámetro de entrada. En el caso de *Qiskit*, al estar escrito sobre *Python*, permite la definición de una función de manera local para utilizarla como argumento de un método sin necesidad de especificar el tipado (pues *Python* carece del mismo). Por otro lado, Q# requiere la definición de dicho tipado y, además, no permite la definición de funciones de manera local, por lo que debemos hacerlo de manera global.

Vamos a analizar la ejecución de las dos opciones que tenemos para realizar el análisis del proceso de testing (a las que hemos denominado anteriormente *determinista* y *probabilista*) sobre 3 tests, uno por cada U_f definida. Comencemos con *ProbabilisticTest*. Para ello, se ha tomado la decisión de ejecutar cada mutante con cada test 1000 veces. Nótese que pese al determinismo de este algoritmo, los mutantes generados pueden no serlo. Hemos adecuado la salida para que sea una cadena de caracteres que retorne "balanceada" o "constante" según corresponda.

Analicemos a continuación los resultados obtenidos para este tipo de testing en cada uno de los lenguajes. Cabe mencionar que el parámetro de confianza utilizado en ambos casos ha sido del 1 %. En el caso de la ejecución del primer test, el resultado para ambos lenguajes es muy similar. En Qiskit, se han matado 8 de los 9 mutantes generados (*Mutant Score* de 88,9 %), mientras que en Q# el mutante adicional sobrevive, luego se obtiene un *Mutant Score* de 80 %. Es interesante analizar el mutante común superviviente, y lo haremos al final de esta sección, pues adelantamos que sobrevive a los 3 test realizados.

Para el segundo test se obtienen resultados semejantes para ambos lenguajes. En el caso de Qiskit se consigue matar a 6 de 9 mutantes (*Mutant Score* de 66,7 %) mientras que en el caso de Q# se elimina a los mismos mutantes y al adicional, obteniendo un *Mutant Score* de 70 %.

Por último, los resultados obtenidos para el tercer test son, una vez más, similares para Qiskit y Q#. De hecho, los resultados son casi idénticos (salvo pequeñas variaciones en los

porcentajes de las salidas, que podrían aproximarse mediante un mayor número de ejecuciones) a los obtenidos en el segundo test. En total, mediante estos tres test, se eliminan todos los mutantes menos uno de ellos. Este mutante es el obtenido mediante la sustitución de la puerta X, que niega el qubit de salida, por la puerta Y. Esto se debe a que al aplicar una puerta X o Y al estado $|0\rangle$ se obtiene el mismo estado $|1\rangle$ módulo un cambio de fase global, por lo que el comportamiento cuántico de ambos estados es el mismo. De esta forma, se obtiene un mutante equivalente que sobrevivirá independientemente del test.

En conclusión podemos determinar que el primer test es el más adecuado, no solamente por tener un *Mutant Score* más elevado, sino porque los otros dos se identifican con una U_f que representa una función constante. Independientemente de la dimensión del espacio de entrada, sólo existen dos funciones constantes: la función constante cero y la función constante uno, cuya U_f asociada se construye mediante una única puerta I o X, respectivamente, aplicada exclusivamente al qubit de salida, que es el único que no es relevante a la hora de medir. Esto implica que todas las mutaciones realizadas sobre los operadores que afecten a dicho qubit (en nuestro caso la puerta X), no sean detectadas en el caso de que la función sea constante. Es por ello que el primer test, al representar una función balanceada, es propenso a eliminar este tipo de mutantes.

Por último, analicemos los resultados obtenidos a la hora de ejecutar el *QStateTest*. Debido a la naturaleza determinista de este algoritmo, este tipo de test no es el más indicado. Para una mejor adaptación, se ha considerado adecuado eliminar las lineas de código referentes a la medición, pues esta altera de manera no determinista el estado interno del programa. Es por esto que ahora disponemos de 9 mutantes para cada lenguaje, pues el mutante adicional que teníamos previamente para `Q#` era relativo a la medida de los qubits.

Para el primer test, los resultados son idénticos a los obtenidos mediante *ProbabilisticTest*, obteniendo un *Mutant Score* de 88,9 % para ambos lenguajes. Sin embargo, en el segundo y tercer test notamos una mejoría al lograr matar un mutante adicional respecto a *ProbabilisticTest*, logrando un *Mutant Score* de 77,8 %. Esto se debe a que hemos detectado una mutación que afecta al qubit de salida que, como se ha explicado previamente, era indetectable mediante el uso de funciones constantes, ejecutadas usando *ProbabilisticTest*. Véase que estamos obteniendo las probabilidades asignadas al estado cuántico, lo que incluye también al qubit de salida de U_f. De nuevo, el mutante equivalente no muere bajo ningún test, pues las amplitudes de los estados son iguales. En definitiva, el primer test es de nuevo el más eficiente.

```python
def deutschjozsa (qc, qr, cr, uf):
    if len(qr) != 5:
        print("El numero de qubits debe ser 5.")
        return

    # Negacion  del ultimo qubit
    qc.x(qr[-1])

    # Aplicacion de una puerta de Hadamard a cada qubit
    for r in qr:
        qc.h(r)

    # Aplicamos U_f
    uf(qc, qr)

    # Aplicacion de una puerta de Hadamard a cada qubit de entrada
    for r in qr[:-1]:
        qc.h(r)

    qc.measure(qr[:-1], cr)

def uf_1 (qc, qr):
    # f(x_1,x_2,x_3,x_4) = 1 si x_1 vale 1, 0 en otro caso. Balanceada
    # U_f cnot, como control el primer cubit, X aplicada a qubit de salida
    qc.cx(qr[0], qr[-1])

def uf_2 (qc, qr):
    # f(x_1,x_2,x_3,x_4) = 0. Constante
    # U_f Identidad qubit de salida
    qc.iden(qr[-1])

def uf_3 (qc, qr):
    # f(x_1,x_2,x_3,x_4) = 1. Constante
    # U_f not en qubit de salida
    qc.x(qr[-1])
```

Figura 6.2: Código en *Qiskit* de una implementación de Deutsch-Jozsa.

```
operation DeutschJozsa(register:Qubit[], U_f:(Qubit[] => Unit)):String {
  let nQubits = Length(register);
    if (nQubits != 5) {
      return "El numero de Qubits debe ser 5 ";
    }
    else {
        // Negacion del ultimo Qubit (salida)
        X(register[nQubits - 1]);
        // Poner cada qubit en superposicion
        for(q in register) {
          H(q);
        }
        // Aplicamos la Uf
        U_f(register);
        // Aplicamos una puerta Hadamard a cada Qubit de entrada
        for(i in 0..nQubits - 2) {
          H(register[i]);
        }
        // Constante si medimos el estado 0. Balanceada en otro caso.
        mutable allZeros = true;
        for(i in 0..nQubits - 2) {
            if(M(register[i]) == One){
              set allZeros = false;
            }
        }
        if (allZeros){
          return "Constante";
        } else {
          return "Balanceada";
        }
  }
}
operation uf(register : Qubit[]) : Unit{
    //f(x_1,x_2,x_3,x_4) = 1 si x_1 vale 1, 0 en otro caso. Balanceada
    CNOT(register[0], register[Length(register) - 1]);
}
operation uf2(register : Qubit[]) : Unit{
    //f(x_1,x_2,x_3,x_4) = 1. Constante
    I(register[Length(register) - 1]);
}
operation uf3(register : Qubit[]) : Unit {
    //f(x_1,x_2,x_3,x_4) = 0. Constante
    X(register[Length(register) - 1]);
}
```

Figura 6.3: Código en Q# de una implementación de Deutsch-Jozsa.

Capítulo 7

Conclusiones

Para acabar este trabajo presentamos algunas conclusiones sobre los temas tratados, tanto teóricos como prácticos, y discutimos algunas ideas propicias para la mejora y continuidad del proyecto.

7.1. Conclusiones (español)

En vista a los resultados del capítulo anterior prácticamente no se encuentran diferencias sustanciales entre los resultados obtenidos para uno y otro lenguaje, al menos en algoritmos sencillos. Estas diferencias pueden ser más visibles en algoritmos de mayor complejidad e incluso añadiendo otros operadores de mutación de carácter clásico, pues ambos lenguajes combinan instrucciones cuánticas con otras tradicionales como la instrucción `for` en nuestro ejemplo.

Por otro lado, hay que tener en cuenta que, sea o no determinista un algoritmo cuántico, su ejecución en un computador cuántico actual, como los que pone a nuestra disposición IBM, acarreará múltiples errores de cómputo. Debido a ellos, algunos de los estados que teóricamente deben tener probabilidad cero podrán obtenerse como resultado. Esto se debe a la inestabilidad de los ordenadores cuánticos actuales, lo que denota que aún queda mucho trabajo por hacer, no sólo con el objetivo de la ampliación del número de qubits, sino también minimizando los errores que aparecen por ruido e interferencias en estos sistemas. Este hecho hace que en la aplicación de mutation testing sobre programas ejecutados en un ordenador cuántico debamos considerar aumentar el número de ejecuciones para cada mutante y test, con respecto a las que realizaríamos en un simulador. De esta manera conseguimos reducir el impacto de dichas interferencias.

7.2. Conclusions (English)

In view of the results of the previous chapter, there are practically no substantial differences between the results obtained for the two languages, at least in simple algorithms. These differences may be more visible in more complex algorithms and even adding other classical mutant operators, since both languages combine quantum instructions with other traditional ones like the `for` instruction in our example.

On the other hand, we have to take into account that, whether a quantum algorithm is deterministic or not, its execution in a current quantum computer, such as those made available by IBM, will lead to multiple computation errors. Due to them, some of the states that theoretically should have zero probability can be obtained as a result. This is due to the instability of current quantum computers, which denotes that there is still a lot of work to be done, not only with the aim of increasing the number of qubits, but also minimizing the errors that appear due to noise and interference in these systems. This fact means that in the application of mutation testing on programs executed in a quantum computer we must consider increasing the number of executions for each mutant and test, with respect to those we would carry out in a simulator. In this way we manage to reduce the impact of such interferences.

7.3. Implementaciones futuras

Queremos aportar algunas ideas con las que se puede dar continuidad a este proyecto. Estas funcionalidades y líneas de trabajo han surgido durante la implementación de MTQC.

La implementación de *weak mutation testing* podría ser una buena alternativa a los tipos de testing ya existentes en el programa. Se trataría de un testing determinista a realizar sobre simulador que verificara las condiciones dadas al final del capítulo 3.

Aunque no relacionado directamente con mutation testing, sería interesante estudiar si algunos procesos de testing en computación clásica se pueden codificar como problemas de computación cuántica.

Por otro lado, sería conveniente darle más facilidades al usuario para analizar sus programas. Entre ellas, hemos detectado las siguientes. En primer lugar, se podría permitir que puedan añadirse nuevos operadores de mutación durante la ejecución y que estos puedan ser guardados para futuras pruebas de mutación o guardar el proceso en un determinado momento para que la tarea pueda ser retomada en otra ejecución. En segundo lugar, sería conveniente facilitar el uso de otros archivos y librerías externas para evitar, como mencionamos antes, que el usuario no tenga que realizar una adición de código para encontrar la ruta a estos archivos. Otra herramienta útil podría ser la exportación de la tabla final de

resultados a un formato más manejable como *Excel* o *.csv*.

Además de los dos lenguajes considerados en este proyecto, puede ser interesante comparar el comportamiento de los algoritmos clásicos con los obtenidos en un lenguaje más puramente cuántico como es el caso de *QASM*.

Por último, sería conveniente habilitar una opción para que un mutante pueda tener 2 o más mutaciones y mejorar la eficiencia del programa mediante la paralelización de cada una de las ejecuciones realizadas al aplicar los casos de prueba.

Bibliografía

Ammann, Paul y Jeff Offutt (2016). *Introduction to software testing.* 2nd. Cambridge University Press.

Arute, Frank y col. (2019). «Quantum supremacy using a programmable superconducting processor». En: *Nature* 574.7779, págs. 505-510.

Bennett, Charles H y Gilles Brassard (1987). «Quantum public key distribution reinvented». En: *ACM SIGACT News* 18.4, págs. 51-53.

Bernstein, Ethan y Umesh Vazirani (1997). «Quantum complexity theory». En: *SIAM Journal on computing* 26.5, págs. 1411-1473.

DeMillo, Richard A., Richard J. Lipton y Frederick G. Sayward (1978). «Hints on test data selection: Help for the practicing programmer». En: *Computer* 11.4, págs. 34-41.

Deutsch, David (1985). «Quantum theory, the Church–Turing principle and the universal quantum computer». En: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818, págs. 97-117.

Deutsch, David y Richard Jozsa (1992). «Rapid solution of problems by quantum computation». En: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907, págs. 553-558.

Dijkstra, Edsger W. (1972). *Chapter I: Notes on structured programming.* Academic Press Ltd.

Feynman, Richard P. (1982). «Simulating Physics with Computers». En: *International Journal of Theoretical Physics* 21.6/7.

Häffner, Hartmut y col. (2005). «Scalable multiparticle entanglement of trapped ions». En: *Nature* 438.7068, págs. 643-646.

Hierons, Robert M., Mercedes G. Merayo y Manuel Núñez (2010). «Mutation Testing». En: *Encyclopedia of Software Engineering.* Ed. por Phillip A. Laplante. Taylor & Francis, págs. 594-602.

Jones, Jonathan A., Michele Mosca y Rasmus H. Hansen (1998). «Implementation of a quantum search algorithm on a quantum computer». En: *Nature* 393.6683, págs. 344-346.

Kintis, Marinos y col. (2018). «Detecting Trivial Mutant Equivalences via Compiler Optimisations». En: *IEEE Transactions on Software Engineering* 44.4, págs. 308-333.

Lipton, Richard (1971). «Fault diagnosis of computer programs. Student Report». En: *Carnegie Mellon University* 2, pág. 2.

Madeyski, Lech y col. (2014). «Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation». En: *IEEE Transactions on Software Engineering* 40.1, págs. 23-42.

Matuschak, Andy y Michael A. Nielsen (2019). *Quantum Computing for the Very Curious.* `https://quantum.country/qcvc`.

Nielsen, Michael A. e Isaac L. Chuang (2001). «Quantum computation and quantum information». En: *Phys. Today* 54, págs. 60-2.

Offutt, A. Jefferson (1992). «Investigations of the software testing coupling effect». En: *ACM Transactions on Software Engineering and Methodology* 1.1, págs. 5-20.

Polo Usaola, Macario (2020). «Quantum Software Testing». En: *CEUR-WS.org*.

Rieffel, Eleanor y Wolfgang Polak (2000). «An introduction to quantum computing for non-physicists». En: *ACM Computing Surveys* 32.3, págs. 300-335.

Shor, Peter W. (1994). «Algorithms for quantum computation: discrete logarithms and factoring». En: *35th Annual Symposium on Foundations of Computer Science.* IEEE, págs. 124-134.