Noname manuscript No.
(will be inserted by the editor)

Bug Characteristics in Quantum Software Ecosystem

Mohamed Raed El Aoun · Heng Li · Foutse Khomh · Lionel Tidjon

Received: date / Accepted: date

Keywords Quantum computing, quantum programming, quantum software engineering, issue reports, quantum program bug.

1 Experiment examples and results

In this section, we report and discuss the results and more examples as complementary to the paper of our research questions 2 . We discuss the bugs with detailed examples for each quantum component and most occurring bug type.

RQ2: How are the bug distributed among the quantum program components?

12 quantum computing components are identified in the bug issue reports. Figure 1 shows the identified hybrid-quantum program components. Prior work identified [1] five components `pre-processing, post-processing, gate operation, state preparation, and measurement`. In Figure 1 we extended the component set to finally have 13 as follows.

- `Compiler (CP)` translates the quantum program circuit into device-level language
- `Gate operation (GO)` is a set of reversible operations that alter the state of the qubit and generate superposition. These reversible gate can be represented as matrix.
- `Simulator (SM)` simulates the execution of a circuit on a quantum computer.

Mohamed Raed El Aoun, Heng Li, Foutse Khomh, Lionel Tidjon
Department of Computer Engineering and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
E-mail: {mohamed-raed.el-aoun, heng.li, foutse.khomh, lionel.tidjon}@polymtl.ca

Table 1: Distribution of the quantum component bugs across the quantum software categories. (In the 376 analyzed bug reports, 43 questions were not identified as bugs.)

	VIS	SP	GO	Post-P	Pre-P	EM	CP	MS	PC	SM	MN	QCA	O	# bugs
Quantum programming framework	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	58
Experimental quantum computing	✓	✓	✓				✓	✓	✓	✓		✓	✓	47
Quantum algorithms	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	34
Quantum circuit simulators	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	43
Quantum compilers			✓				✓	✓	✓				✓	37
Quantum utility tools	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	41
Quantum-based simulation	✓	✓	✓	✓	✓		✓		✓		✓	✓	✓	38
Quantum Machine learning	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	35
Total Bugs	15	40	56	17	9	15	68	19	17	10	2	25	40	333

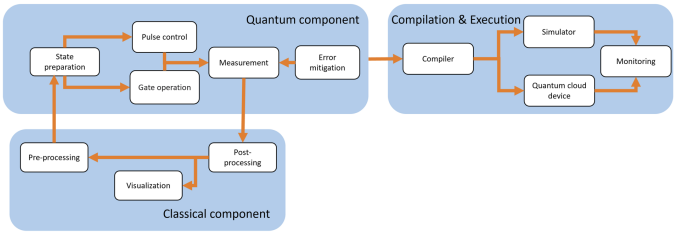

Fig. 1: A general component in a hybrid quantum program application

- `State preparation (SP)` is an operation to generate an arbitrary state of qubit.
- `Measurement (MS)` measures the final state of a qubit after superposition.
- `Post-processing (Post-P)` translates the quantum information (i.e., measurement probability) to classical computer.
- `Pre-processing (Pre-P)` generates the state preparation circuit and initializes the register for quantum computer. This component is executed on classical computer.
- `Pulse control (PC)` generates and controls signals to create custom gates and calibrate the qubits
- `Error mitigation (EM)` corrects the measurement error when computing the qubit state probability.
- `Quantum cloud access (QCA` is a client interface to connect into quantum cloud service provider
- `Visualisation (VIS)` creates plots and visualisations.
- `Monitoring (MN)` controls the good execution of the program and the performance of the execution environment.
- `Other (O)`: Every bug other than the defined components.

The result of our qualitative analysis for identifying the bugs in the quantum program software component is presented in the table 1. Among the 376 analyzed bug reports, 43 questions were not bugs.

Table 2: Distribution of bug types across the quantum component

	VIS	SP	GO	Post-P	Pre-P	EM	CP	MS	PC	SM	MN	QCA	O	# bugs
Configuration bugs	3	3	4	-	-	2	11	3	1	2	-	8	10	47
Data types and structures bugs	3	5	10	1	3	-	12	2	2	1	-	2	5	46
Missing Error handling	-	1	-	-	-	1	4	-	3	2	-	2	-	12
Performance bugs	-	-	3	1	1	3	3	1	1	2	-	1	3	19
Permission/deprecation bugs	-	-	2	1	-	-	1	-	-	-	-	1	2	7
Program anomaly bugs	2	21	30	9	4	5	26	12	6	1	-	5	2	123
Test code-related bugs	-	2	4	-	-	4	12	1	4	-	1	-	5	33
DataBase related bugs	-	-	-	-	-	-	-	-	-	-	-	-	1	1
Documentation	-	1	3	-	-	-	-	-	-	1	-	1	11	17
Gui related bugs	8	3	1	3	-	-	-	-	-	-	-	-	-	15
Misuse	-	1	1	-	-	-	-	1	-	-	-	1	-	2
Network bugs	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Monitoring	1	-	-	-	-	-	-	-	-	-	-	-	-	1

We identified 5 components that are dominated by bugs directly related to quantum computing including `gate operation, state preparation, measurement, pulse control, and error mitigation` ordered by their number of bugs. Moreover, the `simulator` shows a slightly high number of hybrid bugs (classical and quantum bugs), this can be explained by the complexity of the operation that has to be simulated since they are trying to simulate quantum physic phenomena using very complex linear algebra calculation. Finally, the `compilers` appears to be the most buggy with 63 bugs identified.

13 bug types were identified in the quantum software ecosystem. The table 2 shows the type of the derived bug in the quantum software projects. During the manual sampling, we have identified 13 bug types as follows.

- `Program anomaly bugs` are introduced when enhancing existing code or caused by prior bad implementation. Bug concerned about bad return values or unexpected crashes due to logic behind the code.
- `Test-code related bugs` are happening in the test code. Problem reported due to adding or updating test cases and intermittently executed tests.
- `Data type and structure bugs` are related to first data type problems such as undefined or mismatch type second data structure bugs like bad shape bugs or using the wrong data structure.
- `Missing error handling` is reported for the absence of exceptions that cause the program to crash or bad error messages defined to debug the program.
- `Configuration bugs` are related to configuration files building. The bug can be caused by the external library that must be updated or incorrect file paths or directories.
- `Network bugs` are related to connection or server issues.
- `GUI related bugs` are related to the graphical user interface such as layout, text box, and button layout.
- `Performance bugs` are related to performance. This category cover memory overuse, endless loops, and energy consumption.
- `Permission/deprecation bugs` cover two type of bugs. On one hand, we have bugs related to the removal or modification of deprecated calls or APIs on the other hand bugs related to missing or incorrect API permission.

- `Database related bugs` are related to the connection between the database and the main application.
- `Documentation` are related to documentation typos, not up to date documentations and misleading function documentation.
- `Monitoring bugs` are related to bad logging practices such as wrong logging levels, too much logging, or missing log statement.
- `Misuse` is a wrong usage of a function that leads to a bug in the code.

We identified 5 dominant bug type while doing the manual analysis: `program anomaly`, `data type and structure`, `configuration issues`, `test-code related` and `enhancement and feature request`. `Program anomaly` is the most dominant bug type with 123 bugs. While inspecting the bug reports and their commit message, we notice that most program anomaly bugs come from bad implementation in the quantum algorithm or mathematical formulation of the problem. In fact, gate operation and state preparation components are based on complex maths, in those two categories, program anomaly is the dominant bug type. `Configuration issues` is the second most appeared bug type with 45 bugs. Most of the quantum concepts are based on linear algebra and use the complex data type, this makes the implementation challenging for developers. Therefore, We find that data type and structure is the third most occurring category with 40 bugs.

Bug types in different quantum components. In this section, we discuss the types of bugs in each of the quantum components as shown in the figure1. For every component, we present the top two or three bug types identified in the table2. For every bug, we show the bug code, error message, and potential fix.

Gate operation (GO): **We identify 56 bugs in the gate operation component**. Gate operations are based on complex mathematical operations using imaginary numbers. Translating the maths into code is challenging for developers which increases the chances of errors. Table 1 shows that `gate operations` component is the second most buggy with 56 bugs detected in our sample. In this component, we have the two occurring bug types which are `program anomaly` and `data-type and data structure`. They respectively occur 27 and 7 times. As an example for each bug type:

1. **Program anomaly**: Qubit states are measured values that are prone to noise. In the code snippet 1(a) a users reported a bug about the `Qiskit Optimize1qGatesDecomposition()` function basis operation. The bug state that the function will not optimize the qubit state if its net operation is sufficiently close to the identity. The problem root cause was using the array_equal function instead of trying to approximate the qubit state value as shown in the bug fix 1(b).
2. **Data type and structure**: In `Pennylane`[1], passing a complex array/-matrix into `QubitUnitary()` function in the code 2(a), raises `typeError` exception and changes the other element in the matrix into complex type

[1] `https://github.com/PennyLaneAI/pennylane`

```
qc = qk.QuantumCircuit(2)
qc.cx(0, 1)
qc.u1(1e-17, 0)
qc.cx(0, 1)
# Optimize chains of single-qubit gates does not optimize single
    qubit rotation epsilon (bug triggered in this line)
Optimize1qGatesDecomposition(basis=['u1', 'u2', 'u3'])(qc)
```

(a) Trigger of the bug "qiskit (Id 6473)"

```
# Repository: Qiskit/qiskit-terra
# Fix in file:qiskit/transpiler/passes/optimization/
    optimize_1q_decomposition.py
# Class:Optimize1qGatesDecomposition, Method : run(), Line:83
-if np.array_equal(run[0].op.to_matrix(), np.eye(2)):
+if np.allclose(run[0].op.to_matrix(), np.eye(2), 1e-15, 0):
    dag.remove_op_node(run[0])
```

(b) Bug fix

Listing 1: The execution of the code snippet (as shown in (a)) triggers a wrong gate decomposition operation because it does not optimize single qubit rotation. The bug was fixed by changing the comparison operator from np.equal to np.allclose (as shown in (b)). "qiskit (Id 6473)"

as shown in 2(b). The error was caused by the casting rules of `Numpy`, the presence of one complex element will cause all the arrays to become complex-valued. A developer has fixed this problem using a list instead of calling for `Numpy` array with a small change in the code as shown in 2(c). It took developers 16 days to locate and fix the bugs even though the bug seems simple and easy to fix.

State preparation (SP): **state preparation has the third-highest number of bugs with 40 bugs**. As shown in the table1, the bugs appear in all the project categories except `quantum compilers` and `quantum utility tools` categories. During the state preparation phase, one tries to give the qubit an arbitrary state of 0 or 1. For this purpose, one tries to rotate or control the qubit into the desired state. As an example, one can use gate decomposition as a technique for state preparation. In the table2, we observe that program anomaly is the most dominant bug type with 21 bugs, while data type and structure come second with only 5 bugs. However, we detected a rare bug of missing error handling type. For a deep understanding of the nature of the bugs in state preparation, we are going to look into examples for each of the three bug types. Quantum states are mathematical entity provides a probability distribution of each outcome of a state measurement. Since a qubit can have multiple states, mathematically they are represented as matrices. To initialize the state of a qubit, one has to manipulate that matrix. The number of state preparation bug indicates the difficulty that developers are having with implementing the quantum algorithms. These bugs can be addressed with the

```python
def circuit1(theta, matrix):
    qml.RX(theta, wires=0)
    # Unitary transformation triggers the bug when storing the
    current values of the quantum function parameters
    qml.QubitUnitary(matrix, wires=1)
    return qml.expval(qml.PauliZ(0)), qml.expval(qml.PauliZ(1))
```

(a) Trigger of the bug "PennyLaneAI (Id 541)"

```
TypeError: RX: Real scalar parameter expected, got <class 'numpy.
    complex128'>
```

(b) Error message

```
#Repository: PennyLaneAI/pennylane
#Fix in file: pennylane/qnodes/base.py,
#class:BaseQNode, method:_set_variables(), line: 329
-    Variable.positional_arg_values = np.array(list(_flatten(args)
    ))
+    Variable.positional_arg_values = list(_flatten(args))
```

(c) Bug fix

Listing 2: The execution of the code snippet (shown in (a)) triggers the error message (shown in (b)) because when storing the current values of the circuit with complex arrays, due to NumPy casting rules, a single complex value will cause the entire array to become complex valued even if the values were originally reals. The bug fix was using list instead of numpy array (as shown in (c))

help of data refinement and verification tools. Also, missing error handling bugs can be avoided by encouraging the quantum developer for testing their code.

1. **Program anomaly**: State preparation is the first step in constructing the circuit for a quantum program. In this step, you are looking to arbitrarily set the state of the qubit into 0 or 1. In the example3, we observe a program anomaly where the developer has written the program in the wrong order. The purpose of the circuit is to measure the state of qubit 0 and map the state into a classical bit. However, the program raises a `CircuitError`3. Firstly, the problem was due to the use of the line `QuantumCircuit()` which is not doing anything since it is immediately over-ridden by `statepreparation()`. Secondly, the measurement is not mapped to the register (classical bit) because of the misuse of the `measure()` function. A developer proposed a fix as shown in the code3.
2. **Data type and structure**: Quantum states are represented as matrices. Developers represent the matrices as arrays or lists. Also, state preparation uses linear algebra mathematics to control the state of qubits. This can prove to be challenging without data validation tools to support it. In the example4, the output of the operation for the gate decomposition function has the wrong shape. This bug has given an error in the qubit state. The bug was caused by the returned shape of the state matrices and a simple fix

```
q = QuantumRegister(4)
c = ClassicalRegister(1)
circuit = QuantumCircuit(q,c)
ang = feature_train[3]
# Applying state preparation after quantum Circuit initialization
    triggers the bug
circuit = statepreparation(ang, circuit, [0,1,2,3])
circuit.measure(0, c)
```

(a) Trigger of the bug "qiskit (issue Id 5837)"

```
CircuitError: 'register not in this circuit'
```

(b) Code to reproduce bug

```
#Fix file: In the code snipped (shown in (a)), line 3 and 7
-circuit = QuantumCircuit(q,c)
-circuit.measure(0, c)
+circuit.measure(0, 0)
```

(c) Code to reproduce bug

Listing 3: The execution of the code snipped (shown in (a)) trigger the message error (shown in (b)) because the quantum circuit (line 3 in code snipped (a)) is immediately overridden by the state preparation (line 6 in a code snipped (a)). The bug fix was to map the 0th measured qubit into the 0th classical bit and remove the circuit initialization (as shown in (c).

```
x = enr_destroy([3,3],2)
z = x[0].eigenstates()[1][0]
#  The fock state representation return object with a list of the
    quantum states with wrong shape which triggers the bug (Z and
    y are incompatible)
y = enr_fock([3,3],2,[1,1])
```

(a) Trigger of the bug "qutip (issue Id 820)"

```
#Repository : qutip/qutip
# Fix file: qutip/states.py, Method: enr_fock(), line: 930
-    return Qobj(data, dims=[dims, 1])
+    return Qobj(data, dims=[dims, [1]*len(dims)])
```

(b) Bug fix

Listing 4: The execution of the code sipped (shown in (a)) causes a bug because the dimension property (shape) of the fock state representation (line 4 in code snipped (a)) is not compatible with the dimension of z (line 2 in code snipped (a)) and the multiplication of z and y does not work. The bug fix was to adjust the dimension of the fock state representation return object (as shown in (b))

has removed this bug as shown in the example 4. This bug did not generate any error in the code. However, to identify the bug, the user had to execute

the program until the end and notice that the state preparation output is wrong with the help of prints. To help the community, data verification tools should prove helpful to avoid such errors.

3. **Missing error handling**: State preparation is based on complex mathematical operations in linear algebra. Thus, usual error messages make it harder to fix a bug. In the example 5 a developer is struggling to understand a bug in `qutip` program during the state preparation step. In fact, when passing an initial state vector that is neither a state vector nor square density matrix in `qutip` should throw an exception but instead gives a ValueError. This has led the developer to open an issue. The problem was diagnosed as a missing error handling in the code. To fix the bug, a developer proposed to add error handling clauses and provide a more comprehensive error message as shown in the code5.

```
from qutip import *
D = 13
psi0 = tensor(coherent(D, 100e-9), qeye(D))
# mesolve return wrong error message when passing incompatible
    dimensions (Bug trigger)
result = mesolve(qeye(D**2), psi0,[0, 100e-9], [destroy(D**2)])
```

(a) Trigger of the bug "qiskit (Id 1456)"

```
ValueError: operands could not be broadcast together with shapes
    (28561,) (2197,)
```

(b) Error message

```
# Repository: qutip/qutip
# Fix file: qutip/qutip/mesolve.py, method:
    _test_liouvillian_dimensions, line:320
-if isket(rho0):
-    rho0 = ket2dm(rho0)
+if L_dims[0] != L_dims[1]:
+    raise ValueError("Liouvillian had nonsquare dims: " + str(
    L_dims))
```

(c) Bug fix

Listing 5: The execution of the code snipped (shown in (a)) correctly raise a message error (shown in b), but the message error content does not reflect the problem. The bug fix was overwriting the error message (as shown in (c))

the

Pulse control (PC): From the table1, the pulse control component registered 17 bugs during the manual analysis. Pulse control bugs generally appear when a developer would like to define a custom gate and manually calibrate the qubits. For this purpose, developers need to generate a pulse (signal) and a scheduler to calibrate the qubit. From the table 2, the most 2 occurring bug types in pulse control component are `program anomaly` and `test-code related`.

We present examples of the pulse control bugs for each identified type.

1. **Program anomaly**: In `qutip`, when trying to modify the frequency of a periodic pulse, the frequency stays constant. The problem was in the definition of the pulse initial phase as follows

```
# Repository: qutip/qutip
# Fix file : qutip/control/pulsegen.py
# Class: PulseGen, method: PulseGenTriangle(), Line: 863
-phase = 2*np.pi*self.freq*t + self.start_phase
+phase = 2*np.pi*self.freq*t + self.start_phase + np.pi/2.0
```

Listing 6: Pulse generation function return a signal with fixed period when changing the frequency. The bug fix was to redefine the phase parameter "qutip (issue id 412)"

To fix the bug, the developer had Changed the pulse's initial starting point to zero by adding the `np.pi/2.0` to the phase.

2. **Test-code related**: Pulse generation and control give the possibility to developers to customize and create quantum gates. The complexity of this step makes it important to emphasize testing since any wrong transformation is going to lead to a wrong result. Test code-related bug is the second most occurring bug type in this component. For example, in `Qiskit` bug reports number 2527[2], a unitest is failing because of the wrong specified parameter type. The expected input variable type in this test should be int, float, or complex. Therefore to fix the bug, a developer did as shown in the code 7

```
#Repository: Qiskit/qiskit-terra
#Fix File: qiskit-terra/test/python/pulse/test_cmd_def.py
#Class: TestCmdDef, method:test_parameterized_schedule, line:
    86
#Bug triggered in line 4 and 8
-sched = cmd_def.get('pv_test', 0, '0', P2=-1)
+sched = cmd_def.get('pv_test', 0, 0, P2=-1)
self.assertEqual(sched.instructions[0][-1].command.value, -1)
with self.assertRaises(PulseError):
    -cmd_def.get('pv_test', 0, '0', P1=-1)
    +cmd_def.get('pv_test', 0, 0, P1=-1)
```

Listing 7: The execution of the test building parameterized schedule fails because of the wrong parameter type (as shown in code snippet line 1 and 5). The bug fix was to change the parameter type from string to integer "qiskit (issue id 2527)"

Measurement (MS): We identified 19 bugs present in six of the quantum software categories as presented in the table 1. Six bug types were detected in our sample. `Program anomaly bugs` are the most occurring with 12 bugs. Also, we identified 3 `configuration bugs` and 2 `data type and structure bugs`.

[2] `https://github.com/Qiskit/qiskit-terra/issues/2527`

1. **Data type and structure**: Qubit measurement returns the state probability of a qubit. The sum of probability states must be equal to 1. In `qibo`, we identifies a bug that states : "`Probabilities do not sum to 1`[3]". When running a circuit with more than 25 qubits. In the code8, we illustrate the code to reproduce the bug along with the message error and the bug fix proposed. The problem has occurred because the `sum()` function does not return a normalized set of probabilities distribution. To fix the problem, a developer proposed to cast the tensor into complex128. This bug was caused by data type and structure issues which look to commune with 33 bugs in our sample. Quantum software programs algorithms are based on probability and linear algebra which makes the implementation of quantum algorithms complex for developers. Therefore, providing libraries with pre-implemented mathematical functions and data validation tools can help the quantum community.
2. **Program anomaly**: In quantum programming, the order of the component (e.g., state preparation, unitary transformation, gate decomposition) impacts the final state of the qubit. In the example9, a user is measuring the qubit value and then computing the expectation value. This error has to lead to a non-valid measurement because the user is computing the expectation value of the collapsed state. To fix the error, a user must switch the order of the last two lines in the code 9

Error mitigation(EM): We identified 15 error mitigation bugs distributed along 5 quantum software categories highlighted in the table 1. Among the identified bugs we have detected 5 `program anomaly` bugs, 4 `test-code related bugs` as shown in table 2.

1. **Program anomaly** Quantum computers introduce noise to the computation while executing state operation or unitary transformation. In the measurement, to reduce the noise effects on the qubit, developers use error mitigating techniques. Error mitigation should not fail on different qubit mapping to classical bits. In qiskit, the issues id 832 [5] report a bug measurement with error mitigation on the same set of qubits but in a different order. This caused a faulty state measurement and causes the error mitigation to fail. The bugs were challenging to fix, it took 25 days and 3 files changed to resolve the problem [6].

Classical Component In this section, we present bug reports of the classical components as `pre-processing`, `post-processing` and `visualization` component as described in Fig. 1. For each component, we present bug types along with a detailed code snipped, error message, and the fix.

Post-processing(Post-P): We detected 17 bugs related to `post-processing` distributed among all the quantum software categories except `experimental`

[3] `https://github.com/qiboteam/qibo/issues/517`
[5] `https://github.com/Qiskit/qiskit-aqua/issues/832`
[6] `https://github.com/Qiskit/qiskit-aqua/pull/865`

```
NQUBITS = 26
c = Circuit(NQUBITS)
for i in range(NQUBITS):
    c.add(gates.H(i))
output = c.add(gates.M(TARGET, collapse=True))
for i in range(NQUBITS):
    c.add(gates.H(i))
# Running the circuit with more than 25 qubit return sum of qubit
    states probabilities =! 1 (Bug triggered)
result = c()
```

(a) Trigger of the bug "qibo (issue id 517)"

```
File "mtrand.pyx", line 933, in numpy.random.mtrand.RandomState.
    choice
ValueError: probabilities do not sum to 1
```

(b) Error message

```
#Repository:qiboteam/qibo
#Fix file: qibo/src/qibo/core/states.py
#Class:VectorState, method: probabilities(), line: 95
def probabilities(self, qubits=None, measurement_gate=None):
    unmeasured_qubits = tuple(i for i in range(self.nqubits) if i
    not in qubits)
-    state = K.reshape(K.square(K.abs(K.cast(self.tensor))), self.
    nqubits * (2,))
+    state = K.reshape(K.square(K.abs(K.cast(self.tensor, dtype="
    complex128"))), self.nqubits * (2,))
    return K.sum(state, axis=unmeasured_qubits)
```

(c) Bug fix

Listing 8: The execution of the code snippet (shown in (a)) triggers the message error (shown in (b)) because the sum of final state probabilities is different from 1. The bug fix was to cast the array value to complex type (as shown in (c))

`quantum computing` and `quantum compilers` as highlighted in the table 1. In table 2 we observe 9 `program anomaly` bug and 3 `gui related` bugs.

1. **Program anomaly**: To exploit the quantum information in the qubit, developers need to map the information from the quantum dimension to the classical dimension. Therefore, `Qiskit` and `Cirq` offer the functionality, after measurement, to map the quantum state to the classical bit register. `mitiq`[7] offers the possibility to convert `Cirq` circuit to `Qiskit` and vise versa. However, to correctly map the quantum state to a classical bit, developers have to keep the same order of the qubits. In the issue report of `mitiq`[8], the conversion of the circuit from `Cirq` to `Qiskit` affect the measurement because it reverses the qubits order. The code 10 reports

[7] `https://github.com/unitaryfund/mitiq`
[8] `https://github.com/unitaryfund/mitiq/issues/551`

```
#Repository: qiskit/qiskit-aer
N = 12
#Initialize the circuit
qc = QuantumCircuit(N)
qc.x(range(0, N))
qc.h(range(0, N))
for kk in range(N//2,0,-1):
    qc.ch(kk, kk-1)
for kk in range(N//2, N-1):
    qc.ch(kk, kk+1)
# Measaure the qubit state before compute the expectation value
    triger the bug
qc.measure_all()
qc.save_expectation_value(Pauli('Z'*12), range(12), label='
    after_meas_expval')
```

Listing 9: The execution of the code snippet returns a different probability expectation state from the awaited result. The bug is triggered because we measure the state before computing the expectation value. The bug fix was to reverse the measure (line 12) and computing the expectation value (line 13) "qiskit-aer (issue id 1334)"[4]

the bug and its fix to retaining the measurement order when converting to `Qiskit` circuits. We encourage the quantum developers to use tools

```
q, c = QuantumRegister(2), ClassicalRegister(2)
circuit = QuantumCircuit(q, c)
circuit.x(q[0])
circuit.measure(q, c)
# to_qiskit : shuffle the order of qubit and causes the bug
converted = to_qiskit(from_qiskit(circuit))
```

(a) Trigger of the bug "mitiq (issue Id 551)"

```
#Repository: unitaryfund/mitiq
#Fix file:  mitiq/conversions.py, method: _measurement_order, line
    : 131
def _measurement_order(circuit: qiskit.QuantumCircuit):
    """
    Returns measurements ordered as they appear going left-to-
    right through the circuit
    """
    order = []
    for (gate, qubits, cbits) in circuit.data:
        if isinstance(gate, qiskit.circuit.Measure):
            order.append((*qubits, *cbits))
    return order
```

(b) Proposed bug fix

Listing 10: The execution of the code snippet (shown in (a)) triggers a bug because he shuffle the order of the qubits when converting from Qiskit to Cirq circuit. This leads to mix which ClassicalRegisters correspond to which QuantumRegisters. The bug fix was to add a function to keep the same qubit order during the conversion (shown in (b))

or frameworks for test case generations. Also, reviewing the equivalence classes for the function can prove effective for quantum developers to help improve the quantum software quality.

Pre-processing(Pre-P): 9 bugs are classified as pre-processing bugs distributed among the quantum software categories as shown in the table 1. Among the identified bugs, we find 4 bug types as `data types and structures bugs`, `performance`, `program anomaly` and `enhancement and feature request`. The `program anomaly` and `data types and structures` bugs are the most occurring with respective values of 4 and 3 bugs. An example of each bug type is presented below.

1. **Program anomaly**: To execute quantum programs efficiently, data is embedded into quantum bits. In fact, this process is mapping the classical data into n-qubit quantum states by transforming data into a new space where it is linear. AmplitudeEmbedding is a mapping technique in PennyLaneAI in which a bug[9] was reported and it took 6 days to fix. In the code11, the normalization in the embedding function is breaking the differentiability for any data that is encoded as a `TensorFlow` or `Pytorch` tensor. Developers have tried pre-implemented normalization techniques in `NumPy` and `TensorFlow` but the bug persists. The only left solution was to hand-code the normalization as illustrated in the bug fix code11. Providing the quantum computing community with libraries dedicated to linear algebra computation in quantum will support the progress of quantum programs.
2. **Data types and structures**: For quantum computation, developers initialize multiple qubits represented as tensors in a quantum program. Data type and structure bugs look common because quantum computation is done in high-dimensional space. For example, the code **??**, shows a data type structure bugs in the tensor object size. It is caused by the quirk of fancy indexing of `Numpy` during the tensor contraction. To fix the bug, developers had to modify the indexing approach for adjacent indices in the sum function as shown in the bug fix report11.

Visualization (VIS): Quantum developers use visualization first to picture the qubit states to observe the quantum state vectors and the transformation actions. Second, draw the circuit as the last step to building a quantum program. We identify 15 bugs in the visualization component that appear in all the quantum programs categories except the `quantum compilers`. In the detected bugs, we find 8 in the `Gui related bugs` category. An example is presented below.

1. **Gui related**: Quantum circuits are the main focus to build a quantum program. The circuit can be challenging to debug or to validate its built structure by reading only the code. Libraries like `Cirq` and `Qiskit` offer visualization features to draw the circuit architecture. The drawing is meant to depict the physical arrangement of the gates, wires, and components. The visualization functions in `Qiskit` are basic support modules

[9] `https://github.com/PennyLaneAI/pennylane/issues/365`

```
import numpy as np
import pennylane as qml
from pennylane.templates.embeddings import AmplitudeEmbedding

dev = qml.device('default.qubit', wires=3)
@qml.qnode(dev)
def circuit(data):
#Amplitude embedding normalization function not behaving as
    expected (Bug trigger)
    AmplitudeEmbedding(data, wires=[0, 1, 2])
    return qml.expval.PauliZ(0)

data = np.ones(shape=(8,)) / np.sqrt(8)
circuit(data)
```

(a) Trigger of the bug "PennyLaneAI (issue id 365)"

```
TypeError: unsupported operand type(s) for +: 'Variable' and '
    Variable'
```

(b) Error message

```
#Repository: PennyLaneAI/pennylane
#Fix file: pennylane/templates/embeddings.py, method:
    AngleEmbedding, line: 118
-if normalize and np.linalg.norm(features, 2) != 1:
-    features = features * (1/np.linalg.norm(features, 2))
+norm = 0
+for f in features:
+if type(f) is Variable:
    +norm += np.conj(f.val)*f.val
+else:
    +norm += np.conj(f)*f
```

(c) Proposed bug fix

Listing 11: The execution of the code snippet (shown in (a)) triggers a message error (shown in (b)) because Amplitude embedding normalization in numpy is too strict and does not allow small tolerance. The bug fix was redefine the normalization function (as shown in (c))

and can come with bugs. For example, the bug reports [10] in `Qiskit` is stating misaligned barriers in the circuit, which change the interpretation of the circuit. In the code13, we show the bug in Qiskit when a user wants to draw a circuit and the proposed fix. Libraries dedicated to quantum circuit visualization such us `matplotlib` and `seaborn` in python benefit the developer to analyze and debugging the circuit.

Compilation and Execution In this section, we provide the bugs in the compiler, simulator, quantum cloud device, and monitoring component. For each component, we present the bug type and examples.

[10] `https://github.com/Qiskit/qiskit-terra/issues/3107`

```
import numpy as np
import qutip as qt
dat = np.arange(2 * 3 * 3 * 4).reshape((2 * 3, 3 * 4))
qobj = qt.Qobj(dat, dims=[[2, 3], [3, 4]])
# Due to numpy fancy indexing tensors fail to adjust dimensions (
    Bug trigger)
qt.tensor_contract(qobj, (1, 2)
```

(a) Trigger of the bug qutip (issue id 386)

```
ValueError: total size of new array must be unchanged
```

(b) Error message

```
#Repository: qutip/qutip
#Fix file: qutip/tensor.py, method: _tensor_contract_single, Line:
    362
-return np.sum(arr[sl], axis=0)
+contract_at = i if j == i + 1 else 0
+return np.sum(arr[sl], axis=contract_at)
```

(c) Bug fix

Listing 12: The execution of the code snippet (shown in (a)) triggers a message error(shown in (b)) because the tensor fails to contract out a pair (i,j) when j==i+1 due of numpy fancy indexing. The bug fix was define a function to contracts a dense tensor along a single index pair (shown in (c))

```
qc = qk.QuantumCircuit(2)
#The circuit draw in latex places barrier in the wrong place
qc.draw(output='latex')
```

(a) Trigger of the bug qiskit (issue id 3107)

```
#Repository: Qiskit/qiskit-terra
#Fix file:  qiskit/visualization/latex.py
#Class: QCircuitImage, method: _build_latex_array, line:772
-self._latex[start][column] = "\\qw \\barrier{" + str(span) + "}"
+self._latex[start][column - 1] += " \\barrier[0em]{" + str(span)
    + "}"
+self._latex[start][column] = "\\qw"
```

(b) Trigger of the bug

Listing 13: The execution of the code snippet (shown in (a)) plots a circuit with barrier not aligned correctly. The bug fix was to adjust the barrier column index (shown in (b))

Compiler (CP): From the table 1 we observe that compiler component issues are present in all the quantum ecosystem software categories with a total of 68 bugs. As shown in the table2, the four most occurring bug types are program anomaly (23 bugs), test-code related (12 bugs), data type and structures (11 bugs), and configuration (11 bugs). In order to have a deeper look into how the bugs are manifesting for each one of the four bug types, we present a detailed example.

1. **Program anomaly**: The example 14presents a qisit-terra bug, error message, and fix. In this bug, a developer is complaining because he can not use `Reset()` function from Qiskit in the transpilation phase. Eventually, he got stuck with the Qiskit error 14(b). As a fix, a developer suggested changing a condition statement in the function definition. while performing the fix, another bug in the code was discovered in the same function where he was verifying the instance of the object in which he was checking the wiring variable to map the qubit measurement to classical bits. The bug appears hard to fix since it had many interactions and two other issues related to this bug were opened on GitHub (Issues number 5127,5543). Moreover, to fix these issues, it took the developers two releases.
2. **Test-code related bug**: In `Pennylane`, we identify bugs in a test file because of dependency issues. In the issue id 661 [11], a test is failing because a subtracting NumPy array from torch tensors is not possible with `Torch 1.3`. To fix the problem, a developer has suggested using a minimal version of `Torch 1.3`. Even though the solution was simple, to understand, locate and fix the test, it took developers 4 days.
3. **Data type and structure**: the example 15illustrates a failure in the *sdg* gate when calling the compile function. This bug was caused by a bad structure used in the definition of the *sdg gate*, a simple change was to pass a list of qubits into the gate instead of one qubit as shown in the fix report15.

In a summary, bugs in quantum compilers seem to be common and are challenging to fix. Hence, developers can benefit from debugging tools and more test practices.

Simulator (SM): Since quantum computers are still not conveniently accessible by the public, simulators are widely used by quantum developers. We identify 10 bugs spread across the `quantum programming framework`, `experimental quantum computing`, `quantum circuit simulators` and `quantum utility tools`. The bugs are characterized mostly by `missing Error handling` and `performance bugs` with 2 bugs at each type. We present a bug example for each type.

1. **Performance bug** In `Qiskit` simulator, we report a bug that causes a huge performance regression in the simulation due to a large increase in serialization overhead for loading noise models from Python into C++. Even though the bug fix took only one line of code, developers spent 4 days locating and fixing it. Quantum developer has to rely on quantum simulators, therefore, we have to ensure good performance. Creating a performance benchmark for the simulators can help the community to select the most appropriate simulator and encourage the providers to improve the simulator's performance.
2. **Missing error handling**: Improper error handling can lead to serious consequences for any system, and the quantum software ecosystems are

[11] https://github.com/PennyLaneAI/pennylane/issues/661

```
qc = QuantumCircuit(1, 1)
qc.x(0)
qc.x(0)
# reset function is ignored when the qubit is in the groud state (
    Triiger the bug)
qc.reset(0)
```

(a) Trigger of the bug "qiskit-terra (issue id 5736 and 5127)"

```
# cannot compose this gate and raise an error.
if obj.definition is None:
    raise QiskitError('Cannot apply Instruction: {}'.format(obj.
    name))
if not isinstance(obj.definition, QuantumCircuit):
    raise QiskitError('Instruction "{}" '
QiskitError: 'Cannot apply Instruction: reset'
```

(b) Error message

```
#Repository: Qiskit/qiskit-terra
#Fix file: qiskit/dagcircuit/dagcircuit.py
#Class: DAGCircuit, method:_collect_1q_runs, line:1444
-len(s[0].cargs) == 0 and \
+not s[0].cargs and \
not s[0].op.is_parameterized() and \
-isinstance(node.op, Gate):
+isinstance(s[0].op, Gate):
```

(c) Bug fix

Listing 14: The execution of the code snippet (shown in (a)) triggers the error message (shown in (b)) because when a qubit is already is in ground state the reset function (line 5 in code snippet (a)) is being ignored. The fix was controlling the instance of the gate and properties (as shown in (c))

not an exception. In `qutip`, we detect the absence of error handling in the simulator which has to lead to the bug report id 396 [12]. The problem is when a developer tries to create a device object the application crashed if there is an error in the object attributes or methods. A developer has proposed to add an exception block as shown in the code17. The fix of the bug was quick (1 day) and not much code change was required. Improper error handling can be costly and lead to data leaks and many other exploits in the code. Therefore, developers must be careful when, where, and how to correctly handle errors in the code. Code analysis tools for error handling and logging recommendation can support the quantum program development and help avoid those bugs.

```
#Repository: m-labs/artiq
#Fix file: artiq/master/worker_db.py
#Class: DeviceManager, method: get, line: 144
-dev = _create_device(self.get_desc(name), self)
+try:
```

[12] https://github.com/m-labs/artiq/issues/396

```
backend = StatevectorSimulator()
q = QuantumRegister(1)
qc = QuantumCircuit(q)
# Calling the compile wih the sgd gate causes the code to crash (
    bug trigger)
qc.sdg(q)
qobj = compile(qc, backend, basis_gates=['u1', 'u2', 'u3'])
```

(a) Trigger of the bug "qiskit compiler (issue Id 2068)"

```
DAGCircuitError: 'not a QuantumRegister instance.
```

(b) Error message

```
#Repository:Qiskit/qiskit-terra
#Fix file:  qiskit/extensions/standard/s.py
#Class: SdgGate, method:_define, line:61
rule = [
-            (U1Gate(-pi/2), q[0], [])
+            (U1Gate(-pi/2), [q[0]], [])
        ]
```

(c) Bug fix

Listing 15: The execution of the code snippet (shown in (a)) triggers an error message (shown in (b)) because the sbg gate definition causes the compiler to crash. The bug fix was passing a list of qubit to the gate definition rule instead of one qubit (as shown in (c)).

```
#qiskit/qiskit-aer
# Fix file:  qiskit/providers/aer/noise/errors/quantum_error.py,
    method: to_dict, line: 462
-instructions = [exp.to_dict()['instructions'] for exp in qobj.
    experiments]
+instructions: [[op[0].assemble().to_dict() for op in circ.data]
                            for circ in self._circs]
```

Listing 16: Calling qiskit.assemble causes a huge performance regression in noisy simulator "qiskit-aer (issue id 1398)"

```
+    desc = self.get_desc(name)
+except Exception as e:
+    raise DeviceError("Failed to get description of device
    '{}'".format(name)) from e
+try:
+    dev = _create_device(desc, self)
+except Exception as e:
```

```
12 +     raise DeviceError("Failed to create device '{}'".format(
       name)) from e
```

Listing 17: Calling the device description causes the program to crash with inappropriate error message hard to understand.The bug fix was adding exception block in the get method with descriptive error message "artiq (issue id 396)".

Monitoring (MN) From table 1, Only 2 bugs were assigned to the monitoring component in which they appear in two quantum software categories. For example, the bug id 100 [13] in `quisp` project describes an overlogging problem in the code in the wrong place which makes debugging difficult because of the overloaded text. Even though, logging is a known practice in software engineering, the developer has to know when and where to log. Tools for logging and log level recommendations can help the community grow and improve the quality of the application.

Quantum cloud access (QCA): Cloud service providers like Amazon, Microsoft, and IBM give hand to developers to access quantum computers. We have detected 25 bugs in which we find `configuration bugs` are the most occurring with 8 bugs and `program anomaly bugs` are the second most appear bugs with 5 bugs as shown in table2. We present examples of the most appeared bug types.

1. **Configuration bugs**: The bug example 18happens in `DWave cloud access service` when trying to list the list of solvers. However, the list of solvers is filtered by the client type. To locate and fix the bug, it took some effort from developers since developers needed 14 days to close the issue [14]. Finally, a developer has proposed a solution as shown in the code18.

```
1  #Repository:dwavesystems/dwave-cloud-client
2  #Fix file: dwave/cloud/cli.py, method: solvers, line:346
3  +@click.option('--client', 'client_type', default=None,
4                 type=click.Choice(['base', 'qpu', 'sw', 'hybrid'
       ], case_sensitive=False),
5                 help='Client type used (default: from config)')
6  -def solvers(config_file, profile, solver_def, list_solvers,
       list_all):
7  +def solvers(config_file, profile, client_type, solver_def,
       list_solvers, list_all):
8      if list_all:
9  +        client_type = 'base'
10     with Client.from_config(
11 -            config_file=config_file, profile=profile, solver=
       solver_def) as client:
12 +            config_file=config_file, profile=profile,
13 +            client=client_type, solver=solver_def) as client:
```

Listing 18: Dwave solver is sensitive to the client type and does not return all the solvers with option -all because developers missed the client type argument in the solvers function. The bug fix was passing the client type as argument "dwave-cloud-client(issue id 457)

[13] https://github.com/sfc-aqua/quisp/issues/100

[14] https://github.com/dwavesystems/dwave-cloud-client/issues/457

References

1. Sodhi B (2018) Quality attributes on quantum computing platforms. CoRR abs/1803.07407, URL http://arxiv.org/abs/1803.07407, 1803.07407