

The Road to Reliable Quantum Software

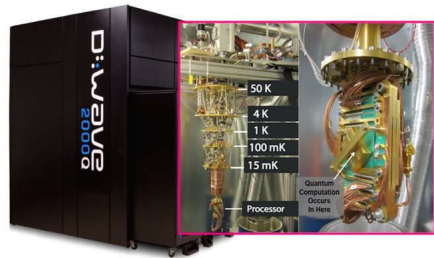
Jianjun Zhao

Kyushu University, Japan

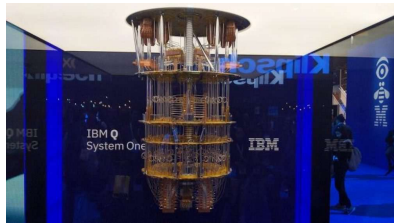
Mini-workshop (2026.09.29)

The promise of quantum computing

- Quantum computing has the potential to solve problems beyond the practical reach of classical computing.



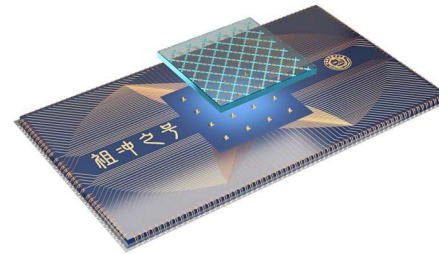
D-Wave



IBM



Google



USTC



Riken

The promise of quantum computing

- Quantum computing has the potential to solve problems beyond the practical reach of classical computing.
- Promising applications are being explored in chemistry, optimization, and machine learning.

Supply Chain Optimization

- Forecast and optimize for future inventory demand
- NP-hard scheduling map into quantum applications



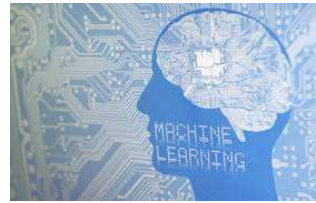
Machine Learning

- Development of new training sets and algorithms
- Classification and sampling of large data sets



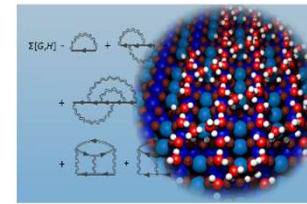
Robotic Manufacturing

- Reduce manufacturing time and cost
- Maps to a Traveling Salesman Problem addressable by quantum constrained optimization



Computational Material Science

- Design of better catalysts for batteries
- Quantum algorithms for calculating electronic structure



Alternative Energy Research

- Efficiently convert atmospheric CO₂ to methanol
- Powered by existing hybrid quantum-classical algorithms + machine learning

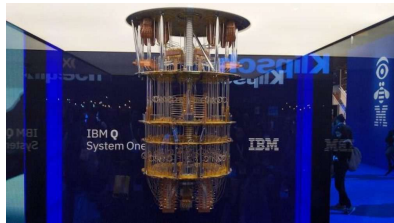


The promise of quantum computing

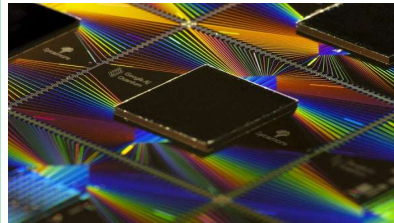
- Quantum computing has the potential to solve problems beyond the practical reach of classical computing.
- Promising applications are being explored in chemistry, optimization, and machine learning.
- Yet reliable quantum software remains a critical missing link.



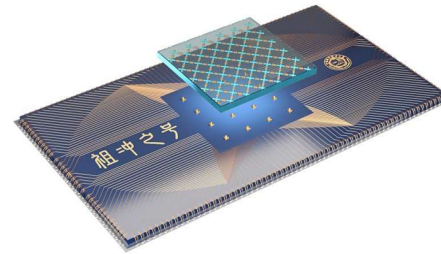
D-Wave



IBM



Google



USTC



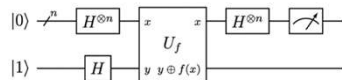
Riken

Programming quantum computers

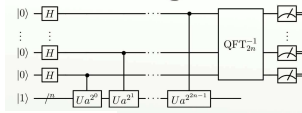
Quantum algorithms

Many quantum algorithms have been developed so far

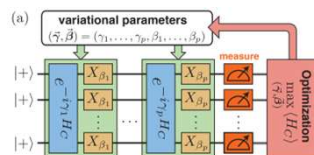
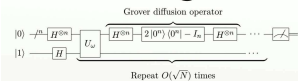
Deutsch-Jozsa algorithm



Shor's algorithm



Grover's algorithm



Quantum languages

A considerable number of quantum PLs have been developed

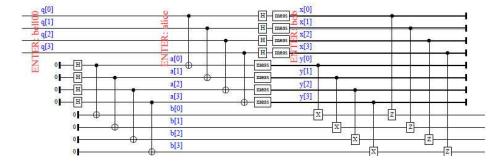


High-Level Quantum Programming

Quantum programs

Many companies are involved in developing quantum software applications

```
teleport_generic_labeled :: (QData qa) => qa -> Circ qa
teleport_generic_labeled q = do
  comment_with_label "ENTER: bell00" q "q"
  (a,b) <- bell00_generic (qc_false q)
  comment_with_label "ENTER: alice" (a,b) ("a","b")
  (x,y) <- alice_generic q a
  comment_with_label "ENTER: bob" (x,y) ("x","y")
  b <- bob_generic b (x,y)
  return b
```



Google AI
Quantum



HUAWEI

Why software reliability matters

- Quantum programs are complex and prone to subtle errors.
- Hardware noise can mask or compound software defects.
- Testing and debugging are difficult due to unobservable states.
- Reliable software is essential for trust and reproducibility.

Classical vs. quantum programs

Aspect	Classical Program	Quantum Program
State	Observable and stable	Hidden and collapses when measured
Execution	Typically deterministic, reproducible outcomes	Probabilistic, measurement-dependent outcomes
Testing	Direct comparison of outputs	Statistical validation through sampling
Debugging	Straightforward, step-by-step tracing	Direct step-by-step tracing is generally unavailable

Quantum programs introduce new sources of errors

Quantum indeterminacy

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

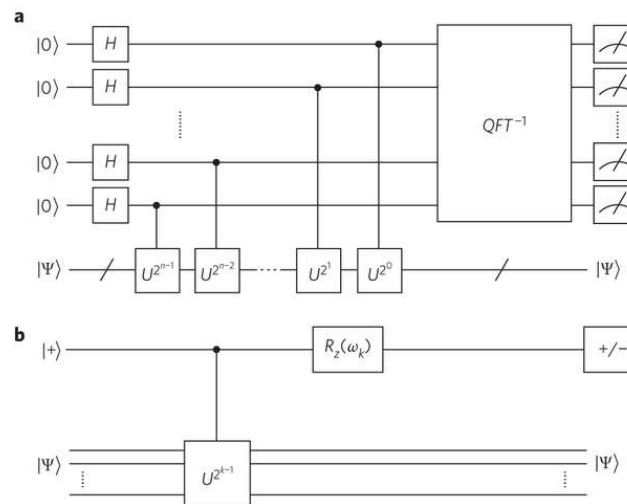
$$|\alpha|^2 + |\beta|^2 = 1$$

Probability of 0

Probability of 1

Quantum programs are probabilistic

Quantum programs are written as circuits



Quantum programs use new primitives

“prepare a uniform superposition”
“perform a Fourier transform”

```
simulator = Aer.get_backend('qasm_simulator')

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.h(2)

circuit.cx(0, 1)

circuit.measure([0,1,2], [0,1,2])

job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

Quantum programs introduce new sources of errors

Quantum indeterminacy

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Quantum programs are written as circuits



Quantum programs use new primitives

“prepare a uniform superposition”

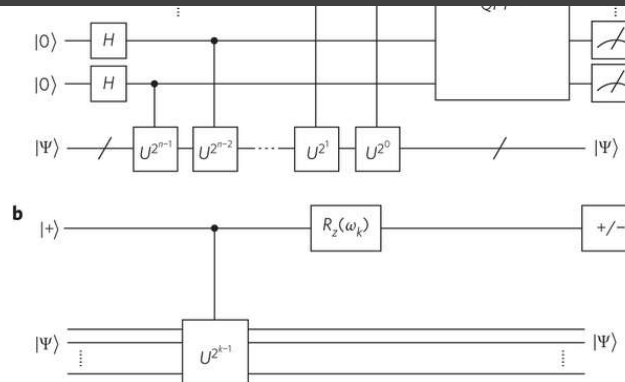
How to assure the reliability of the quantum software?

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

Probability of 0

Probability of 1

Quantum programs are probabilistic



```
simulator = Aer.get_backend('qasm_simulator')

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.h(2)

circuit.cx(0, 1)

circuit.measure([0,1,2], [0,1,2])

job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

Key challenges in reliability for quantum software

- **Unobservability:** Measuring qubits collapses their states.
- **Noise and instability:** Hardware errors distort execution results.
- **Complex semantics:** Entanglement creates non-local dependencies among qubits.
- **Hybrid structure:** Mixing classical and quantum code increases complexity.
- **Limited tooling:** Few mature testing or debugging tools exist today.

Bridging the gap

- New methods are needed for testing, debugging, and repairing.
- Our longer-term goal is to connect these building blocks into a broader QA framework for quantum software.
- Our journey begins with **understanding**, then **detection**, then **repair**.

Towards building reliable quantum software

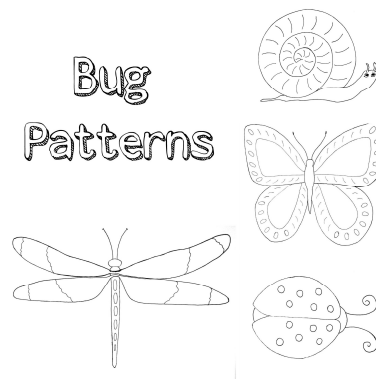
- **Bugs4Q:** What bug characteristics appear most often in quantum programs?
- **QChecker:** Can we detect these bugs via static analysis?
- **ChatGPT-based repair:** Can LLMs (ChatGPT) repair real quantum bugs effectively?

Bugs4Q:

What bug characteristics appear most often in quantum programs?

Bugs4Q

Bug patterns and bug
benchmark suite for
quantum programs

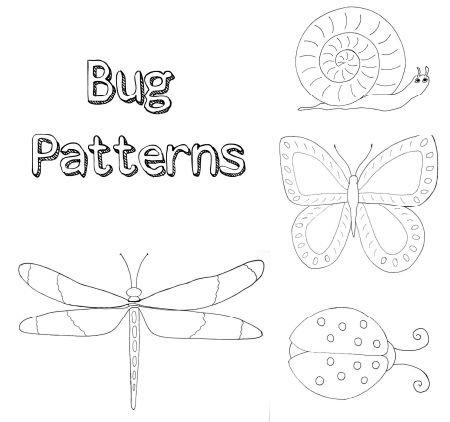


- P. Zhao, J. Zhao, and L. Ma. "Identifying bug patterns in quantum programs." Q-SE 2021. May 2021.
- P. Zhao, J. Zhao, Z. Miao, S. Lan. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs" ASE 2021, November 2021.

Bugs and bug patterns

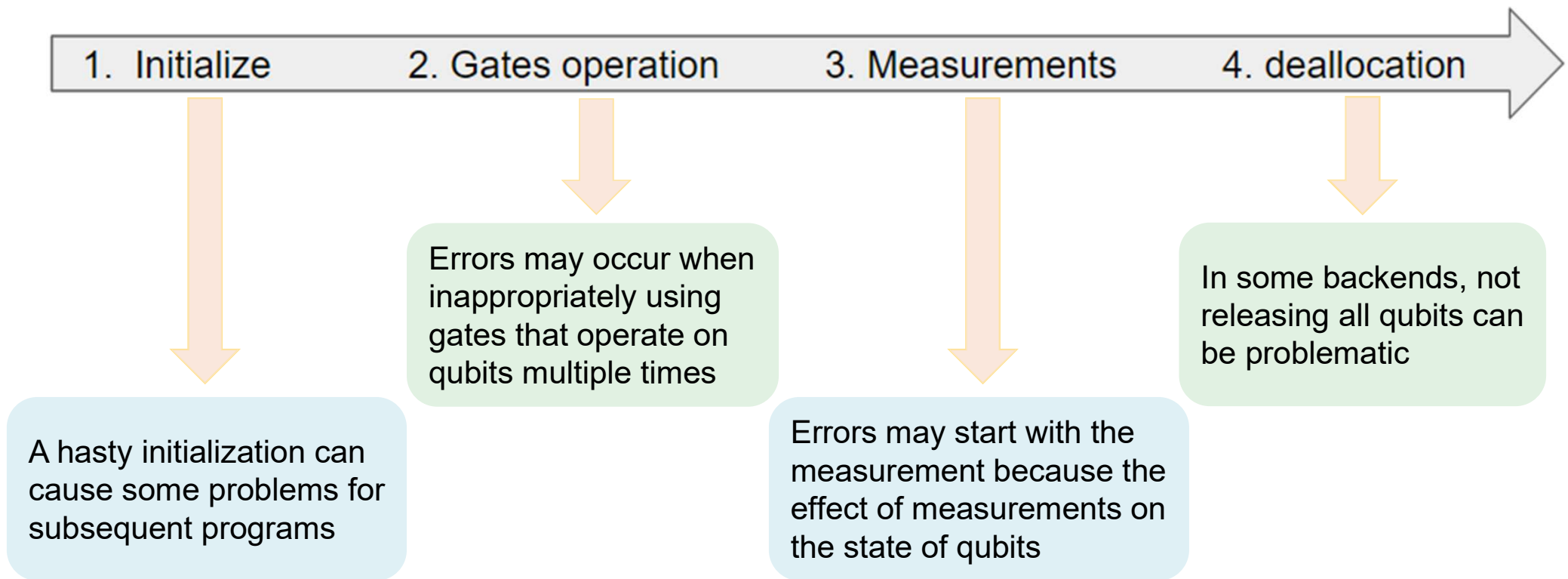
A **bug** is a defect in a program that may cause its behavior to deviate from the intended behavior or specification.

Bug patterns are recurring coding patterns or practices that are frequently associated with defects.



- ❑ Identifying bug patterns in programs can help
 - **Programmers** to improve their productivity in detecting bugs
 - **Language designers** to develop efficient bug detection tools

Error-prone features in quantum programs



Bug patterns in quantum programs

Classifying bug patterns helps identify unique quantum bugs

Developers may know what type of bugs are most likely to happen in a quantum program, and how to prevent them

Testers will have sufficient knowledge of how to write adequacy test cases that can effectively cover the most common potential errors

Maintenance staff will know which language features are more likely to result in the incorrect code

The information on bug patterns also provide a basis for further research on debugging and testing quantum programs

A bug pattern example

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)

teleport = QuantumCircuit(tq, tc0, tc1, tc2)
teleport.h(tq[1])
teleport.cx(tq[1], tq[2])

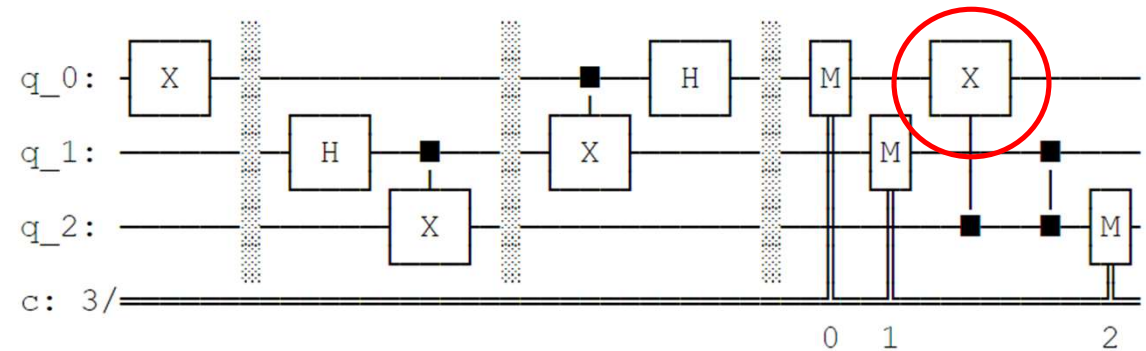
teleport.ry(np.pi/4, tq[0])

teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()

teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])

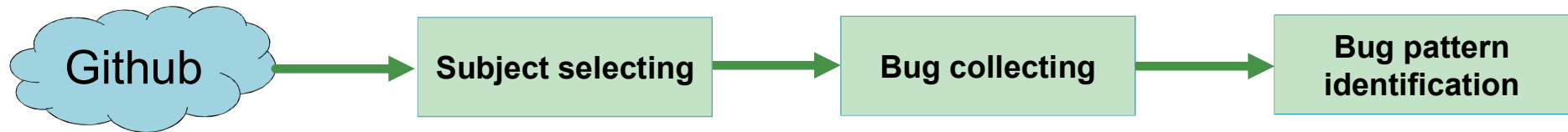
teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend('qasm_simulator')
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
print(job.get_memory()[0])
```



Incorrect operations after measurement

Process of identifying bug patterns in quantum programs



Collecting results

Repository	Number of bugs	Bugs with fixes
Qiskit	138	71
Cirq	25	16
Q#	21	2
ProjectQ	3	2
Total	187	91

Partial classification of Bug patterns (Qiskit)

Bug Patterns	Description	Cases & Examples
IG	Incorrect uses of quantum gates	Gates are not among the backend's basis gates
MI	Measurement related issue	Ignoring the effects of measurement
IS	Incorrect initial state	Number of qubits larger than the registers defined
PE	Parameters error	Using classical bits for entanglement
CM	Command misuse	Command misuse leads to unrecognized parameters
CE	Call error	Translating error
QE	QASM error	Issue with new from QASM str() method
DO	Discarded orders	Method has been deprecated

<https://github.com/Z-928/Bugs4Q>

What's the next step after having bug patterns at hand?

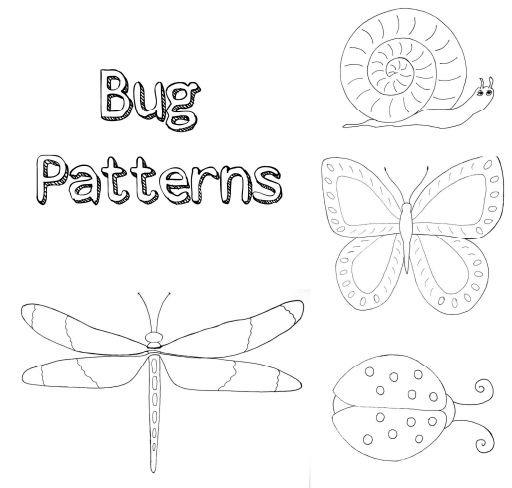
We can do two things at least:

A **bug benchmark suite** for quantum software to enable controlled testing and debugging studies for quantum programs.

Previous work on bug patterns provides a foundation for developing such a benchmark suite for quantum programs.

A **static analysis tool** for detecting various types of bugs in quantum software

Based on bug patterns identified in previous work



Bug benchmarks

A **bug benchmark suite** is a standard that consists of reproducible bugs and corresponding test cases to support the measurement or evaluation of bug-detecting tools

Research progress in software development is closely dependent on **high-quality bug benchmarks**, which can be used for:

- detecting, locating, and fixing bugs
- estimating the validity of debugging and testing tools

Without a unified benchmark suite, it is difficult to determine suitable debugging or testing methods for quantum software, as there is no basis for developing and validating bug detection tools.

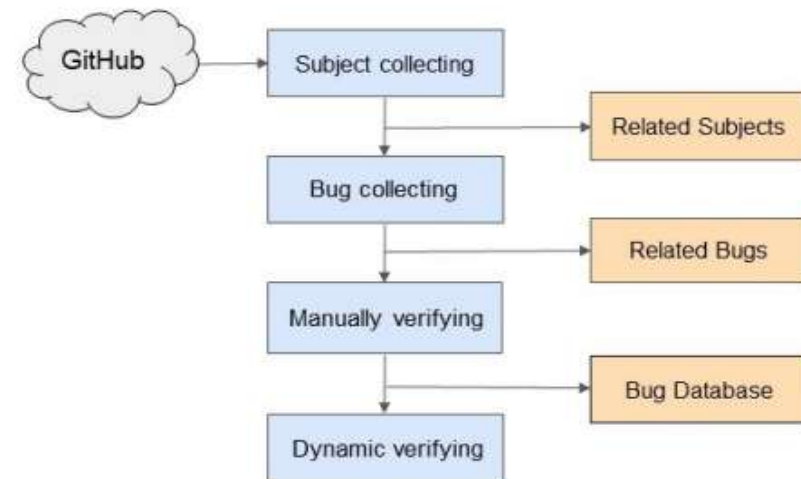
Bugs4Q: A Benchmark of Real Bugs for Quantum Programs

High-quality bug benchmarks are essential for quantum software testing and debugging:

- Effectiveness evaluation of debugging and testing tools
- Quantum bug finding and fixing

Bugs4Q

- Manually verified bug benchmarks of actual quantum programs in Qiskit
- Collect and validate reproducible real-world bugs in Qiskit programs
- Support downloading and executing test cases for quantum program testing



Buggy version

```
circuit = QuantumCircuit(3)
circuit.cx(0, 2)

basis = ['u1', 'cx']
coupling_map = CouplingMap([(0, 1), (1, 2)])
result = transpile(circuit,
                   optimization_level=1,
                   basis_gates=basis,
                   coupling_map=coupling_map,
                   seed_transpiler=42)

print(result)
```

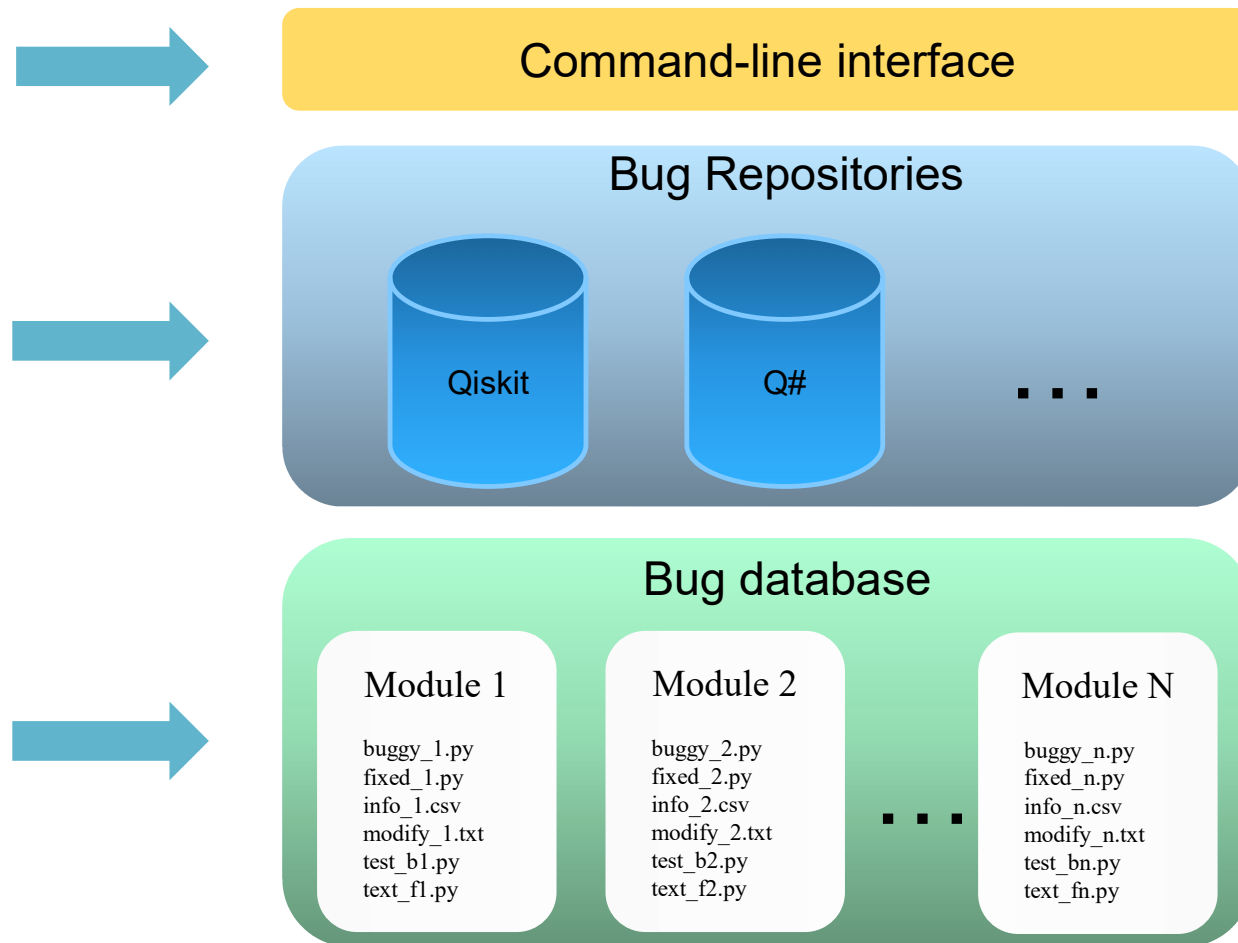
Fixed version

```
circuit = QuantumCircuit(3)
circuit.cx(0, 2)

basis = ['u1', 'cx']
coupling_map = CouplingMap([(0, 1), (1, 2)])
result = transpile(circuit,
                   optimization_level=1,
                   basis_gates=basis,
                   coupling_map,
                   seed_transpiler)

print(result)
```

The structure of Bugs4Q framework



Info command

- `python main.py info -h/help`
- `python main.py info -r qiskit -b 1`

Checkout command

- `python main.py checkout -h/help`
- `python main.py checkout -r qiskit -b 1 -v buggy -o out`

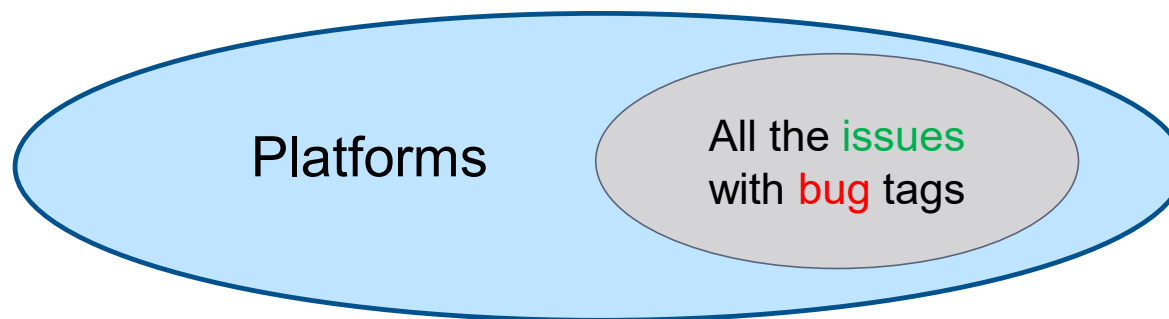
Run command

- `python main.py run -h/help`
- `python main.py run -r qiskit -b 1 -v buggy`

Test command

- `python main.py test -h/help`
- `python main.py test -r qiskit b 1 -v buggy`

Statistics of collection



The number of reproducible real-world bugs in Bugs4Q

Platform	Source	Objectives	Number of items
Github	IBM Qiskit	Issues in Qiskit-related repositories	4621
StackOverflow	IBM Qiskit	Questions tagged with Qiskit	465
Stack Exchange	IBM Qiskit	Questions with “Qiskit” as keyword	4984
Total	IBM Qiskit	Issues and questions	10069

Statistics of validation

Description		Counts
Initial number of bugs with source code		306
Manual validation	Platforms bug	206
	Incomplete source code	35
	No fix code	27
After manual validation		84
Dynamic validation	Fixes not work	4
	Bug not as described	12
	Source code can not run	21
	The buggy version runs smoothly	5
After dynamic validation		42
Unit test	Can not catch exceptions	3
	Output is matplotlib diagrams	4
	No output	2
	Output too complex	3
Final number		30

Statistics of validation

Description		Counts
Initial number of bugs with source code		306
Manual validation	Platforms bug	206
	Incomplete source code	35
	No fix code	27
After manual validation		84
Dynamic validation	Fixes not work	4
	Bug not as described	12
	Source code can not run	21
	The buggy version runs smoothly	5
After dynamic validation		42
Unit test	Can not catch exceptions	3
	Output is matplotlib diagrams	4
	No output	2
	Output too complex	3
Final number		30

Statistics of validation

Description		Counts
Initial number of bugs with source code		306
Manual validation	Platforms bug	206
	Incomplete source code	35
	No fix code	27
After manual validation		84
Dynamic validation	Fixes not work	4
	Bug not as described	12
	Source code can not run	21
	The buggy version runs smoothly	5
After dynamic validation		42
Unit test	Can not catch exceptions	3
	Output is matplotlib diagrams	4
	No output	2
	Output too complex	3
Final number		30

Statistics of validation

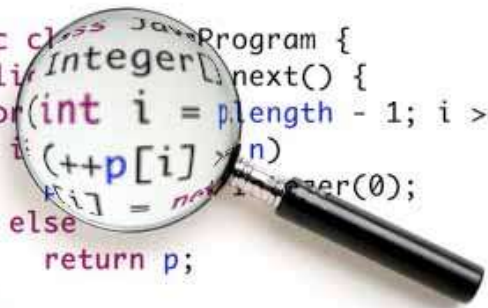
Description		Counts
Initial number of bugs with source code		306
Manual validation	Platforms bug	206
	Incomplete source code	35
	No fix code	27
After manual validation		84
Dynamic validation	Fixes not work	4
	Bug not as described	12
	Source code can not run	21
	The buggy version runs smoothly	5
After dynamic validation		42
Unit test	Can not catch exceptions	3
	Output is matplotlib diagrams	4
	No output	2
	Output too complex	3
Final number		30

QChecker:

Can we detect these bugs via static analysis?

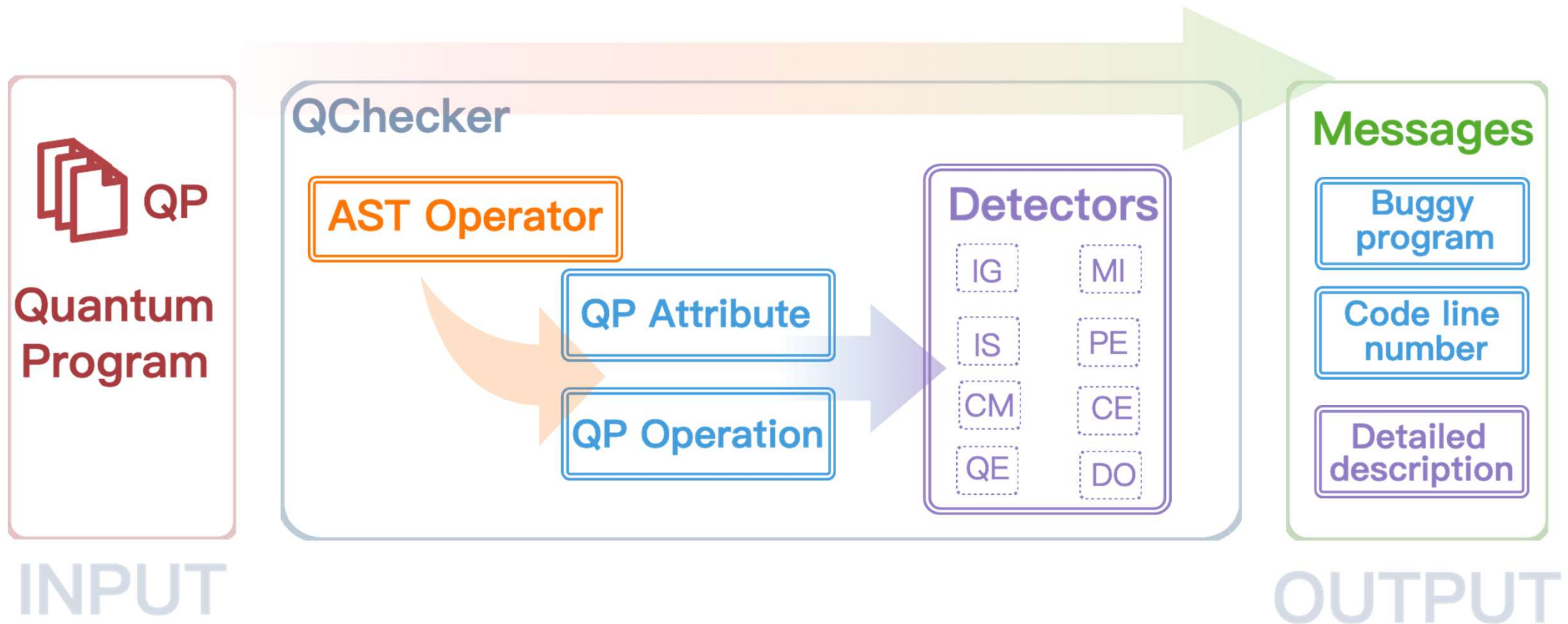
QChecker

Pattern-based bug
detector for quantum
programs

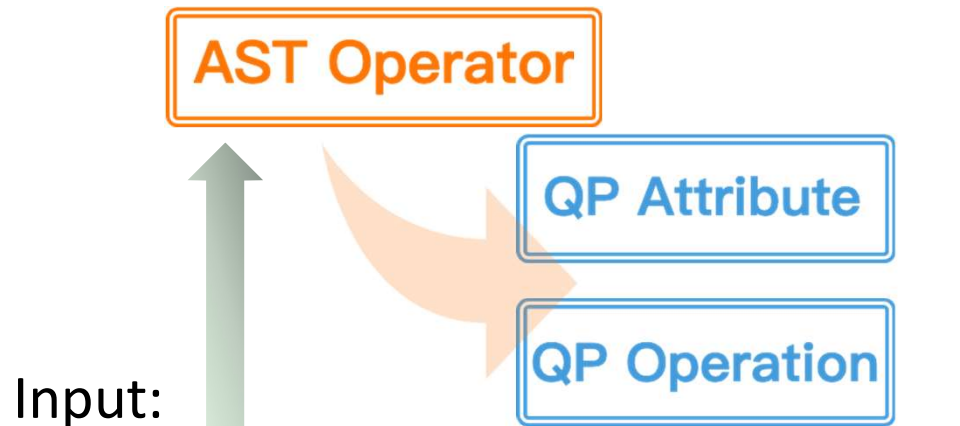


```
public class JavaProgram {  
    public Integer[] next() {  
        for (int i = p.length - 1; i >= 0; i--)  
            if (++p[i] > n)  
                p[i] = nextInteger(0);  
        else  
            return p;  
    }  
    throw new NoSuchElementException();  
}
```

Overview of QChecker



Structure of AST_Operator



```
simulator = Aer.get_backend("qasm_simulator")

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.h(2)
circuit.cx(0, 1)
circuit.measure([0,1,2], [0,1,2])
job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

```
('simulator', 'Aer.get_backend("qasm_simulator")')
('qreg', 'QuantumRegister(3)')
('creg', 'ClassicalRegister(3)')
('circuit', 'QuantumCircuit(qreg,creg)')
('job', 'execute(circuit,simulator,shots=1000)')
('result', 'job.result()')
('counts', 'result.get_counts(circuit)')
```

```
'Aer.get_backend("qasm_simulator")'
'QuantumRegister(3)'
'ClassicalRegister(3)'
'QuantumCircuit(qreg,creg)'
'circuit.h(0)'
'circuit.h(2)'
'circuit.cx(0,1)'
'circuit.measure([0,1,2],[0,1,2])'
'execute(circuit,simulator,shots=1000)'
'job.result()'
'result.get_counts(circuit)'
'print(counts)'
```

Detectors

Name	Description	Identified bug patterns
IG	Incorrect use of quantum gates	Gates are not among the backends' basis gates Handle custom multi-qubit gates Random gate is not defined
MI	Measurement related issue	Ignoring the effects of measurement
IS	Incorrect initial state	Number of qubits larger than the registers defined The insufficient number of qubits Insufficient length of classical registers
PE	Parameters error	Instruction not in basis gates Incorrect parameters in gates Using classical bits for entanglement
CM	Command misuse	Unrecognized parameters Quantum circuit interaction error Create redundant classical registers
CE	Call error	Object call error Backend error Translating error
QE	QASM error	Issue with new from qasm str() method
DO	Discarded orders	Method has been deprecated

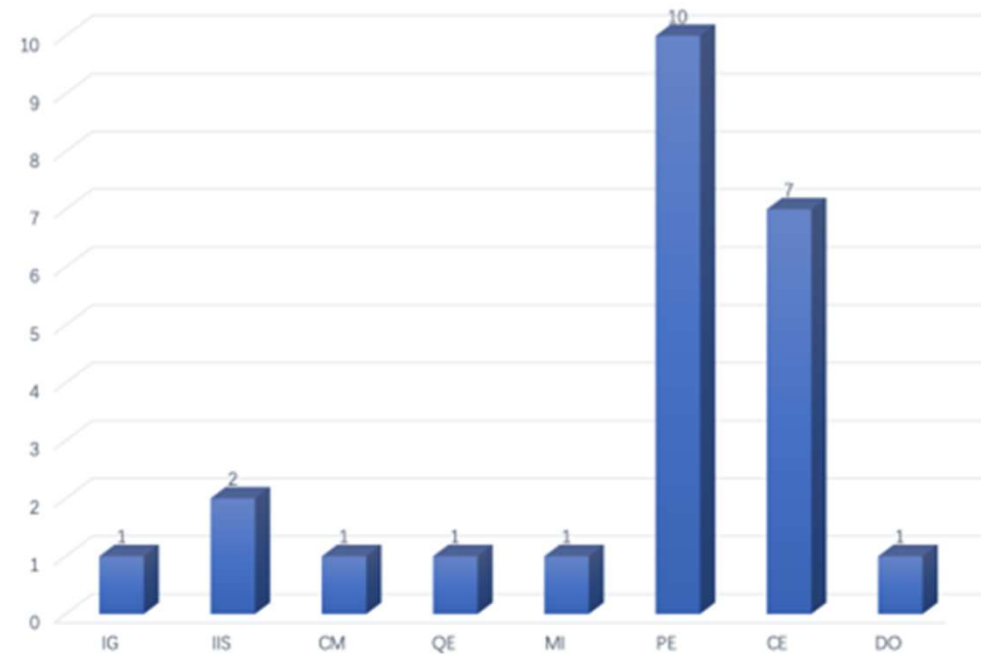
Evaluation of QChecker

Database:

Bugs4Q benchmark suite:
42 real-world buggy
programs written in Qiskit

Environment:

- Intel i9-10940X CPU
- 128G RAM, Ubuntu 20.04
- Python 3.10

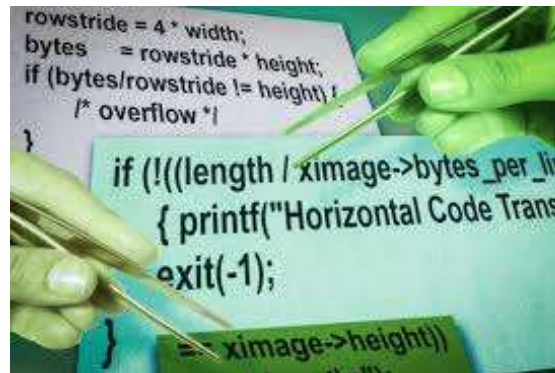


Distribution of bugs found by each detector

QChecker detected 24 of the 42 Bugs4Q bugs in our evaluation.

Can LLMs repair real quantum bugs effectively?

Repairing Quantum Programs using ChatGPT



X. Guo, J. Zhao, P. Zhao. On Repairing Quantum Programs Using ChatGPT. In Proc. 5th International Workshop on Quantum Software Engineering (Q-SE 2024), April 2024.

Quantum software testing and debugging

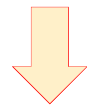
Testing approaches	Reference
Fuzz testing	Wang et al. [2018]
Testing criteria	Ali et al. [2021] , Long and Zhao [2023]
Property-based testing	Honarvar et al. [Q-SE 2020] Li et al. [OOPSLA 2020]
Search-based testing	Wang et al. [2022]
Combinatorial Testing	Wang et al. [QRS 2021]
Mutation testing	Mendiluze et al. [2021] Fortunato et al. [2022]
Metamorphic testing	Abreu et al. [Q-SE2022]
Testing multi-subroutine quantum programs	Long and Zhao [2022, 2023]

“Testing can be used to show the presence of bugs, but never to show their absence.”
- E. W. Dijkstra

Debugging approaches	Reference
Debugging tactics	Miranskyy et al. [2020]
Assertion-based debugging	Huang et al. [2020] , Li et al. [2021]
Delta Debugging	Pontolillo et al. [Q-SE 2024]
Debugging quantum circuits	Metwalli and Meter [2022]

We want to go further beyond testing and debugging

Many techniques developed for repairing classical programs
(traditional / LLM-based approach)



Can we apply these techniques to repair quantum programs?



How ChatGPT can help to generate repair patches for buggy quantum programs

Research questions

RQ1

Can ChatGPT effectively repair bugs in quantum programs?

Repair efficiency

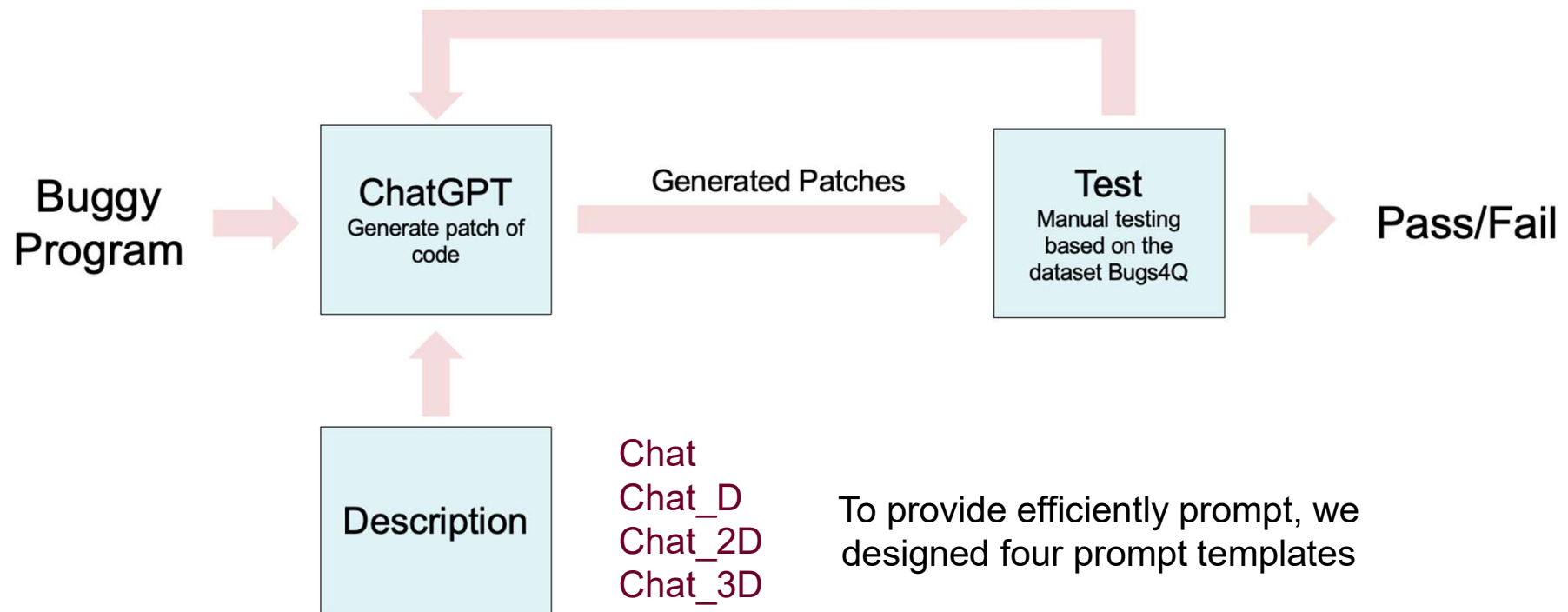
RQ2

What is the performance of ChatGPT when applied to real-world quantum program bugs?

Practical utility

Workflow

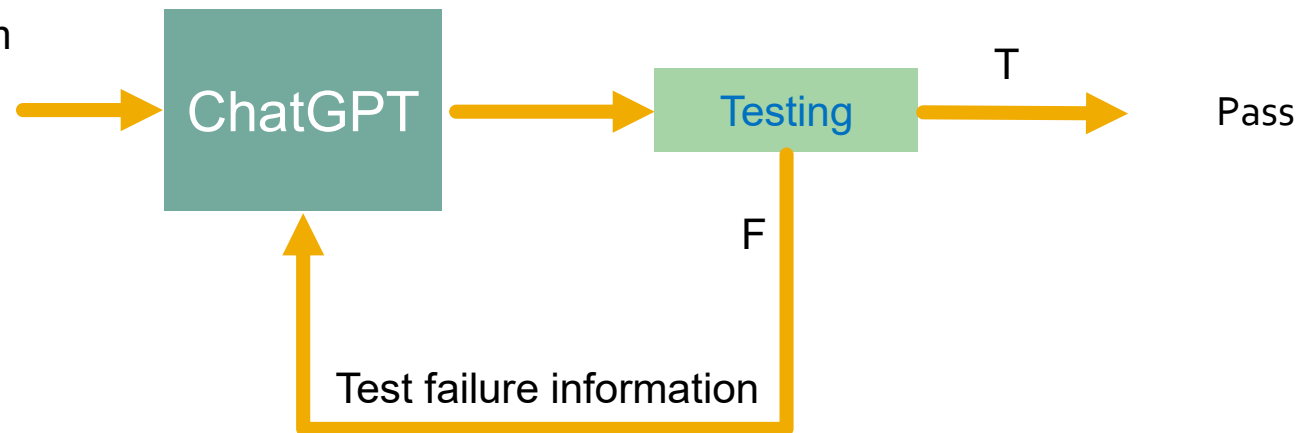
The preparation requires prompt to utilize ChatGPT conversational function



Templates: Chat | Chat_D | Chat_2D | Chat_3D

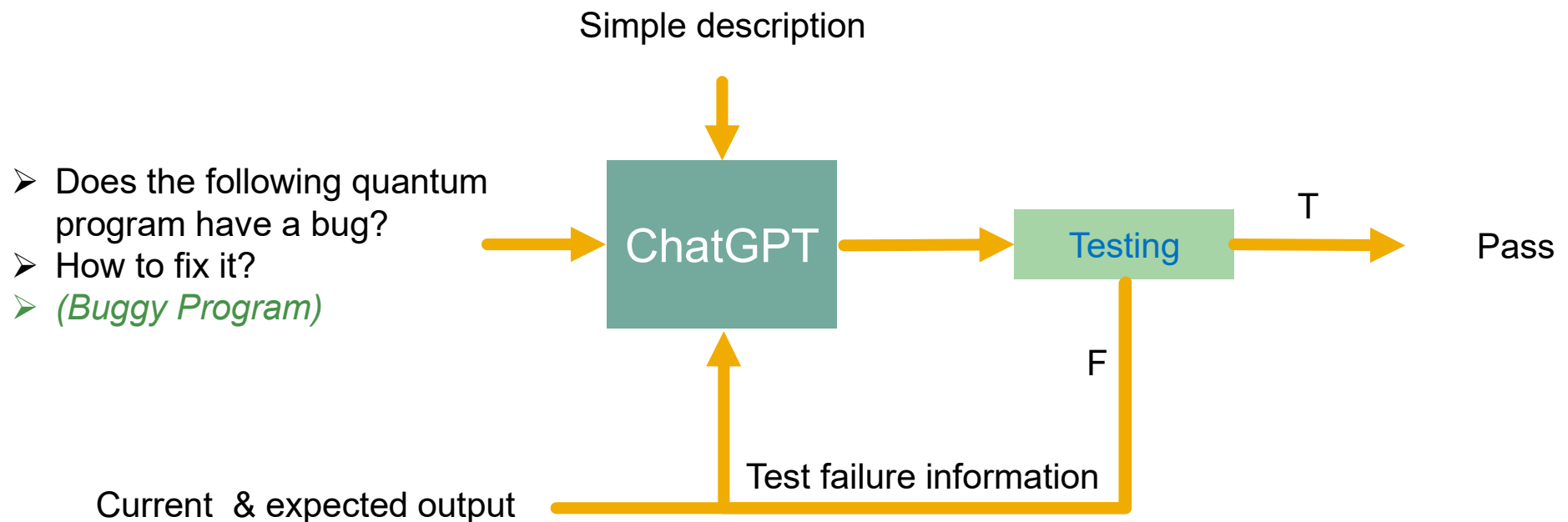
Chat: indicates the exclusive use of basic ChatGPT

- Does the following quantum program have a bug?
- How to fix it?
- (*Buggy Program*)



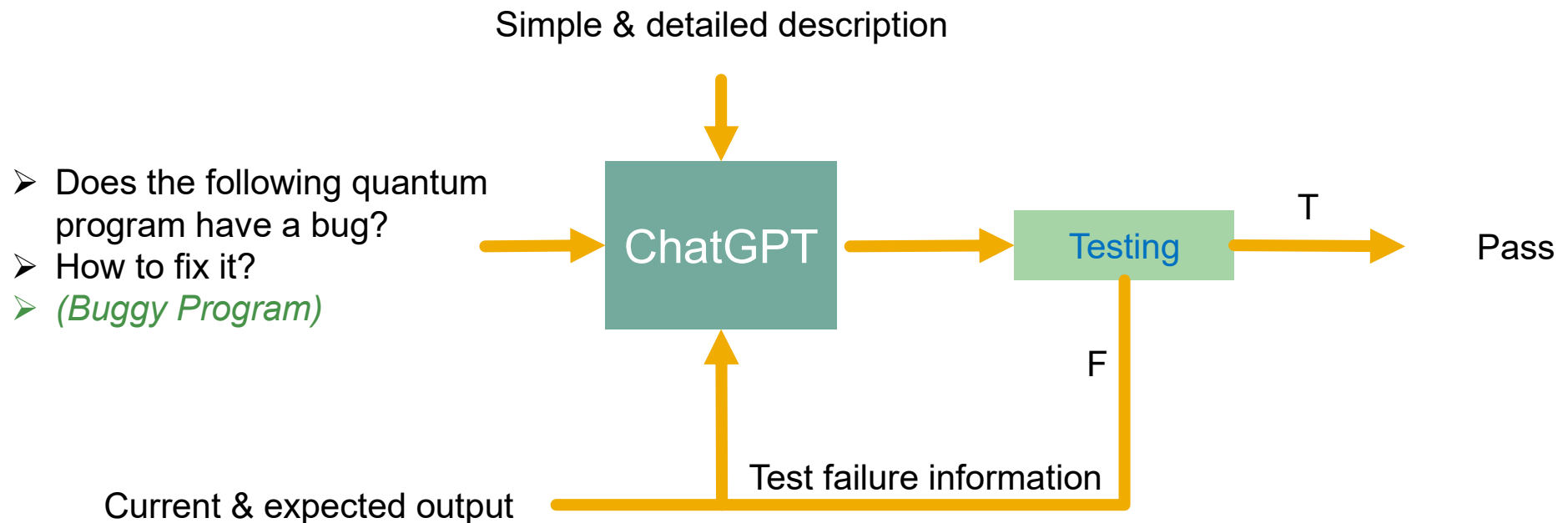
Templates: Chat | Chat_D | Chat_2D | Chat_3D

Chat_D: utilizes ChatGPT in conjunction with a basic quantum buggy program description



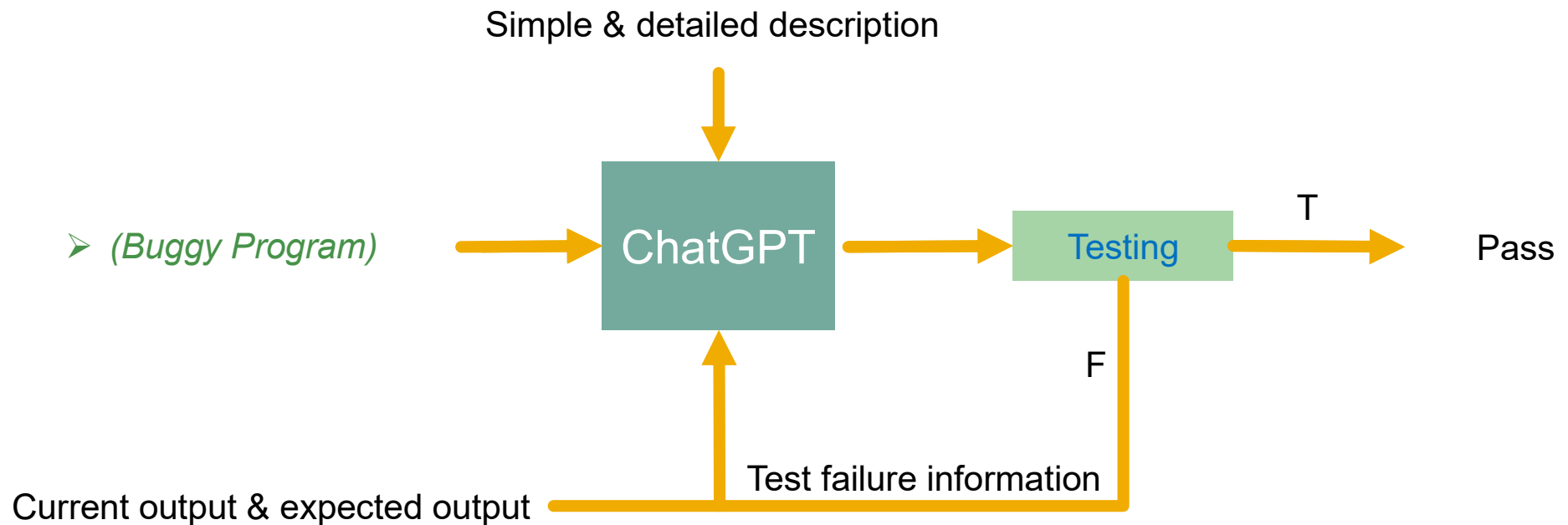
Templates: Chat | Chat_D | Chat_2D | Chat_3D

Chat_2D: builds upon Chat_D to offer a more detailed description



Templates: Chat | Chat_D | Chat_2D | Chat_3D

Chat_3D: directly identifies the location of bugs



Description generation

Simple Description

qiskit.quantum_info.operators.Operator will give wrong result in the following condition #5580

 Closed yanhuchen opened this issue on Dec 30, 2020 · 2 comments

Detailed Description

What is the current behavior?

For the same unitary operation, we attempt to use two ways to implement it.

1. Use the existing unitary gates of qiskit to construct a quantum circuit for denoting the unitary matrix.
2. Use the qiskit.quantum_info.operators.Operator to turn the unitary matrix directly into a quantum circuit.

I found the two ways have different results. I attach the code to reproduce the problem.



What is the expected behavior?

According to the above code, they want to implement

```
qc.crz(pi/2,qae[0],reg_b[0])
```

```
qc.crz(pi/4,qae[0],reg_b[1])
```

But the two ways give different unitary matrices based on their quantum circuit. The result is obviously wrong.

Note that, the problem may only appear in the qiskit.quantum_info.operators.Operator that turn a unitary matrix directly into a quantum circuit and the unitary matrix not act on all qubits. I provide the example shows the unitary matrix act on 3 qubits, but the complete circuit is 4 qubits.

<https://github.com/Qiskit/qiskit/issues/5580>

Experimental Evaluation

Bug benchmark: Bugs4Q

A benchmark of real, manually validated bugs for Qiskit programs

Buggy version

```
circuit = QuantumCircuit(3)
circuit.cx(0, 2)

basis = ['u1', 'cx']
coupling_map = CouplingMap([(0, 1), (1, 2)])
result = transpile(circuit,
                    optimization_level=1,
                    basis_gates=basis,
                    coupling_map=coupling_map,
                    seed_transpiler=42)

print(result)
```

Fixed version

```
circuit = QuantumCircuit(3)
circuit.cx(0, 2)

basis = ['u1', 'cx']
coupling_map = CouplingMap([(0, 1), (1, 2)])
result = transpile(circuit,
                    optimization_level=1,
                    basis_gates=basis,
                    coupling_map,
                    seed_transpiler)

print(result)
```

We use 38 reproducible buggy programs from Bugs4Q in this study.

Evaluation metrics

Correctness

TOP-5

$$\begin{cases} 1, & \text{One success in five attempts} \\ 0, & \text{Otherwise} \end{cases}$$

AVG-5

$$AVG - 5 = \frac{\text{correct generated patches}}{\text{five attempts}}$$

Evaluating the correctness of the patched code generated by ChatGPT for RQ1 and RQ2

Repair Rate

$$\text{Repair rate} = \frac{\text{correct generated patches}}{\text{all generated patches}}$$

Assessing ChatGPT's performance on these prompt templates

RQ1: Can ChatGPT effectively repair bugs in quantum programs?

We conduct the TOP-5 metric to evaluate the effectiveness of ChatGPT

BUG ID	BUG Type	ChatGPT	ChatGPT_1D	ChatGPT_2D	ChatGPT_3D
#6540	Parameter Error	✗(0/5)	✓(1/5)	✓(2/5)	✓(1/5)
#6255	QASM Calculation	✗(0/5)	✗(0/5)	✗(0/5)	✗(0/5)
#664	Measurement Error	✗(0/5)	✓(1/5)	✓(5/5)	✓(4/5)
#6892	Parameter Error	✗(0/5)	✗(0/5)	✓(1/5)	✓(2/5)
#5580	Circuit Design	✗(0/5)	✗(0/5)	✗(0/5)	✗(0/5)
#5098	Parameter Error	✗(0/5)	✓(3/5)	✓(3/5)	✓(5/5)
#302	Parameter Error	✓(5/5)	✓(5/5)	✓(5/5)	✓(5/5)
.....					
#9209	Parallel	✗(0/5)	✓(4/5)	✓(4/5)	✓(5/5)
#6697	Import Error	✗(0/5)	✓(1/5)	✓(3/5)	✓(5/5)
#5249	Wrong Function Name	✓(2/5)	✓(4/5)	✓(5/5)	✓(5/5)
#9224	Output Wrong	✓(1/5)	✓(2/5)	✓(4/5)	✓(5/5)
Total		10	23	29	29

ChatGPT can effectively repair quantum program bugs when providing detailed test failure information and descriptions

RQ2: What is the performance of ChatGPT when applied to real-world quantum program bugs?

We conduct the TOP-5 and AVG-5 metrics to evaluate the practical utility of ChatGPT

Bug Type	ChatGPT	ChatGPT_D	ChatGPT_2D	ChatGPT_3D
Simple Quantum Bug	6/25(0.1)	17/25(0.46)	21/25(0.66)	21/25(0.67)
Quantum Algorithm Bug	0/7(0.00)	0/7(0.00)	2/7(0.01)	2/7(0.17)
Classic Bug	4/6(0.43)	6/6(0.70)	6/6(0.87)	6/6(0.97)
Total	10/38(0.12)	23/38(0.41)	29/38(0.58)	29/38(0.64)

The experimental result shows that Chat_3D achieved the highest AVG-5 repair rate of 64%.

Conclusion

- Understanding, detection, and repair are three connected steps toward reliable quantum software.
- Insights from Bugs4Q support both bug detection and repair.
- **Long-term goal:** connect these building blocks into systematic quality assurance for quantum software.

The road ahead

- Extend QA to hybrid quantum–classical systems.
- Develop measurable quality metrics and coverage criteria.
- Move from research prototypes to production-ready tools.

Acknowledgements

We thank all the members of the research group on QPL and QSE at Kyushu University, especially:

- Pengzhan Zhao
- Xiaoyu Guo
- Shuhan Lan
- Zhongtao Miao
- Zhuo Li
- Xiongfei Wu

We also thank Prof. Lei Ma from the University of Tokyo.

Thank you for listening!

