MQT QCEC

A tool for Quantum Circuit Equivalence Checking

Version 3.0.0b2

Chair for Design Automation
Technical University of Munich

Mar 10, 2025

Abstract

MQT QCEC is an open-source C++17 and Python library for *quantum circuit equivalence checking* developed as part of the *Munich Quantum Toolkit (MQT)* [1] by the Chair for Design Automation at the Technical University of Munich.

This documentation provides a comprehensive guide to the MQT QCEC library, including *installation instructions*, a *quickstart guide*, and detailed *API documentation*. The source code of MQT QCEC is publicly available on GitHub at cda-tum/mqt-qcec, while pre-built binaries are available via PyPI for all major operating systems and all modern Python versions. MQT QCEC is fully compatible with Qiskit 1.0 and above.

> **Note**
>
> A live version of this document is available at mqt.readthedocs.io/projects/qcec.

Contents

I Installation

MQT QCEC is mainly developed as a C++17 library with Python bindings. The resulting Python package is available on PyPI and can be installed on all major operating systems using all modern Python versions.

Tip

We highly recommend using `uv` for working with Python projects. It is an extremely fast Python package and project manager, written in Rust and developed by Astral (the same team behind `ruff`). It can act as a drop-in replacement for `pip` and `virtualenv`, and provides a more modern and faster alternative to the traditional Python package management tools. It automatically handles the creation of virtual environments and the installation of packages, and is much faster than `pip`. Additionally, it can even set up Python for you if it is not installed yet.

If you do not have `uv` installed yet, you can install it via:

macOS and Linux

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows

```
$ powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Check out their excellent documentation for more information.

uv *(recommended)*

```
$ uv pip install mqt.qcec
```

pip

```
(.venv) $ python -m pip install mqt.qcec
```

In most practical cases (under 64-bit Linux, MacOS incl. Apple Silicon, and Windows), this requires no compilation and merely downloads and installs a platform-specific pre-built wheel.

Once installed, you can check if the installation was successful by running:

```
(.venv) $ python -c "import mqt.qcec; print(mqt.qcec.__version__)"
```

which should print the installed version of the library.

Attention

As of version 2.8.0, support for Python 3.8 has been officially dropped. We strongly recommend that users upgrade to a more recent version of Python to ensure compatibility and continue receiving updates and support. Thank you for your understanding.

I-A Building from source for performance

In order to get the best performance and enable platform-specific optimizations that cannot be enabled on portable wheels, it is recommended to build the library from source via:

uv *(recommended)*

```
$ uv pip install mqt.qcec --no-binary mqt.qcec --no-binary mqt.core
```

pip

```
(.venv) $ pip install mqt.qcec --no-binary mqt.qcec --no-binary mqt.core
```

This requires a C++ compiler supporting C++17 and a minimum CMake version of 3.19. The library is continuously tested under Linux, MacOS, and Windows using the latest available system versions for GitHub Actions. In order to access the latest build logs, visit the GitHub Actions page.

I-B Integrating MQT QCEC into your project

If you want to use the MQT QCEC Python package in your own project, you can simply add it as a dependency in your `pyproject.toml` or `setup.py` file. This will automatically install the MQT QCEC package when your project is installed.

uv *(recommended)*

```
$ uv add mqt.qcec
```

pyproject.toml

```
[project]
# ...
dependencies = ["mqt.qcec>=3.0.0"]
# ...
```

setup.py

```
from setuptools import setup

setup(
    # ...
    install_requires=["mqt.qcec>=3.0.0"],
    # ...
)
```

If you want to integrate the C++ library directly into your project, you can either

- add it as a git submodule and build it as part of your project, or
- use CMake's FetchContent module.

FetchContent

This is the recommended approach for projects because it allows to detect installed versions of MQT QCEC and only downloads the library if it is not available on the system. Furthermore, CMake's FetchContent module allows for lots of flexibility in how the library is integrated into the project.

```
include(FetchContent)
set(FETCH_PACKAGES "")

# cmake-format: off
set(MQT_QCEC_VERSION 3.0.0
    CACHE STRING "MQT QCEC version")
set(MQT_QCEC_REV "v3.0.0"
    CACHE STRING "MQT QCEC identifier (tag, branch or commit hash)")
set(MQT_QCEC_REPO_OWNER "cda-tum"
    CACHE STRING "MQT QCEC repository owner (change when using a fork)")
# cmake-format: on
if(CMAKE_VERSION VERSION_GREATER_EQUAL 3.24)
  FetchContent_Declare(
    mqt-qcec
    GIT_REPOSITORY https://github.com/${MQT_QCEC_REPO_OWNER}/mqt-qcec.git
    GIT_TAG ${MQT_QCEC_REV}
    FIND_PACKAGE_ARGS ${MQT_QCEC_VERSION})
  list(APPEND FETCH_PACKAGES mqt-qcec)
else()
  find_package(mqt-qcec ${MQT_QCEC_VERSION} QUIET)
  if(NOT mqt-qcec_FOUND)
    FetchContent_Declare(
      mqt-qcec
      GIT_REPOSITORY https://github.com/${MQT_QCEC_REPO_OWNER}/mqt-qcec.git
      GIT_TAG ${MQT_QCEC_REV})
    list(APPEND FETCH_PACKAGES mqt-qcec)
  endif()
endif()

# Make all declared dependencies available.
FetchContent_MakeAvailable(${FETCH_PACKAGES})
```

git submodule

Integrating the library as a git submodule is the simplest approach. However, handling git submodules can be cumbersome, especially when working with multiple branches or versions of the library. First, add the submodule to your project (e.g., in the `external` directory) via:

```
$ git submodule add https://github.com/cda-tum/mqt-qcec.git external/mqt-qcec
```

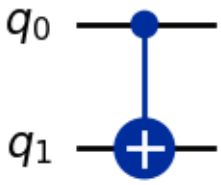
Then, add the following lines to your `CMakeLists.txt` to make the library's targets available in your project:

```
add_subdirectory(external/mqt-qcec)
```

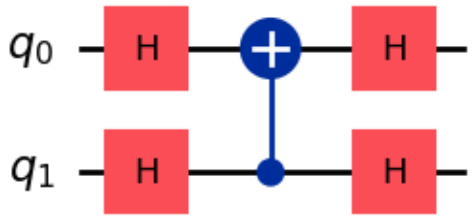
II Quickstart

Assume you want to prove that the following two circuits are equivalent:

```
1 from qiskit import QuantumCircuit
2
3 qc1 = QuantumCircuit(2)
4 qc1.cx(0, 1)
5 qc1.draw(output="mpl", style="iqp")
```



```python
from qiskit import QuantumCircuit

qc2 = QuantumCircuit(2)
qc2.h(0)
qc2.h(1)
qc2.cx(1, 0)
qc2.h(1)
qc2.h(0)
qc2.draw(output="mpl", style="iqp")
```



Then, using QCEC to check the equivalence of these two circuits is as easy as

```python
from mqt import qcec

qcec.verify(qc1, qc2)
```

```
<EquivalenceCheckingManager.Results: equivalent>
```

Check out the *reference documentation* for more information.

III Quantum Circuit Equivalence Checking

Consider a quantum circuit $G = g_0 \dots g_{|G|-1}$ acting on n qubits. Each gate g_i of G is described by a corresponding unitary matrix U_i that is subsequently applied during the execution of the quantum circuit. Thus, the functionality of the circuit G can be obtained as a unitary *system matrix* U itself by computing $U = U_{|G|-1} \dots U_0$.

The *equivalence checking problem for quantum circuits* can be defined as follows:

> Given two quantum circuits $G = g_0 \dots g_{|G|-1}$ and $G' = g'_0 \dots g'_{|G'|-1}$ acting on the same set of qubits, do they realize the same functionality? More specifically, does it hold that $U = e^{i\theta}U'$ or, equivalently $UU' = e^{i\theta}\mathbb{I}$, where $\theta \in (-\pi, \pi]$ denotes a physically unobservable global phase?

Conceptually, checking the equivalence of two quantum circuits is as simple as constructing and comparing the respective system matrices. However, the size of these matrices grows exponentially with the number of qubits, which quickly renders straight-forward methods (such as arrays) infeasible. Equivalence checking of quantum circuits has even been shown to be QMA-complete.

QCEC provides several complementary methods for efficiently tackling this challenging problem—each with their respective use cases, capabilities, and drawbacks.

III-A Construction Equivalence Checker (using Decision Diagrams)

While the underlying matrices are exponentially large with respect to the number of qubits, they frequently are sparse or contain inherent redundancies. Decision diagrams have been proposed as a means to exploit these redundancies. For a detailed intro to this topic, see [2] and the references therein. In many cases, using decision diagrams allows to obtain polynomially-sized representations for the otherwise exponentially large system matrices—in the best case even linear.

The functionality of both circuits is constructed by subsequently multiplying the decision diagrams representing the individual gates of G and G'—until, eventually, decision diagram representations for the system matrices U and U' are obtained. Since decision diagrams are canonical representations of the underlying matrices, checking the equivalence of both circuits can be reduced to comparing the root pointers of the resulting decision diagrams.

Most effective for: proving equivalence or non-equivalence of circuits whose functionality is sparse and/or contains redundancies.

Capable of showing: equivalence and non-equivalence

Drawback: Decision diagrams might still be exponentially large in the worst case.

III-B Alternating Equivalence Checker (using Decision Diagrams)

Due to the inherent reversibility of quantum computations, equivalence checking of two circuits G and G' can also be reduced to proving that the composition of one circuit with the inverse of the other implements the identity, i.e., $G'^{-1}G = \mathbb{I}$. While the identity is represented by a decision diagram of linear size, sequentially constructing the representation for $G'^{-1}G$ still constructs the (potentially exponentially large) representation of one of the circuit's system matrix along the way.

The main idea is to, instead, start with the identity $\mathbb{I}$ in between both circuits, which is symbolized by $G \rightarrow \mathbb{I} \leftarrow G'$. Then, gates from either circuit are applied in an alternating fashion according to some *application scheme*, with the goal of staying as close as possible to the identity. Given that a suitable order of execution can be determined, an efficient representation can be maintained throughout the entire verification process.

For further information on this method, see [3].

Most effective for: proving equivalence of two circuits sharing a common structure, e.g., one circuit being a compiled or optimized version of the other.

Capable of showing: equivalence and non-equivalence

Drawback: Performance depends on the availability of a suitable *application scheme* or oracle for predicting when to apply gates from either circuit.

III-C Simulation Equivalence Checker (using Decision Diagrams)

It has been shown in [3], that even small differences in quantum circuits frequently affect the entire functional representation due to the reversibility of quantum computations. As a consequence, simulating both circuits with a couple of arbitrary input states, i.e., considering only a small part of the whole functionality, provides an attractive alternative to the methods described above.

In [4], several types of states have been proposed that allow to trade off efficiency versus performance:

- Classical stimuli (i.e., random *computational basis states*) already offer extremely high error detection rates in general and are comparatively fast to simulate, which makes them the default.
- Local quantum stimuli (i.e., random *single-qubit basis states*) are a little bit more computationally intensive, but provide even better error detection rates.
- Global quantum stimuli (i.e., random *stabilizer states*) offer the highest available error detection rate, while at the same time incurring the highest computational effort.

Most effective for: quickly detecting non-equivalence, even in cases where both circuits only differ slightly.

Capable of showing: non-equivalence

Drawback: Decision diagrams for state vectors might still be exponentially large in the worst case.

III-D ZX-Calculus Equivalence Checker

In [5], Kissinger and van de Wetering proposed an algorithm for optimizing quantum circuits based on ZX-calculus rewriting. In their initial article they also show that this rewriting approach can be used to prove the equivalence of quantum circuits. The idea is to construct the ZX-diagram of $G'^{-1}G$ and reduce this ZX-diagram using the rules of the ZX-calculus until no further rewrites can be made. If the resulting diagram consists only of wires (the identity ZX-diagram) then $G = G'$. This approach has been extended in [6] to also handle ancillary qubits, floating point inaccuracies in gate parameters, as well as permutations of input and output layouts of quantum circuits.

In [7], it has been shown that equivalence checking with the ZX-calculus naturally complements equivalence checking with decision diagrams. Since the size of the ZX-diagram during rewriting is bounded by the size of the initial diagram, this checker can be easily executed in parallel to the aforementioned approaches based on decision diagrams. In cases where the size of the decision diagrams explodes, the rewriting approach can often prove equivalence much more efficiently.

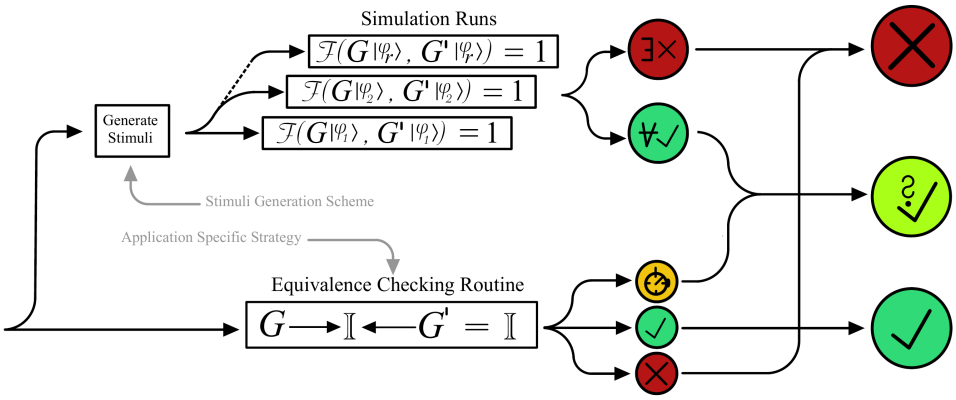
Most effective for: proving equivalence of two circuits involving many rotation gates with angles of the form $\frac{\pi}{k}$ for $k \in \mathbb{N}$.

Capable of showing: equivalence

Drawback: Due to the incompleteness of the rewriting rules, this equivalence checker cannot prove non-equivalence. Furthermore, multi-controlled gates have to be decomposed prior to the equivalence check, which can quickly lead to large ZX-diagrams and slow runtimes.

III-E Resulting Equivalence Checking Flow

QCEC implements and expands upon the flow proposed in [3] as illustrated in the following figure that orchestrates all the above equivalence checkers.



In general, the following steps are performed:

- First, a couple of simulation runs with random computational basis states are started. Should any of these simulations show a difference in the resulting states, the check is finished.
- In parallel, the alternating equivalence checker is started. In case the check finishes, i.e., it does not run into a timeout, a definitive result is returned. Otherwise, if none of the simulations have shown a difference, this strongly indicates that both circuits are probably equivalent.
- In addition to the above, the ZX-calculus equivalence checker is started. In case it finishes with an affirmative answer, the check is finished. In case it finishes and was not able to reduce the ZX-diagram to the identity, this indicates that the circuits are probably not equivalent.

IV Verifying the Results of Compilation Flows

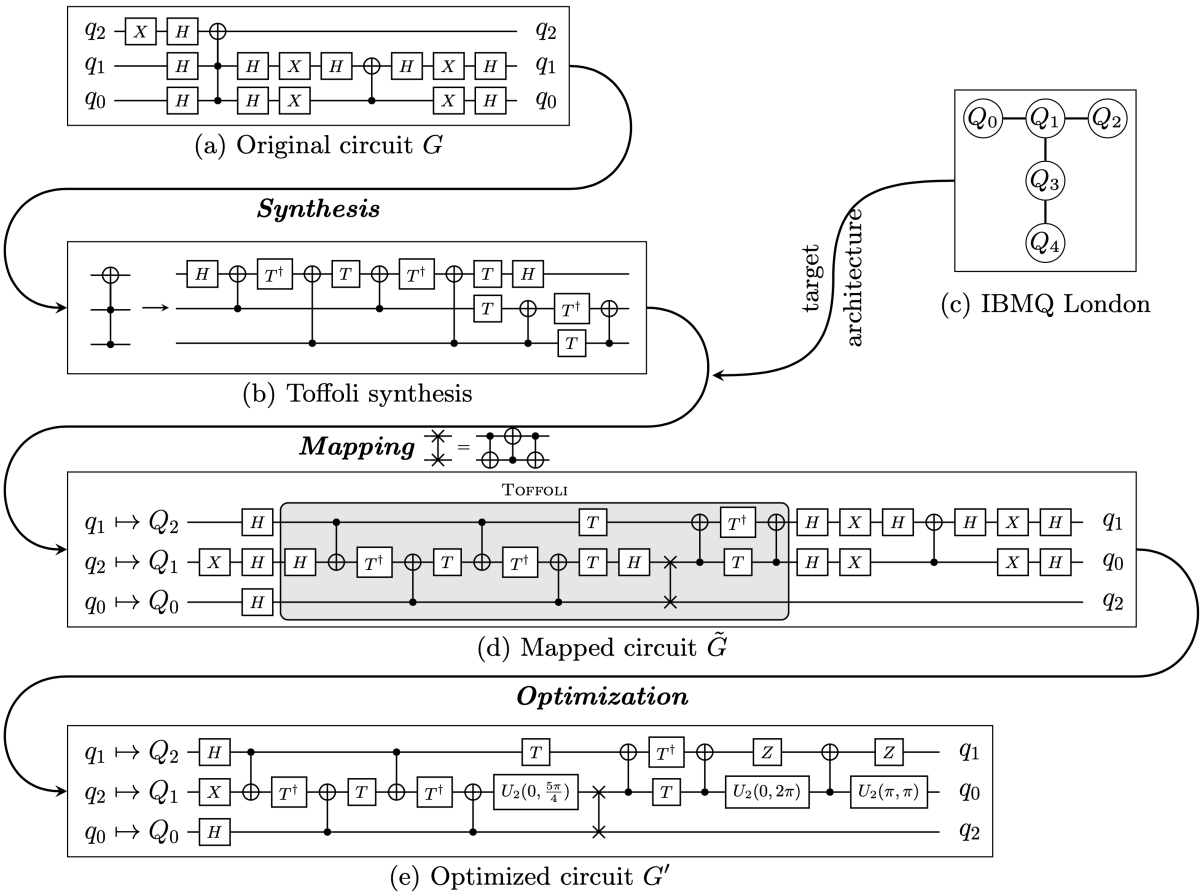
IV-A Quantum Circuit Compilation

Initially, quantum algorithms are described in a way which is agnostic of the device they are planned to be executed on. However, physical devices today impose several constraints on the circuits to be executed:

1. **Limited gate-set:** Typically, only a small set of gates is natively supported by devices, e.g., consisting of arbitrary single-qubit gates and the two-qubit CNOT operation.
2. **Limited connectivity:** Devices frequently limit the pairs of qubits that operations may be applied to. This is usually described by a coupling graph, where the graph's nodes represent the qubits and an edge between two nodes indicates that a CNOT operation may be applied to those qubits.
3. **Short coherence times and limited fidelity:** A device's physical qubits are inherently affected by noise. Until a certain threshold concerning the number of available qubits is reached, error correction is not yet an option.

The first two, i.e., the limited gate-set and connectivity, constitute hard constraints—a computation not conforming to these restrictions may not be executed on the device. In contrast, the short coherence time and limited gate fidelity represent soft constraints—a quantum circuit may be executed on a device, but it is not guaranteed to produce meaningful results if the circuit, e.g., is too large for the state to stay coherent.

Just as in classical computing, a conceptual algorithm needs to be compiled to the targeted architecture to address these constraints. Throughout the compilation process, the gates and the structure of the underlying circuit is changed considerably. The following figure exemplary illustrates the compilation of a 3-qubit circuit implementing the Grover search algorithm to the 5-qubit IBMQ London architecture.

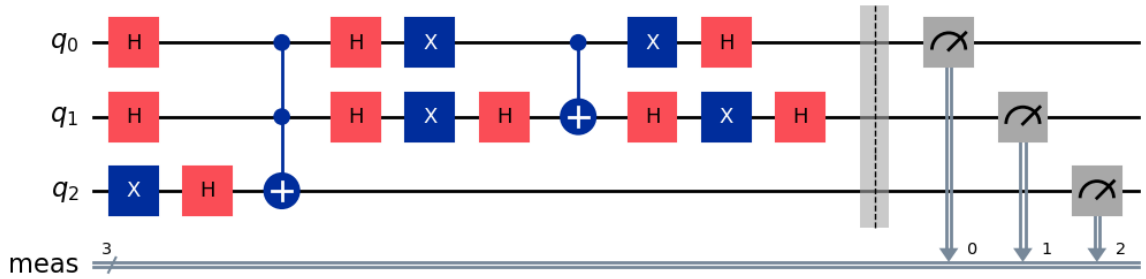


It is of utmost importance that, after compilation, the resulting (compiled) circuit still implements the same functionality as the originally given circuit. This can be guaranteed by verifying the results of the compilation flow, i.e., checking the equivalence of the original circuit description with the compiled quantum circuit.

IV-B Using QCEC to Verify Compilation Flow Results

First, construct your quantum circuit as usual.

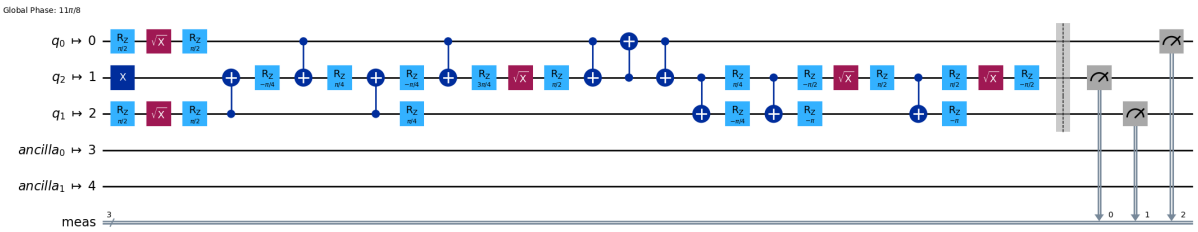
```
from qiskit import QuantumCircuit

circ = QuantumCircuit(3)
circ.x(2)
circ.h(range(3))
circ.ccx(0, 1, 2)
circ.h(range(2))
circ.x(range(2))
circ.h(1)
circ.cx(0, 1)
circ.h(1)
circ.x(range(2))
circ.h(range(2))
circ.measure_all()
circ.draw(output="mpl", style="iqp")
```



> **Note**
>
> It is essential to include measurements at the end of the circuit, since the equivalence checker uses the measurements to determine the final location of the logical qubits in the compiled circuit. Failing to do so may result in incorrect results because the checker will then simply assume that the logical qubits are mapped to the physical qubits in the same order as they appear in the circuit. Make sure to insert measurements *before* the circuit is compiled to the target architecture.

Then, compile the circuit to the desired target architecture.

```
from qiskit import transpile
from qiskit.providers.fake_provider import GenericBackendV2

# define the target architecture
backend = GenericBackendV2(num_qubits=5, coupling_map=[[0, 1], [1, 0], [1, 2], [2, 1], [1, 3],␣
↪[3, 1], [3, 4], [4, 3]])

# compile circuit to the target architecture
optimization_level = 1
circ_comp = transpile(circ, backend=backend, optimization_level=optimization_level)
circ_comp.draw(output="mpl", fold=-1, style="iqp")
```



Then, using QCEC to verify that the circuit has been compiled correctly is as easy as

```
from mqt import qcec

results = qcec.verify_compilation(circ, circ_comp, optimization_level=optimization_level)
results.equivalence
```

```
<EquivalenceCriterion.equivalent_up_to_global_phase: 4>
```

Check out the *reference documentation* for more information.

V Verifying Parameterized Quantum Circuits

V-A Variational Quantum Algorithms

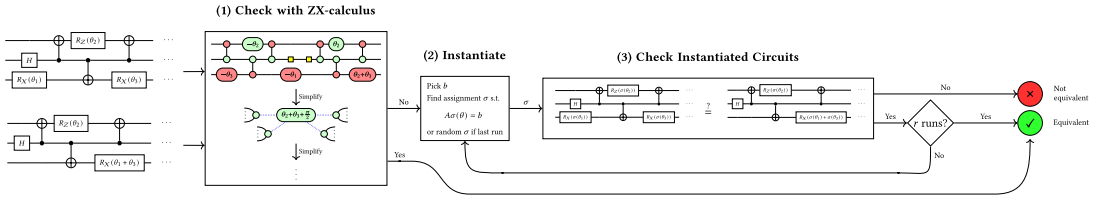
Variational quantum algorithms are a family of mixed quantum-classical algorithms that try to achieve a quantum advantage via low-depth circuits. This is achieved by offloading a substantial amount of computational work to a classical processor. The quantum circuits employed by variational quantum algorithms involve *parameterized gates* which depend on some a-priori uninstantiated variable.

Variational quantum algorithms try to optimize the circuit's parameters in each iteration with the classical post-processor while the quantum circuit is used to compute the cost function that is being optimized. Because recompiling the quantum circuit in each of these iterations is a costly procedure, the circuit is usually compiled in *parameterized* form in which the parameters tuned by the classical optimization routine are not bound to specific values.

As is the case with parameter-free circuits, errors can be made during the compilation process. Therefore, verifying the correctness of compilations of parameterized quantum circuits is an important task for near-term quantum computing.

V-B Equivalence Checking of Parameterized Quantum Circuits

Having unbound parameters in a quantum circuits brings new challenges to the task of quantum circuit verification as many data structures have difficulty supporting symbolic computations directly. However, *ZX-diagrams* are an exceptions to this as most rewrite rules used for equivalence checking with the *ZX-calculus* only involve summation of parameters. The ZX-calculus equivalence checker in QCEC cannot be used to prove non-equivalence of quantum circuits. To still show non-equivalence of parameterized quantum circuits QCEC uses a scheme of repeatedly instantiating a circuits parameters in such a way as to make the check as simple as possible while still guaranteeing that either equivalence or non-equivalence can be proven. The resulting *equivalence checking flow* looks as follows

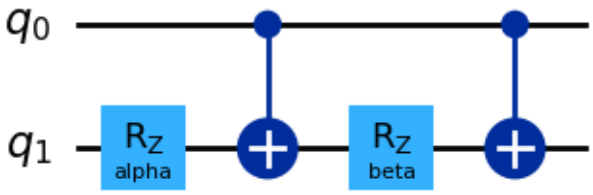


See [8] for more information.

V-C Using QCEC to Verify Parameterized Quantum Circuits

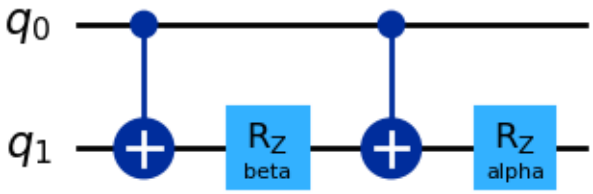
Consider the following quantum circuit

```python
from qiskit import QuantumCircuit
from qiskit.circuit import Parameter

alpha = Parameter("alpha")
beta = Parameter("beta")

qc_lhs = QuantumCircuit(2)
qc_lhs.rz(alpha, 1)
qc_lhs.cx(0, 1)
qc_lhs.rz(beta, 1)
qc_lhs.cx(0, 1)
qc_lhs.draw(output="mpl", style="iqp")
```



A well known commutation rule for the R_Z gate, states that this circuit is equivalent to the following one

```
qc_rhs = QuantumCircuit(2)
qc_rhs.cx(0, 1)
qc_rhs.rz(beta, 1)
qc_rhs.cx(0, 1)
qc_rhs.rz(alpha, 1)
qc_rhs.draw(output="mpl", style="iqp")
```



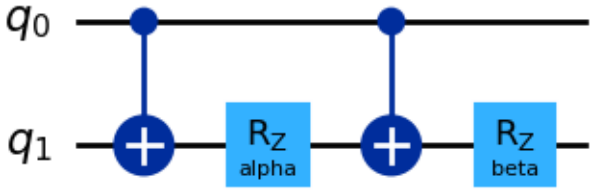
This equality can be proved with QCEC by using the *verify()* function just as with any regular circuit

```
from mqt import qcec

qcec.verify(qc_lhs, qc_rhs)
```

```
<EquivalenceCheckingManager.Results: equivalent>
```

QCEC also manages to show non-equivalence of parameterized quantum circuits. It is easy to erroneously exchange the parameters in the above commutation rule.

```
qc_rhs_err = QuantumCircuit(2)
qc_rhs_err.cx(0, 1)
qc_rhs_err.rz(alpha, 1)
qc_rhs_err.cx(0, 1)
qc_rhs_err.rz(beta, 1)
qc_rhs_err.draw(output="mpl", style="iqp")
```



QCEC will tell us that this is incorrect

```
qcec.verify(qc_lhs, qc_rhs_err)
```

```
<EquivalenceCheckingManager.Results: not_equivalent>
```

Check out the *reference documentation* for more information.

VI Partial Equivalence Checking

VI-A Partial Equivalence vs. Total Equivalence

Different definitions of quantum circuit equivalence have been proposed, with the most commonly used being total equivalence. Two circuits are considered totally equivalent when their matrix representations are exactly the same. However, it is often sufficient to consider observational equivalence, because the only way to extract information from a quantum circuit is through measurement. Therefore, partial equivalence has been defined, which is a weaker equality than total equivalence. Two circuits are considered partially equivalent if, for each possible input state, they have the same probabilities for each measurement outcome. In particular, partial equivalence doesn't consider qubits that are not measured or different phase angles for each measurement.

This definition is especially useful for circuits that do not measure all of their qubits at the end, or where some qubits are initialized to a certain value at the beginning. We refer to the qubits that are initialized to a certain value at the beginning of the computation as *ancillary* qubits. Without loss of generality, we assume that these qubits are initially set to $|0\rangle$. Any other initial state can be reached from the $|0\rangle$ state by applying the appropriate gates. The remaining qubits are the qubits that hold the input state, and they are referred to as *data* qubits.

Similarly, we differentiate between *measured* and *garbage* qubits. In order to read the output at the end of a circuit, a measurement is performed on some qubits, but not necessarily all of them. These qubits are the *measured* qubits. All qubits that are not measured are called the *garbage* qubits. Their final state doesn't influence the output we obtain from the computation.

For a circuit C we denote $P(t|\psi, C)$ as the probability that the quantum circuit C collapses to state $|t\rangle$ upon a measurement on the measured qubits, given that the data qubits have the initial state $|\psi\rangle$. Two circuits C_1 and C_2 are considered partially equivalent if, for each initial state $|\psi\rangle$ of the data qubits and each final state $|t\rangle$ of the measured qubits, it holds that $P(t|\psi, C_1) = P(t|\psi, C_2)$.

VI-B Checking Partial Equivalence of Quantum Circuits

Partial equivalence checking is a bit more costly than regular equivalence checking. It is necessary to reduce the contribution of garbage qubits and normalize the phase of each measurement outcome. Therefore, it is not enabled by default in QCEC. It can be activated using an option in the configuration parameters. If the option `check_partial_equivalence` is set to `True`, the equivalence checker will return `equivalent` not only for totally equivalent circuits, but also for partially equivalent circuits. The result will be `not_equivalent` if and only if the circuits are not partially equivalent. On the other hand, if the option `check_partial_equivalence` is set to `False`, then the equivalence checker considers total equivalence modulo ancillary qubits. This means that the garbage qubits are considered as if they were measured qubits, while ancillary qubits are set to zero, as in the partial equivalence check. The equivalence checker will return `equivalent` for totally equivalent circuits or `equivalent_up_to_global_phase` for circuits which differ only in their global phase and `not_equivalent` for other circuits.

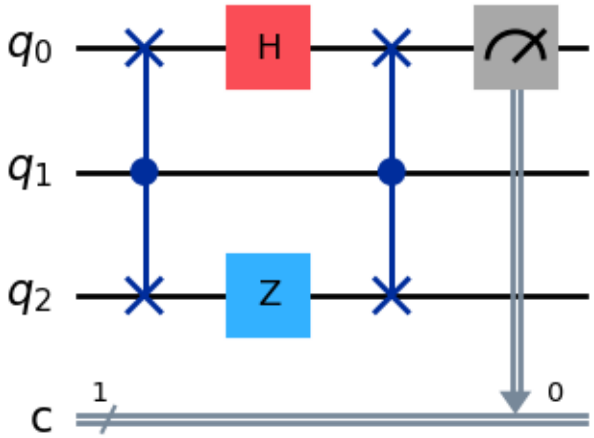
The following is a summary of the behaviour of each type of equivalence checker when the `check_partial_equivalence` option is set to `True`.

1. **Construction Equivalence Checker:** The construction checker supports partial equivalence checking by using decision diagrams to calculate the normalized phases of the possible outputs and then summing up the contribution of garbage qubits, in order to consider only the measurement probabilities of measured qubits. Then it compares the reduced decision diagrams of the two circuits.
2. **Alternating Equivalence Checker:** The alternating checker can only be used for circuits where at least one of the two does not contain ancillary qubits. It computes the composition of one circuit with the inverse of the other and verifies that the result resembles the identity. When checking for partial equivalence, it suffices to verify that the result resembles the identity modulo garbage qubits.
3. **Simulation Equivalence Checker:** The simulation checker computes a representation of the state vector resulting from simulating the circuit with certain initial states. Partial equivalence is enabled by summing up the contributions of garbage qubits.
4. **ZX-Calculus Equivalence Checker:** The ZX-calculus checker doesn't directly support partial equivalence, which is not a problem for the equivalence checking workflow, given that the ZX-calculus checker cannot demonstrate non-equivalence of circuits due to its incompleteness. Therefore it will simply output 'No Information' for circuits that are partially but not totally equivalent.

VI-C Using QCEC to Check for Partial Equivalence

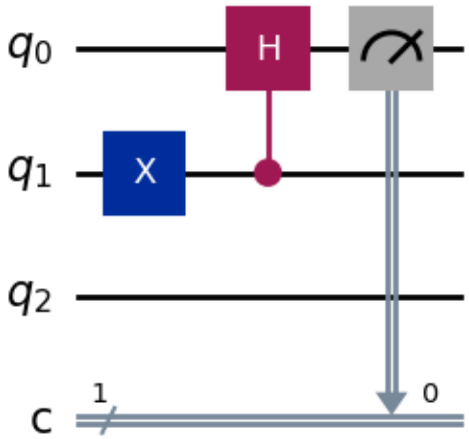
Consider the following quantum circuit with three qubits, which are all data qubits, but only one of them is measured.

```
from qiskit import QuantumCircuit

qc_lhs = QuantumCircuit(3, 1)
qc_lhs.cswap(1, 0, 2)
qc_lhs.h(0)
qc_lhs.z(2)
qc_lhs.cswap(1, 0, 2)
qc_lhs.measure(0, 0)

qc_lhs.draw(output="mpl", style="iqp")
```



Additionally, consider the following circuit, which only acts on two qubits.

```
qc_rhs = QuantumCircuit(3, 1)
qc_rhs.x(1)
qc_rhs.ch(1, 0)
qc_rhs.measure(0, 0)
qc_rhs.draw(output="mpl", style="iqp")
```



Then, these circuits are not totally equivalent, as can be shown using QCEC. The library even emits a handy warning in this case that indicates that two circuits have been shown to be non-equivalent, but at least one of the circuits does not measure all its qubits (i.e., has garbage qubits) and partial equivalence checking support has not been enabled.

```
from mqt.qcec import verify

verify(qc_lhs, qc_rhs)
```

```
[QCEC] Warning: at least one of the circuits has garbage qubits, but partial equivalence␣
↪checking is turned off. In order to take into account the garbage qubits, enable partial␣
↪equivalence checking. Consult the documentation for moreinformation.
```

```
<EquivalenceCheckingManager.Results: not_equivalent>
```

However, both circuits are partially equivalent, because they have the same probability distribution of measured values for any given initial input state. This equality can be proven with QCEC by enabling the `check_partial_equivalence` option:

```
verify(qc_lhs, qc_rhs, check_partial_equivalence=True)
```

```
<EquivalenceCheckingManager.Results: equivalent>
```

Source: The definitions and example for partial equivalence are described in [9].

QCEC is academic software. Thus, many of its built-in algorithms have been published as scientific papers.

If you use *QCEC* in your work, we would appreciate if you cited [3] (which subsumes [10] and [11]).

Furthermore, if you use any of the particular algorithms such as

- the compilation flow result verification scheme [12],
- the dedicated stimuli generation schemes [4],
- the transformation scheme for circuits containing non-unitaries [13],
- the equivalence checker based on ZX-diagrams [6], or
- the method for checking equivalence of parameterized circuits [8]

please consider citing their respective papers as well. A full list of references is given below.

VII mqt.qcec

MQT QCEC library.

This file is part of the MQT QCEC library released under the MIT license. See README.md or go to https://github.com/cda-tum/qcec for more information.

VII-A Submodules

mqt.qcec.compilation_flow_profiles

Module for generating compilation flow profiles for the equivalence checking process.

Classes

AncillaMode	Enum for the ancilla mode.

Functions

generate_profile_name(→ str)	Generate a profile name based on the given optimization level and ancilla mode.
generate_profile(→ None)	Generate a compilation flow profile for the given optimization level and ancilla mode.

Module Contents

class AncillaMode(**args*, ***kwds*)

Bases: enum.Enum

Enum for the ancilla mode.

NO_ANCILLA = 'noancilla'

No ancilla qubits are used.

RECURSION = 'recursion'

A single ancilla is used in a recursive manner.

V_CHAIN = 'v-chain'

A chain of ancilla qubits is used.

generate_profile_name(*optimization_level: int = 1*, *mode:* AncillaMode = *AncillaMode.NO_ANCILLA*) → str

Generate a profile name based on the given optimization level and ancilla mode.

generate_profile(*optimization_level: int = 1*, *mode:* AncillaMode = *AncillaMode.NO_ANCILLA*, *filepath: Path | None = None*) → None

Generate a compilation flow profile for the given optimization level and ancilla mode.

Parameters

- **optimization_level** – The IBM Qiskit optimization level to use for the profile (0, 1, 2, or 3). Defaults to 1.
- **mode** – The *ancilla mode* used for realizing multi-controlled Toffoli gates, as available in Qiskit. Defaults to *AncillaMode.NO_ANCILLA*.
- **filepath** – The path to the directory where the profile should be stored. Defaults to the `profiles` directory in the `mqt.qcec` package.

mqt.qcec.configuration

Configuration options for the equivalence checking manager.

Classes

ConfigurationOptions	A dictionary of configuration options.

Functions

augment_config_from_kwargs(→ None)	Augment an existing *Configuration* with options from a collection of keyword arguments.

Module Contents

class ConfigurationOptions

Bases: TypedDict

A dictionary of configuration options.

The keys of this dictionary are the names of the configuration options. The values are the values of the configuration options.

augment_config_from_kwargs(*config:* Configuration, *kwargs:* ConfigurationOptions) → None

Augment an existing *Configuration* with options from a collection of keyword arguments.

Parameters

- **config** – The configuration to augment.
- **kwargs** – The arguments to build the configuration from.

mqt.qcec.parameterized

Functionality for checking equivalence of parameterized quantum circuits.

Functions

check_parameterized_zx(...)	Check circuits for equivalence with the ZX-calculus.
check_instantiated(...)	Check circuits for equivalence with DD equivalence checker.
check_instantiated_random(...)	Check whether circuits are equivalent for random instantiation of symbolic parameters.
check_parameterized(...)	Equivalence checking flow for parameterized circuit.

Module Contents

check_parameterized_zx(*circ1: QuantumComputation*, *circ2: QuantumComputation*, *configuration:* Configuration) → *Results*

Check circuits for equivalence with the ZX-calculus.

check_instantiated(*circ1: QuantumComputation*, *circ2: QuantumComputation*, *configuration:* Configuration) → *Results*

Check circuits for equivalence with DD equivalence checker.

check_instantiated_random(*circ1: QuantumComputation*, *circ2: QuantumComputation*, *params: list[Variable]*, *configuration:* Configuration) → *Results*

Check whether circuits are equivalent for random instantiation of symbolic parameters.

check_parameterized(*circ1: QuantumComputation*, *circ2: QuantumComputation*, *configuration:* Configuration) → *Results*

Equivalence checking flow for parameterized circuit.

mqt.qcec.pyqcec

The Python interface for QCEC.

Classes

EquivalenceCheckingManager	The main class of QCEC.
Configuration	Provides all the means to configure QCEC.
EquivalenceCriterion	Captures all the different notions of equivalence that can be the result of a *run()*.
ApplicationScheme	Describes the order in which the individual operations of both circuits are applied during the equivalence check.
StateType	The type of states used in the simulation checker allows trading off efficiency versus performance.

Module Contents

class EquivalenceCheckingManager(*circ1: QuantumComputation*, *circ2: QuantumComputation*, *config:* Configuration = ...)

The main class of QCEC.

Allows checking the equivalence of quantum circuits based on the methods proposed in [3]. It features many configuration options that orchestrate the procedure.

property qc1: QuantumComputation

The first circuit to be checked.

property qc2: QuantumComputation

The second circuit to be checked.

run() → None

Execute the equivalence check as configured.

property results: *Results*

The results of the equivalence check.

property equivalence: *EquivalenceCriterion*

The *EquivalenceCriterion* determined as the result of the equivalence check.

disable_all_checkers() → None

Disable all equivalence checkers.

set_application_scheme(*scheme:* ApplicationScheme = ...) → None

Set the *ApplicationScheme* used for all checkers (based on decision diagrams).

Parameters: **scheme** – The application scheme. Defaults to *ApplicationScheme.proportional*.

set_gate_cost_profile(*profile: str* = '') → None

Set the *profile* used in the *Gate Cost* application scheme for all checkers (based on decision diagrams).

Parameters: **profile** – The path to the profile file.

class Results

Captures the main results and statistics from *run()*.

preprocessing_time: float

Time spent during preprocessing (in seconds).

check_time: float

Time spent during equivalence check (in seconds).

equivalence: *EquivalenceCriterion*

Final result of the equivalence check.

started_simulations: int

Number of simulations that have been started.

performed_simulations: int

Number of simulations that have been finished.

cex_input: VectorDD

DD representation of the initial state that produced a counterexample.

cex_output1: VectorDD

DD representation of the first circuit's counterexample output state.

cex_output2: VectorDD

DD representation of the second circuit's counterexample output state.

performed_instantiations: int

Number of circuit instantiations performed during equivalence checking of parameterized quantum circuits.

checker_results: dict[str, Any]

Dictionary of the results of the individual checkers.

considered_equivalent() → bool

Convenience function to check whether the result is considered equivalent.

json() → dict[str, Any]

Returns a JSON-style dictionary of the results.

class Configuration

Provides all the means to configure QCEC.

All options are split into the following categories:

- *Execution*
- *Optimizations*

- *Application*
- *Functionality*
- *Simulation*
- *Parameterized*

All options can be passed to the *verify()* and *verify_compilation()* functions as keyword arguments. There, they are incorporated into the *Configuration* using the *augment_config_from_kwargs()* function.

class Execution

Options that orchestrate the *run()* method.

parallel: bool = True

Set whether execution should happen in parallel. Defaults to `True`.

nthreads: int

Set the maximum number of threads to use. Defaults to the maximum number of available threads reported by the OS.

timeout: float = 0.0

Set a timeout for *run()* (in seconds).

Defaults to `0.`, which means no timeout.

> **Note**
>
> Timeouts in QCEC work by checking an atomic flag in between the application of gates (for DD-based checkers) or rewrite rules (for the ZX-based checkers). Unfortunately, this means that an operation needs to be fully applied before a timeout can set in. If a certain operation during the equivalence check takes a very long time (e.g., because the DD is becoming very large), the timeout will not be triggered until that operation is finished. Thus, it is possible that the timeout is not triggered at the expected time, and it might seem like the timeout is being ignored.
>
> Unfortunately, there is no clean way to kill a thread without letting it finish its computation. That's something that could be made possible by switching from multi-threading to multi-processing, but the overhead of processes versus threads is huge on certain platforms and that would not be a good trade-off. In addition, more fine-grained abortion checks would significantly decrease the overall performance due to all the branching that would be necessary.
>
> Consequently, timeouts in QCEC are a best-effort feature, and they should not be relied upon to always work as expected. From experience, they tend to work reliably well for the ZX-based checkers, but they are less reliable for the DD-based checkers.

run_construction_checker: bool = False

Set whether the construction checker should be executed.

Defaults to `False` since the alternating checker is to be preferred in most cases.

run_simulation_checker: bool = True

Set whether the simulation checker should be executed.

Defaults to `True` since simulations can quickly show the non-equivalence of circuits in many cases.

run_alternating_checker: bool = True

Set whether the alternating checker should be executed.

Defaults to `True` since staying close to the identity can quickly show the equivalence of circuits in many cases.

run_zx_checker: bool = True

Set whether the ZX-calculus checker should be executed.

Defaults to `True` but arbitrary multi-controlled operations are only partially supported.

numerical_tolerance: float = 2e-13

Set the numerical tolerance of the underlying decision diagram package.

Defaults to `2e-13` and should only be changed by users who know what they are doing.

class Optimizations

Options that influence which circuit optimizations are applied during pre-processing.

fuse_single_qubit_gates: bool = True

Fuse consecutive single-qubit gates by grouping them together.

Defaults to `True` as this typically increases the performance of the subsequent equivalence check.

reconstruct_swaps: bool = True

Try to reconstruct SWAP gates that have been decomposed (into a sequence of 3 CNOT gates) or optimized away (as a consequence of a SWAP preceded or followed by a CNOT on the same qubits).

Defaults to `True` since this reconstruction enables the efficient tracking of logical to physical qubit permutations throughout circuits that have been mapped to a target architecture.

remove_diagonal_gates_before_measure: bool = False

Remove any diagonal gates at the end of the circuit.

This might be desirable since any diagonal gate in front of a measurement does not influence the probabilities of the respective states.

Defaults to `False` since, in general, circuits differing by diagonal gates at the end should still be considered non-equivalent.

transform_dynamic_circuit: bool = False

Circuits containing dynamic circuit primitives such as mid-circuit measurements, resets, or classically-controlled operations cannot be verified in a straight-forward fashion due to the non-unitary nature of these primitives, which is why this setting defaults to `False`.

By enabling this optimization, any dynamic circuit is first transformed to a circuit without non-unitary primitives by, first, substituting qubit resets with new qubits and, then, applying the deferred measurement principle to defer measurements to the end.

reorder_operations: bool = True

The operations of a circuit are stored in a sequential container. This introduces some dependencies in the order of operations that are not naturally present in the quantum circuit. As a consequence, two quantum circuits that contain exactly the same operations, list their operations in different ways, also apply there operations in a different order. This optimization pass established a canonical ordering of operations by, first, constructing a directed, acyclic graph for the operations and, then, traversing it in a breadth-first fashion.

Defaults to `True`.

backpropagate_output_permutation: bool = False

Backpropagate the output permutation to the input permutation.

Defaults to `False` since this might mess up the initially given input permutation. Can be helpful for dynamic quantum circuits that have been transformed to a static circuit by enabling the *transform_dynamic_circuit* optimization.

elide_permutations: bool = True

Elide permutations from the circuit by permuting the qubits in the circuit and eliminating SWAP gates from the circuits.

Defaults to `True` as this typically boosts performance.

class Application

Options describing the *ApplicationScheme* used for the individual equivalence checkers.

construction_scheme: *ApplicationScheme*

The *ApplicationScheme* used for the construction checker.

simulation_scheme: *ApplicationScheme*

The *ApplicationScheme* used for the simulation checker.

alternating_scheme: *ApplicationScheme*

The *ApplicationScheme* used for the alternating checker.

profile: str

The *Gate Cost* application scheme can be configured with a profile that specifies the cost of gates. This profile can be set via a file constructed like a lookup table. Every line <GATE_ID> <N_CONTROLS> <COST> specifies the cost for a given gate type and with a certain number of controls, e.g., `X 0 1` denotes that a single-qubit X gate has a cost of 1, while `X 2 15` denotes that a Toffoli gate has a cost of 15.

class Functionality

Options for all checkers that consider the whole functionality of a circuit.

trace_threshold: float = 1e-08

While decision diagrams are canonical in theory, i.e., equivalent circuits produce equivalent decision diagrams, numerical inaccuracies and approximations can harm this property.

This can result in a scenario where two decision diagrams are really close to one another, but cannot be identified as such by standard methods (i.e., comparing their root pointers). Instead, for two decision diagrams `U` and `U'` representing the functionalities of two circuits `G` and `G'`, the trace of the product of one decision diagram with the inverse of the other can be computed and compared to the trace of the identity.

Alternatively, it can be checked, whether `U*U`^-1` is "close enough" to the identity by recursively checking that each decision diagram node is close enough to the identity structure (i.e., the first and last successor have weights close to one, while the second and third successor have weights close to zero). Whenever any decision diagram node differs from this structure by more than the configured threshold, the circuits are concluded to be non-equivalent.

Defaults to `1e-8`.

check_partial_equivalence: bool = False

Two circuits are partially equivalent if, for each possible initial input state, they have the same probability for each measurement outcome. If set to `True`, a check for partial equivalence will be performed and the contributions of garbage qubits to the circuit are ignored. If set to `False`, the checker will output 'not equivalent' for circuits that are partially equivalent but not totally equivalent. In particular, garbage qubits will be treated as if they were measured qubits.

Defaults to `False`.

class Simulation

Options that influence the simulation checker.

fidelity_threshold: float

Similar to *trace_threshold*, this setting is here to tackle numerical inaccuracies in the simulation checker. Instead of computing a trace, the fidelity between the states resulting from the simulation is computed. Whenever the fidelity differs from `1.` by more than the configured threshold, the circuits are concluded to be non-equivalent.

Defaults to `1e-8`.

max_sims: int

The maximum number of simulations to be started for the simulation checker.

In practice, just a couple of simulations suffice in most cases to detect a potential non-equivalence. Either defaults to 16 or the maximum number of available threads minus 2, whichever is more.

state_type: *StateType* = Ellipsis

The *type of states* used for the simulations in the simulation checker.

Defaults to *StateType.computational_basis*.

seed: int = 0

The seed used in the quantum state generator.

Defaults to `0`, which means that the seed is chosen non-deterministically for each program run.

class Parameterized

Options that influence the equivalence checking scheme for parameterized circuits.

parameterized_tolerance: float = 1e-12

Set threshold below which instantiated parameters shall be considered zero.

Defaults to `1e-12`.

additional_instantiations: int = 0

Number of instantiations shall be performed in addition to the default ones.

For parameterized circuits that cannot be shown to be equivalent by the ZX checker the circuits are instantiated with concrete values for parameters and subsequently checked with QCEC's default schemes. The first instantiation tries to set as many gate parameters to 0. The last instantiations initializes the parameters with random values to guarantee completeness of the equivalence check. Because random instantiation is costly, additional instantiations can be performed that lead to simpler equivalence checking instances as the random instantiation. This option changes how many of those additional checks are performed.

json() → dict[str, Any]

Returns a JSON-style dictionary of the configuration.

class EquivalenceCriterion(*value: int*)

class EquivalenceCriterion(*criterion: Literal['no_information', 'not_equivalent', 'equivalent', 'equivalent_up_to_phase', 'equivalent_up_to_global_phase', 'probably_equivalent', 'probably_not_equivalent']*)

class EquivalenceCriterion(*criterion: str*)

Captures all the different notions of equivalence that can be the result of a *run()*.

no_information: ClassVar[*EquivalenceCriterion*] = Ellipsis

No information on the equivalence is available. This can be because the check has not been run or that a timeout happened.

not_equivalent: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are shown to be non-equivalent.

equivalent: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are shown to be equivalent.

equivalent_up_to_global_phase: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are equivalent up to a global phase factor.

equivalent_up_to_phase: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are equivalent up to a certain (global or relative) phase.

probably_equivalent: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are probably equivalent. A result obtained whenever a couple of simulations did not show the non-equivalence in the simulation checker.

probably_not_equivalent: ClassVar[*EquivalenceCriterion*] = Ellipsis

Circuits are probably not equivalent. A result obtained whenever the ZX-calculus checker could not reduce the combined circuit to the identity.

class ApplicationScheme(*value: int*)

class ApplicationScheme(*scheme: Literal['sequential', 'one_to_one', 'proportional', 'lookahead', 'gate_cost']*)

class ApplicationScheme(*scheme: str*)

Describes the order in which the individual operations of both circuits are applied during the equivalence check.

In case of the alternating equivalence checker, this is the key component to allow the intermediate decision diagrams to remain close to the identity (as proposed in [3]). See *Verifying the Results of Compilation Flows* for more information on the dedicated application scheme for verifying compilation flow results (as proposed in [12]).

In case of the other checkers, which consider both circuits individually, using a non-sequential application scheme can significantly boost the operation caching performance in the underlying decision diagram package.

sequential: ClassVar[*ApplicationScheme*] = Ellipsis

Applies all gates from the first circuit, before proceeding with the second circuit.

Referred to as *"reference"* in [3].

one_to_one: ClassVar[*ApplicationScheme*] = Ellipsis

Alternates between applications from the first and the second circuit.

Referred to as *"naive"* in [3].

proportional: ClassVar[*ApplicationScheme*] = Ellipsis

Alternates between applications from the first and the second circuit, but applies the gates in proportion to the number of gates in each circuit.

lookahead: ClassVar[*ApplicationScheme*] = Ellipsis

Looks whether an application from the first circuit or the second circuit yields the smaller decision diagram.

Only works for the alternating equivalence checker.

gate_cost: ClassVar[*ApplicationScheme*] = Ellipsis

Each gate of the first circuit is associated with a corresponding cost according to some cost function *f(...)*. Whenever a gate *g* from the first circuit is applied *f(g)* gates are applied from the second circuit.

Referred to as *"compilation_flow"* in [12].

class StateType(*value: int*)

class StateType(*state_type: Literal['computational_basis', 'random_1Q_basis', 'stabilizer']*)

class StateType(*state_type: str*)

The type of states used in the simulation checker allows trading off efficiency versus performance.

- Classical stimuli (i.e., random *computational basis states*) already offer extremely high error detection rates in general and are comparatively fast to simulate, which makes them the default.
- Local quantum stimuli (i.e., random *single-qubit basis states*) are a little bit more computationally intensive, but provide even better error detection rates.
- Global quantum stimuli (i.e., random *stabilizer states*) offer the highest available error detection rate, while at the same time incurring the highest computational effort.

For details, see [4].

computational_basis: ClassVar[*StateType*] = Ellipsis

Randomly choose computational basis states. Also referred to as "*classical*.

random_1Q_basis: ClassVar[*StateType*] = Ellipsis

Randomly choose a single-qubit basis state for each qubit from the six-tuple *(|0>, |1>, |+>, |->, |L>, |R>)*. Also referred to as *local_random*.

stabilizer: ClassVar[*StateType*] = Ellipsis

Randomly choose a stabilizer state by creating a random Clifford circuit. Also referred to as *global_random*.

mqt.qcec.verify

The main entry point for the QCEC package.

Functions

verify(...)	Verify that `circ1` and `circ2` are equivalent.

Module Contents

verify(*circ1: QuantumComputation | str | PathLike[str] | QuantumCircuit*, *circ2: QuantumComputation | str | PathLike[str] | QuantumCircuit*, *configuration:* Configuration | *None* = *None*, ***kwargs: mqt.qcec._compat.typing.Unpack[*ConfigurationOptions*]*) → *Results*

Verify that `circ1` and `circ2` are equivalent.

Wraps creating an instance of *EquivalenceCheckingManager*, calling *EquivalenceCheckingManager.run()*, and returning *EquivalenceCheckingManager.results*.

There are two (non-exclusive) ways of configuring the equivalence checking process:

1. Pass a *Configuration* instance as the `configuration` argument.

2. Pass keyword arguments to this function. These are directly incorporated into the *Configuration*. Any existing configuration is overridden by keyword arguments.

Parameters

- **circ1** – The first circuit.
- **circ2** – The second circuit.
- **configuration** – The configuration to use for the equivalence checking process.
- ****kwargs** – Keyword arguments to configure the equivalence checking process.

Returns The results of the equivalence checking process.

mqt.qcec.verify_compilation_flow

Verify compilation flow results.

Functions

verify_compilation(...)	Verify compilation flow results.

Module Contents

verify_compilation(*original_circuit: QuantumComputation | str | PathLike[str] | QuantumCircuit*, *compiled_circuit: QuantumComputation | str | PathLike[str] | QuantumCircuit*, *optimization_level: int* = *1*, *ancilla_mode:* AncillaMode = *AncillaMode.NO_ANCILLA*, *configuration:* Configuration | *None* = *None*, ***kwargs: mqt.qcec._compat.typing.Unpack[*ConfigurationOptions*]*) → *Results*

Verify compilation flow results.

Similar to *verify*, but uses a dedicated compilation flow profile to guide the equivalence checking process. The compilation flow profile is determined by the `optimization_level` and `ancilla_mode` arguments.

There are two (non-exclusive) ways of configuring the equivalence checking process:

1. Pass a *Configuration* instance as the `configuration` argument.

2. Pass keyword arguments to this function. These are directly incorporated into the *Configuration*. Any existing configuration is overridden by keyword arguments.

Parameters

- **original_circuit** – The original circuit.

- **compiled_circuit** – The compiled circuit.
- **optimization_level** – The optimization level used for compiling the circuit (0, 1, 2, or 3). Defaults to 1.
- **ancilla_mode** – The *ancilla mode* used for realizing multi-controlled Toffoli gates, as available in Qiskit. Defaults to *AncillaMode.NO_ANCILLA*.
- **configuration** – The configuration to use for the equivalence checking process.
- ****kwargs** – Keyword arguments to configure the equivalence checking process.

Returns The results of the equivalence checking process.

References

[1] Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, Nils Quetschlich, Damian Rovara, Aaron Sander, Ludwig Schmid, Daniel Schoenberger, Yannick Stade, and Lukas Burgholzer. The MQT handbook: A summary of design automation tools and software for quantum computing. In *Int'l Conf. on Quantum Software*. 2024. arXiv:2405.17543, doi:10.1109/QSW62656.2024.00013.

[2] Robert Wille, Stefan Hillmich, and Burgholzer Lukas. Tools for quantum computing based on decision diagrams. *ACM Transactions on Quantum Computing*, 2021. [PDF], doi:10.1145/3491246.

[3] Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2021. [PDF], doi:10.1109/TCAD.2020.3032630.

[4] Lukas Burgholzer, Kueng Richard, and Robert Wille. Random stimuli generation for the verification of quantum circuits. In *Asia and South Pacific Design Automation Conf.* 2021. [PDF], doi:10.1145/3394885.3431590.

[5] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. In *Quantum Physics and Logic*. 2019. [PDF], doi:10.4204/EPTCS.318.14.

[6] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking of quantum circuits with the ZX-calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2022. [PDF], doi:10.1109/JETCAS.2022.3202204.

[7] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking paradigms in quantum circuit design: A case study. In *Design Automation Conf.* 2022. [PDF], doi:10.1145/3489517.3530480.

[8] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking of parameterized quantum circuits: Verifying the compilation of variational quantum algorithms. In *Asia and South Pacific Design Automation Conf.* 2023. [PDF], doi:10.1145/3566097.3567932.

[9] Tian-Fu Chen, Jie-Hong R. Jiang, and Min-Hsiu Hsieh. Partial equivalence checking of quantum circuits. In *Int'l Conf. on Quantum Computing and Engineering*. 2022. [PDF], doi:10.1109/qce53715.2022.00082.

[10] Lukas Burgholzer and Robert Wille. Improved DD-based equivalence checking of quantum circuits. In *Asia and South Pacific Design Automation Conf.* 2020. [PDF], doi:10.1109/ASP-DAC47756.2020.9045153.

[11] Lukas Burgholzer and Robert Wille. The power of simulation for equivalence checking in quantum computing. In *Design Automation Conf.* 2020. [PDF], doi:10.1109/DAC18072.2020.9218563.

[12] Lukas Burgholzer, Rudy Raymond, and Robert Wille. Verifying results of the IBM Qiskit quantum circuit compilation flow. In *Int'l Conf. on Quantum Computing and Engineering*. 2020. [PDF], doi:10.1109/QCE49297.2020.00051.

[13] Lukas Burgholzer and Robert Wille. Handling non-unitaries in quantum circuit equivalence checking. In *Design Automation Conf.* 2022. [PDF], doi:10.1145/3489517.3530482.

Index

S

T

V