JKQ: JKU Tools for Quantum Computing

(Invited Paper)

Robert Wille*† Stefan Hillmich* Lukas Burgholzer*
*Johannes Kepler University Linz, Austria
†Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria
{robert.wille,stefan.hillmich,lukas.burgholzer}@jku.at
https://iic.jku.at/eda/research/quantum/

ABSTRACT

With quantum computers on the brink of practical applicability, there is a lively community that develops toolkits for the design of corresponding quantum circuits. Many of the problems to be tackled here are similar to design problems from the classical realm for which sophisticated design automation tools have been developed in the previous decades. In this paper, we present *JKQ*—a set of tools for quantum computing developed at the *Johannes Kepler University (JKU) Linz* which utilizes this design automation expertise. By this, we offer complementary approaches for many design problems in quantum computing such as simulation, compilation, or verification. In the following, we provide an introduction of the tools for potential users who would like to work with them as well as potential developers aiming to extend them.

1 INTRODUCTION

Quantum computing is gaining momentum and the development of physical realizations has shown significant progress in the recent past—indicating that quantum computing approaches the barrier of being an established technology rather than an emerging one. Accordingly, there is a lively community of researchers and industrial stakeholders currently developing toolkits that allow to work with these machines (such as IBM's *Qiskit* [1], Google's *Cirq* [7], or Rigetti's *Forest* [9]). However, developing efficient design methods is an immensely complex task due to the combinatorial and exponential nature of many design problems considered for quantum computing—some have even been proven to be NP-complete [2], coNP-hard [16], or QMA-complete [13].

In the classical realm of electronic circuits and systems, similar design tasks are well-defined and have been studied by researchers and engineers for decades—resulting in sophisticated design tools for software as well as hardware that are taken for granted today and can handle complexity of impressive scales. The availability of those methods and tools is a main reason for the utilization and penetration of (classical) electronic devices into almost all parts of our daily life.

In this work, we present tools for quantum computing that utilize this design automation expertise. More precisely, we introduce *JKQ*—a set of corresponding tools developed at the *Johannes Kepler University (JKU) Linz*. While certainly each tool (including ours) has its strengths and weaknesses, we offer complementary approaches for many of the problems that need to be tackled when designing quantum circuits—including simulation, compilation, or verification of quantum circuits. By making *JKQ* tools publicly available as open source, we also provide other researchers with the option to incorporate the underlying methods into their existing tools. This already motivated, e.g., IBM and Atos to include our simulation approach based on decision diagrams as well as one of our mapping techniques based on informed search algorithms into their tools.

In the following, we provide an introduction of the tools for potential users which would like to work with them as well as potential developers aiming to extend them. Therefore, Sec. 2 reviews the necessary background on quantum computing. Then, Sec. 3 briefly details three fundamental design tasks in quantum computing and how to approach them with *JKQ* tools from a user perspective. Afterwards, Sec. 4 gives insight on how the *JKQ* tool set is organized and where interested developers can start to extend the available tools with their own methods. Finally, Sec. 5 concludes the paper.

2 BACKGROUND

In the realm of quantum computing, the basic unit of information is the *quantum bit* or *qubit*, which—in contrast to classical bits—can not only be in one of its two orthogonal basis states (denoted $|0\rangle$ and $|1\rangle$ using Dirac notation), but also in an (almost) arbitrary *superposition* of both. This is expressed by complex *amplitudes* in front of the basis states, i.e., $\alpha_0 |0\rangle + \alpha_1 |1\rangle$, with the normalization constraint $|\alpha_0|^2 + |\alpha_1|^2 = 1$. This allows an n-qubit quantum system to represent 2^n different complex values at once—exponentially more than classical n-bit systems (which can only represent n different Boolean values at a time). The quantum states themselves are commonly represented by column vectors. A second important quantum mechanical phenomenon exploited for quantum computing is *entanglement*, i.e., operations on a single qubit may affect other qubits as well.

Example 1. *Consider a quantum system with two qubits in the state*

$$|\psi\rangle = \tfrac{1}{\sqrt{2}} \cdot |00\rangle + 0 \cdot |01\rangle + \tfrac{1}{\sqrt{2}} \cdot |10\rangle + 0 \cdot |11\rangle = \tfrac{1}{\sqrt{2}} \left(|00\rangle + |10\rangle\right).$$

This is a valid quantum state since $|1/\sqrt{2}|^2 + |1/\sqrt{2}|^2 = 1$. *The corresponding vector representation is commonly denoted* $\left[\, 1/\sqrt{2}\ 0\ 1/\sqrt{2}\ 0 \,\right]^{\mathrm{T}}$.

Figure 1: Quantum circuit diagram

Operations to be applied to n-qubit quantum states are defined by $2^n \times 2^n$-dimensional unitary matrices[1] and, hence, are inherently reversible.

Example 2. *Frequently used quantum operations are the Hadamard operation (H), the Pauli-X operation (X), and the controlled-NOT operation (CNOT), which set a qubit into superposition (H), invert the state of a qubit (X), or apply an X operation to a target qubit iff the state of a designated control qubit is $|1\rangle$ (CNOT), respectively. As matrices, these are denoted as follows:*

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Applying the CNOT operation to the quantum state $|\psi\rangle$ from Ex. 1 yields the following quantum state $|\psi'\rangle$:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{CNOT} \times \underbrace{\begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \\ 0 \end{bmatrix}}_{|\psi\rangle} = \underbrace{\begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 0 \\ 1/\sqrt{2} \end{bmatrix}}_{|\psi'\rangle} \equiv \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

In a physical quantum computer, the only way to retrieve information about a quantum state is measurement, which yields exactly one basis state. After measuring, the quantum state collapses to the measured quantum state, i.e., superposition and entanglement are destroyed. Therefore, subsequent measurements on the same qubits will yield the same results.

Example 3. *Measuring the the quantum state $|\psi'\rangle$ from Ex. 2 would result in either $|00\rangle$ or $|11\rangle$—each with a probability of 50 %. After this measurement, the quantum state vector would either be $[\,1\;0\;0\;0\,]^{\mathrm{T}}$ or $[\,0\;0\;0\;1\,]^{\mathrm{T}}$, depending on the measurement result.*

Quantum algorithms are commonly described by circuit diagrams with horizontal lines representing the qubits and symbols on these lines representing the quantum operations to be applied from left to right.

Example 4. *Fig. 1 shows a quantum circuit diagram describing a prominent quantum algorithm, namely Grover's algorithm [10]. Control qubits are denoted by "$\bullet$" whereas the target of the CNOT is denoted by "$\oplus$". The two-controlled NOT gate is also called* Toffoli *gate.*

3 JKQ TOOLS FOR USERS

JKQ tools offer an interface for users to leverage the power of design automation for quantum computing as a black box: The user provides the input and does not have to develop a deeper understanding of the methods. In the following, Sec. 3.1 shows how to download and build the JKQ tools. Then, we discuss three fundamental design problems in quantum computing, namely simulation (Sec. 3.2), compilation (Sec 3.3), and verification (Sec 3.4), and show how the JKQ tools can be applied to address them.

3.1 Download and Installation

Using the tools under UNIX operating systems requires cloning the GitHub repository https://github.com/iic-jku/jkq.git and building the tools as follows[2]:

```
1 $ git clone https://github.com/iic-jku/jkq.git
2 $ cd jkq/
3 $ ./jkq build
4 Building JKQ Tools
5 [...]
6 $ ./jkq help
7 [displays available commands]
```

3.2 Simulation

Simulating quantum circuits on a classical machine is essential for the development of (prototypes of) quantum algorithms, for considering the behavior of physical quantum computers, as well as for studying error models. In fact, classical simulations of quantum computations provide deeper insights, since it allows to observe the individual amplitudes of a quantum state (which is impossible when executing a quantum computation on a quantum device). Moreover, costly executions on real quantum devices—whose availability is still rather limited—can be avoided when using simulators.

The simulation of quantum circuits is commonly conducted by matrix-vector multiplication as described in Sec. 2. This approach quickly becomes intractable due to the exponential growth of the quantum state with respect to the number of qubits—quickly rendering such simulations infeasible even on supercomputer clusters. Simulation methodologies based on decision diagrams [17, 23] are a promising complementary approach as decision diagrams often allow to reduce the required memory by exploiting redundancies in the simulated quantum state.

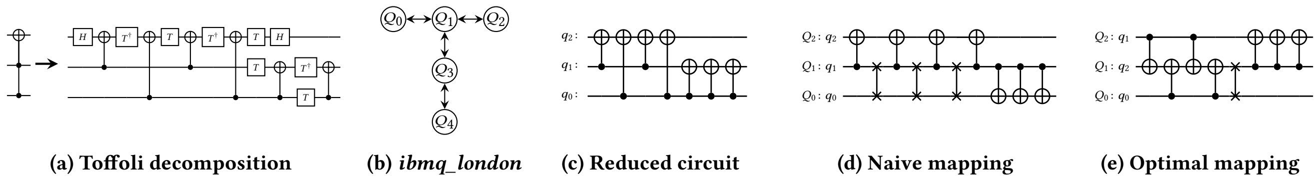
JKQ offers a decision diagram-based quantum circuit simulator (DDSIM) which allows to simulate quantum circuits defined in the REAL [18] or OpenQASM [8] format alongside parameterized Grover's algorithm [10], Shor's algorithm [15], and Quantum Fourier Transformation [14].

Example 5. *Simulating Grover's algorithm with 2 qubits (see Fig. 1) with the JKQ simulator can be conducted as follows:*

```
1  $ ./jkq simulate --simulate_grover 2 \
2    --shots 1000 --ps
3  {
4    "measurements": {
5      "000": 503,
6      "100": 497
7    },
8    "state_vector": [...],
9    "non_zero_entries": 2,
10   "statistics": {
11     "simulation_time": 0.125837,
12     "measurement_time": 0.000180,
```

[1] A matrix U is unitary if and only if $U\,U^\dagger = U^\dagger U = \mathbb{I}$, where $U^\dagger$ denotes the complex conjugate of U and $\mathbb{I}$ denotes the identity matrix.

[2] The building process requires a C++14-compatible compiler, CMake version ≥ 3.10, and the boost (`program_options`) library.



(a) Toffoli decomposition (b) *ibmq_london* (c) Reduced circuit (d) Naive mapping (e) Optimal mapping

Figure 2: Mapping a circuit to a target architecture

```
13      "benchmark": "grover_2",
14      "shots": 1000,
15      "n_qubits": 3,
16      "applied_gates": 16,
17      "max_nodes": 8,
18      "seed": 0
19    }
20  }
```

Here, the parameters define the number of measurements to be performed on the final quantum state (`--shots 1000`) and cause the simulator to output the quantum state vector (`--display_vector`) as well as to print statistics (`--ps`). The output is formatted according to the JSON standard and, hence, easily machine readable.

Simulations of a given REAL or OpenQASM file with the simulator can be started with the parameter `--simulate_file <filename>.<extension>` (the respective format is derived from the extension). The full set of parameters can be listed via `./jkq help simulate`. This includes advanced techniques such as emulation [26], which enable significant speedups for certain quantum algorithms, and weak simulation [12], which more faithfully mimics a physical quantum computer. In the near future, also further improvements, e.g., based on approximation [20] as well as further features such as the consideration of decoherence errors [11] will be supported.

3.3 Compilation

Since superconducting quantum computers in the NISQ era are bound by connectivity constraints, only support a limited set of elementary gates, and are heavily affected by noise, high-level descriptions of quantum algorithms have to be *compiled* through different layers of abstraction before being executable on the actual quantum computer. A major part of this *compilation* flow consists of *mapping*, i.e., making an already decomposed circuit conform to the device's connectivity constraints (which are usually provided as a coupling map).

A circuit is typically *mapped* to the actual device by inserting *SWAP* operations into the circuit—dynamically permuting the location of the logical qubits on the device's physical qubits such that each operation conforms to the coupling map. Due to the inherent influence of noise and short coherence times of today's quantum computers, it is of utmost importance to keep the overhead induced by the mapping procedure as low as possible. As this problem has been shown to be NP-complete [2], there is a high demand for automated and efficient solutions.

JKQ offers quantum mapping (QMAP) tools which satisfy all constraints given by the targeted architecture and, at the same time, aim to keep the overhead in terms of additionally required quantum gates minimal. More precisely, three different approaches based on design automation techniques are provided, which are both generic and can be easily configured for future architectures: The first one is a general solution for arbitrary circuits based on informed-search algorithms [22]. The second one is optimized for a certain set of so-called SU(4) quantum circuits which have been introduced to benchmark compilers [25]. The third one is a solution for obtaining mappings ensuring minimal overhead with respect to SWAP gate insertions [19][3].

Example 6. *Assume we want to perform the computation simulated in Ex. 5 on the "ibmq_london" quantum computer (see Fig. 2b for its coupling map) and assume the Toffoli gate has already been decomposed as shown in Fig. 2a. For the mapping task itself only two-qubit gates matter, as shown in Fig. 2c. While naively executing the task (see Fig. 2d) is not feasible in general, the* JKQ *tool produces results with far less overhead (see Fig. 2e). The mapping task can be conducted as follows:*

```
1  $ ./jkq map --in grover_2.qasm --out grover_2m.qasm \
2    --arch ibmq_london.cm --method heuristic --ps
3  {
4    "circuit": {
5      "name": "grover_2",
6      "n_qubits": 3,
7      "n_gates": 30,
8    },
9    "mapped_circuit": {
10     "name": "grover_2m",
11     "n_qubits": 5,
12     "n_gates": 33,
13   },
14   "statistics": {
15     "mapping_time": 0.001638,
16     "additional_gates": 3,
17     "method": "heuristic",
18     "arch": "ibmq_london.cm"
19   }
20 }
```

Here, the parameters define the input file (`--in`), an output file (`--out`), an architecture to map to (`--arch`), as well as the method to use (`--method`). Several pre-defined architectures for current IBM devices are already available. A complete list as well as detailed information on all available options is available via `./jkq help map`.

3.4 Verification

Compiling quantum algorithms results in different representations of the considered functionality, which significantly differ in their basis operations and structure but are still supposed to be functionally equivalent. Consequently, checking whether the originally intended functionality is indeed maintained throughout all these different abstractions becomes increasingly relevant in order to guarantee an efficient, yet correct design flow. Existing solutions for equivalence checking of quantum circuits suffer from significant shortcomings due to the immense complexity of the underlying problem—which

[3] In order to use this method the Z3 theorem prover library (https://github.com/Z3Prover/z3) needs to be installed.

has been proven to be QMA-complete [13]. However, certain quantum mechanical characteristics provide impressive potential for efficient equivalence checking of quantum circuits.

JKQ offers a quantum circuit equivalence checking (QCEC) tool which explicitly exploits these characteristics based on the ideas outlined in [4–6] and offers a strategy especially suited for verifying compilation results [3].

Example 7. *Verifying the compilation results from Ex. 6 can be conducted as follows:*

```
$ ./jkq verify grover_2.qasm grover_2m.qasm \
--method compilationflow --ps
{
  "circuit1": {
    "name": "grover_2",
    "n_qubits": 3,
    "n_gates": 30
  },
  "circuit2": {
    "name": "grover_2m",
    "n_qubits": 5,
    "n_gates": 33
  },
  "equivalence": "EQ",
  "statistics": {
    "verification_time": 0.000122,
    "max_nodes": 17,
    "method": "Compilation Flow"
  }
}
```

Assuming that an error in the design flow led to a wrong realization `grover_2_error.qasm` *of the desired Grover circuit. Then,* JKQ *also offers a dedicated method for quickly discovering such non-equivalences. This method can be used as follows:*

```
$ ./jkq verify grover_2.qasm grover_2_error.qasm \
--method simulation --ps
{
  "circuit1": {
    "name": "grover_2",
    "n_qubits": 3,
    "n_gates": 30
  },
  "circuit2": {
    "name": "grover_2_error",
    "n_qubits": 5,
    "n_gates": 35
  },
  "equivalence": "NEQ",
  "statistics": {
    "verification_time": 0.008476,
    "max_nodes": 11,
    "method": "Simulation",
    "nsims": 1
  }
}
```

A complete list of the available methods as well as additional configuration options can be listed via `./jkq help verify`.

4 DEVELOPER'S PERSPECTIVE

The tools described above are quite powerful by themselves; nonetheless, there is always room for improvement and further features. Because of this, all *JKQ* tools are open source and developers are kindly invited to extend or modify the methods at their own discretion. This section gives a brief overview of how the tools work internally and serves as a starting point for the interested developer wanting to take a deeper dive.

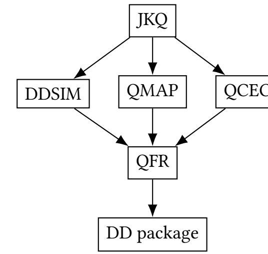


Figure 3: Structure of the JKQ tools

Structurally, the *JKQ* tools are separate applications with a shell script named `jkq` calling the correct application based on the parameters and providing help on the usage. Their dependency relations are illustrated in Fig. 3 and the source code of the individual application can be found on GitHub via https://github.com/iic-jku.

Each application depends on a library referred to as Quantum Functionality Representation (QFR), which handles the input and output of files describing quantum functionalities. Additionally selected quantum algorithms like Grover's search and Shor's algorithm are directly integrated as classes—allowing to programatically construct the respective quantum circuits for simulation, compilation, or verification with parameters controlling, e.g., the number qubits. The QFR itself depends on a package providing the functionality for representing and manipulating quantum states and operations via decision diagrams [21, 24].

Example 8. *Fig. 4 shows the decision diagram of the quantum state* $|\psi'\rangle = [\,1/\sqrt{2}\;0\;0\;1/\sqrt{2}\,]^{\mathrm{T}}$ *from Ex. 2. Following the bold path on the right and multiplying the edge-weights on the way (*1 *weights are omitted), reconstructs the amplitude of the* $|11\rangle$ *state:* $1 \cdot 1/\sqrt{2} \cdot 1 = 1/\sqrt{2}$*. Decision diagrams can also represent matrices, and natively support operations such as addition, multiplication, transposition, tensor product, trace, and fidelity computation.*

The DD package has options that are set during compilation and influence the later execution in simulation and verification. These fall in one of the following categories:

- *Cache sizes:* The underlying routines storing and operating on decision diagrams use different caches for nodes and edges. There is a trade-off between larger cache sizes and better data locality. Developers can adjust these values according to their needs.
- *Floating point representation:* Across all individual projects, the generally used floating point datatype by default is `double` (i.e., 64 bit on most platforms), as defined through the `fp` alias. Depending on the required precision, the developer may change this to `float` (less precision with faster execution) or `long double` (higher precision but slower execution). If the precision is changed, this should be reflected in the `TOLERANCE` (both are defined in `DDcomplex.h`) which mitigates effects caused by the fundamentally limited precision in the representation of complex numbers [21].

Support for additional "hardcoded" algorithms or file formats should be integrated into the QFR, so the tools for simulation, mapping, and verification can access these new features.

The simulator can be used as follows:

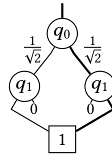


Figure 4: Decision diagram

```
QuantumComputation qc1("shor_115_2.qasm");
QFRSimulator sim(qc1);
sim.Simulate();
auto samples = sim.MeasureAllNonCollapsing(1000);
```

This reads a file `shor_115_2.qasm` into the QFR and uses the resulting `QuantumComputation` object to construct the simulator instance. The actual simulation process is handled in the `Simulate` method, where developers can start optimizing for their specific problem by creating their own simulator. For the simulation of files each instruction in the input file is translated into the corresponding decision diagram (for unitary operations) and applied to the quantum state. Non-unitary operations such as measurements require separate handling in the program. Other paradigms such as *weak simulation* [12] are also supported and easy to extend.

The mapping tool uses configuration files for specifying the coupling maps of different architectures. The file `ibmq_london.cm` for the architecture from Fig. 2b for example looks as follows:

```
0 1 b // connectivity: control target (bidirectional)
1 2 b
1 3 b
3 4 b
```

Developers wanting to use our tool for their applications just need to provide a similar file for the desired architecture.

The equivalence checking methodology described in [3–6] is readily extendable and offers lots of freedom for adapting to specific scenarios. Developers wanting to implement their own equivalence checking strategies can get started at `ImprovedDDEquivalenceChecker.hpp`. There, the `Proportional` strategy for example is realized in the following way:

```
int ratio1 = std::max(1,qc1.getNops()/qc2.getNops());
int ratio2 = std::max(1,qc2.getNops()/qc1.getNops());
while (it1 != end1 && it2 != end2) {
  for (int i=0; i<ratio1 && it1!=end1; ++i, ++it1)
    applyGate(*it1, results.result, perm1, LEFT);
  for (int i=0; i<ratio2 && it2!=end2; ++i, ++it2)
    applyGate(*it2, results.result, perm2, RIGHT);
}
```

In `CompilationFlowEquivalenceChecker.hpp`, the dedicated compilation flow verification strategy can easily be extended to anticipate further optimizations, or adapt to compilation flows different than IBM Qiskit [1].

For more details on the respective implementations, we refer to the documentations in the respective repositories at https://github.com/iic-jku.

5 CONCLUSIONS

This paper provided a brief overview of the *JKQ* design automation tools for quantum computing. We covered the tools from two perspectives: First, for users, who want to solve their problems, but not necessarily develop a deeper understanding of the tools and how they work. Second, for developers, who want to enhance or integrate the tools to tackle their specific problems in quantum computing. We hope that, with these tools, we provide a useful contribution to the growing quantum computing community.

ACKNOWLEDGMENTS

We would like to sincerely thank Alwin Zulehner for his past contributions. This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria as well as by the BMK, BMDW, and the State of Upper Austria in the frame of the COMET program (managed by the FFG).

REFERENCES

[1] Héctor Abraham et al. Qiskit: An open-source framework for quantum computing, 2019.
[2] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. On the complexity of quantum circuit compilation. In *Symp. on Combinatorial Search*, 2018.
[3] 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.
[4] Lukas Burgholzer and Robert Wille. Advanced equivalence checking of quantum circuits. *arXiv:2004.08420*, 2020.
[5] Lukas Burgholzer and Robert Wille. Improved DD-based equivalence checking of quantum circuits. In *Asia and South Pacific Design Automation Conf.*, 2020.
[6] Lukas Burgholzer and Robert Wille. The power of simulation for equivalence checking in quantum computing. In *Design Automation Conf.*, 2020.
[7] Cirq. https://github.com/quantumlib/Cirq, 2020. Accessed: 2020-07-22.
[8] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.
[9] Forest SDK. https://www.rigetti.com/systems, 2020. Accessed: 2020-07-22.
[10] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Theory of computing*, pages 212–219, 1996.
[11] Thomas Grurl, Jürgen Fuß, and Robert Wille. Considering decoherence errors in the simulation of quantum circuits using decision diagrams. In *Int'l Conf. on CAD*, 2020.
[12] Stefan Hillmich, Igor L. Markov, and Robert Wille. Just like the real thing: Fast weak simulation of quantum computation. In *Design Automation Conf.*, 2020.
[13] Dominik Janzing, Pawel Wocjan, and Thomas Beth. "Non-identity check" is QMA-complete. *Int'l Journal of Quantum Information*, 2005.
[14] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[15] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Jour. of Comp.*, 26(5):1484–1509, 1997.
[16] Mathias Soeken, Robert Wille, Oliver Keszocze, D Michael Miller, and Rolf Drechsler. Embedding of large boolean functions for reversible logic. *J. Emerg. Technol. Comput. Syst.*, 12(4):41, 2016.
[17] G. F. Viamontes, I. L. Markov, and J. P. Hayes. *Quantum Circuit Simulation*. Springer, 2009.
[18] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: an online resource for reversible functions and reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 220–225, 2008. RevLib is available at http://www.revlib.org.
[19] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Design Automation Conf.*, 2019.
[20] Alwin Zulehner, Stefan Hillmich, Igor L. Markov, and Robert Wille. Approximation of quantum states using decision diagrams. pages 121–126. IEEE, 2020.
[21] Alwin Zulehner, Stefan Hillmich, and Robert Wille. How to efficiently handle complex values? implementing decision diagrams for quantum computing. In *Int'l Conf. on CAD*, pages 1–7, 2019.
[22] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.
[23] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019.
[24] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 38(5):848–859, 2019.
[25] Alwin Zulehner and Robert Wille. Compiling SU(4) quantum circuits to IBM QX architectures. In *Asia and South Pacific Design Automation Conf.*, pages 185–190, 2019.
[26] Alwin Zulehner and Robert Wille. Matrix-vector vs. matrix-matrix multiplication: Potential in DD-based simulation of quantum computations. In *Design, Automation and Test in Europe*, 2019.