RevKit — User Manual

Version 1.3 – April, 2013
http://www.revkit.org
revkit@informatik.uni-bremen.de

Contents

1. Introduction

RevKit is an open source toolkit aimed to make recent developments in the domain of reversible circuit design accessible to other researchers. Therefore, RevKit provides core functionality (like parsers, export functions, cost calculations, etc.), but also elaborated methods for synthesis, optimization, and verification of reversible (and quantum) circuits. More precisely, the following approaches are available in RevKit.

Synthesis

- A transformation-based synthesis method inspired by the concepts of [7]
- The BDD-based synthesis method as introduced in [11]
- The KFDD-based synthesis method as introduced in [9]
- The heuristic synthesis with output permutation method as introduced in [12]
- The ESOP-based synthesis method inspired by the concepts of [2]
- The exact synthesis method as introduced in [3]
- A Reed Muller Spectra-based synthesis algorithm inspired by the concepts of [5]
- A transposition-based synthesis algorithm

Optimization

- The window optimization method as introduced in [10]
- The circuit line reduction method as introduced in [15]
- The adding lines optimization method as introduced in [8]

Verification

- The SAT-based equivalence checker as introduced in [13]

Further Methods

- A naïve method to embed irreversible functions into reversible ones (needed e.g. to synthesize irreversible functions using the transformation-based method)
- A simple simulation engine (for reversible circuits working on Boolean values)
- A simple decomposition method that maps a given reversible circuit (composed of Toffoli, Fredkin, and Peres gates) to its equivalent quantum circuit (composed of NOT, CNOT, V, and V+ gates) inspired by the concepts of [1] and [4].
- Support of hierarchical circuitry (i.e. modules, flattening of circuits, etc.), sequential circuits, annotations, and more.

- Visualization of circuits.

This document provides a manual describing how to apply the provided approaches and core functionalities of RevKit[1]. The main aspects are thereby kept brief, but are illustrated by means of examples.

In order to invoke approaches, RevKit uses a command line interface enabling an easy and flexible access to the functions and algorithms in the framework. This interface allows to create, modify, and display circuits as well as to call the above mentioned approaches. Additionally, to a certain degree it is also possible to extend the framework with new functionality (even if this is not the main intention of the interface). The interface itself is based on the Python programming language. As a result, commands (or sequences of commands) can easily be specified and processed using a common Python interpreter.

In the following, the usage of RevKit using its command interpreter is described as follows: First, how to get and how to install RevKit is shown, respectively. Section 4 and Section 5 provide a brief introduction into the usage of the basic functionality of RevKit followed by some more advanced examples in Section 6. Finally, the last sections provide the documentation of all data-structures and functions supported by the interpreter so far (including synopsis, examples, etc.).

2. Requirements

RevKit needs a couple of packages which can easily be installed with the distribution's package manager. Below the commands for common distributions are listed. In distributions not listed it can be done analogue but the package names might deviate.

Ubuntu, Mint Linux (for the Python Interface Ubuntu 9.10 or higher is required):

```
> sudo apt-get install build-essential cmake python-dev ipython
  python-qt4 python-numpy
```

openSUSE:

```
> sudo zypper install gcc-c++ cmake python-devel IPython
  python-qt4 python-numpy
```

Fedora:

```
> sudo yum install wget gcc-c++ cmake python-devel ipython PyQt4 numpy
```

[1] For a description of how to extend RevKit with own approaches or how to integrate RevKit in own C++-projects, respectively, we refer the reader to the developers' documentation. The developers' documentation (including an API) is provided by means of doxygen in the sources of RevKit as well as on http://www.revkit.org.

Further packages which are not available in the distrubition's package manager (as e.g. CUDD or PUMA) are downloaded and installed automatically from the bootstrap script. Boost is also required and will be downloaded and installed by default.

For Fedora users: In order to run RevKit, SE Linux needs to be disabled. This can be done temporarily or permanently as follows:

Temporarily:

```
> sudo /usr/sbin/setenforce 0
```

Permanently: Change "enforcing" to "disabled" in "/etc/selinux/config" and reboot.

3. Download and Installation

RevKit can be downloaded from the *www.revkit.org* website. Opening a Bash shell and assuming that the file *revkit-1.3.tar.gz* is in the current working directory, first the package needs to be unpacked:

```
>  tar xvfz revkit-1.3.tar.gz
>  cd revkit-1.3
```

Then, you can build the toolkit. The build process is divided into two scripts. First, the RevKit environment is created using the bootstrap script. Afterwards, the RevKit algorithms are built using the build script. This has the benefit of running the second script only if changes have been made on the algorithms but not on the environment. The bootstrap script should only be called once at the beginning.

More precisely, first run the bootstrap script:

```
> ./make.py bootstrap
```

This will download and compile all dependencies automatically. For that purpose, compiling boost takes some time. If you already have installed boost using the distribution's package manager, the option -DBOOST_PATH can be used to specify its path, e.g. -DBOOST_PATH=/usr. Alternatively you can specify the boost include and libs path separately by using –boost, –boost-include-dir and –boost-lib-dir as arguments. Plaese make sure that your version of boost satisfies the requirements.

After bootstraping the environment, the build script needs to be executed. To run this script manually call:

```
> ./make.py build
```

This will build the whole RevKit suite including the core, algorithms, examples, and the Python bindings. The Python bindings enable the CLI to use RevKit like a shell. If the system cannot build the Python bindings or if they are not needed, they can be deactivated by calling

```
> ./make.py build -DBUILD_BINDINGS=OFF
```

instead.

The build script must be called at least once. Afterwards, the sources only need to be compiled again if local changes have been performed. Alternatively, the program make can be executed manually by calling

```
> make
```

in the build directory.

The build script also provides the options of enabling and disabling the compilation of unstable and example algorithms by using the parameters -DBUILD_UNSTABLE and -DBUILD_EXAMPLES.

4. First Steps Using "Out of the Box"-Tools

After the installation, RevKit is fully functional and ready to use. In this section, we describe how to apply the most important functions using "out of the box"-tools. This should provide a starting point to become familiar with the framework and its functionalities.

4.1. RevKit Graphical User Interface

The *RevKit Graphical User Interface* enables the creation and execution of customized design processes to be executed. Therefore, a GUI is utilized where the respective tasks can easily be put together by means of item blocks connected to a graph. Each item performs an operation and may have ports for the respective input parameters and output results. Input ports can be connected to output ports forming a channel when they support the same data types.

In order to start the RevKit Graphical User Interface, the following command has to be invoked from within the root directory of RevKit:

```
> ./tools/gui/gui.py
```

The use of the GUI is illustrated by means of several tutorial-videos at the *www.revkit.org* website. In the following, we briefly outline the available items.

Sources

- *Path Benchmarks*
  This item opens a set of benchmarks (either Boolean functions provided in *.pla-files, Truth Tables provided in *.spec-files, or Circuits provided in *.real-files[2]) and separately pass them to the succeeding items. This can be applied, if a design process should be applied to a larger set of benchmarks. A right-click on the item opens a file browser, where a path including the respective files can be selected.

- *PLA function*
  This item opens a single function provided in a given *.pla-file. A right-click on the item opens a file browser, where the file can be selected.

- *Truth Table*
  This item opens a single function provided in a given *.spec-file. A right-click on the item opens a file browser, where the file can be selected.

- *Circuit Realization*
  This item opens a single circuit provided in a given *.real-file. A right-click on the item opens a file browser, where the file can be selected.

[2] See *www.revlib.org for a documentation on the respective file formats.*

- *Simulation Pattern*
  This item enables the definition of simulation patterns that should be applied to a circuit. The pattern can be provided manually or in terms a *.sim-file.

Sinks

- *Result Table*
  This item generates a table summarizing the results of an applied process. In particular, this item finds application if more than one benchmark is considered (i.e. in combination with the item "Path Benchmarks"). The result table gets a set of circuits and lists the name of the benchmark, the number of lines, the number of gates, the quantum cost, and the transistor cost of them.

  Using the result table, also different sets of circuits (e.g. obtained by different synthesis approaches) can be compared. Therefore, further item-inputs have to be added by clicking on the button with the "+"-symbol on the left of the first tab in the enlarged item. A double click on the respective tabular enables to name each input individually. Global columns (i.e. table columns which are supposed to be identical for all circuit sets) can be defined at the right-hand side of the enlarged item. Finally, extra columns can be defined using the button named "configure". Here, Python expressions can be defined in order to e.g. automatically compute improvements of certain values (e.g. the number of gates or the quantum cost). The resulting table can be exported either as *.pdf- or as *.tex-file.

  The usage of the result table is explicitly illustrated by means of a tutorial-video at the *www.revkit.org* website.

- *Circuit Viewer*
  This item displays a given circuit.

- *Write Circuits To Path*
  This item dumps a given set of circuits as *.real-files to a given path. In particular, this item finds application if more than one benchmark is considered (i.e. in combination with the item "Path Benchmarks"). A right-click on the item opens a file browser, where the path to which the circuits should be stored can be defined.

- *Write Circuit to File*
  This item dumps a given circuit a *.real-file to a given path. A right-click on the item opens a file browser, where the respective file to which the circuit should be stored can be defined.

Synthesis

- *DD Synthesis*
  This item provides the BDD-based synthesis method as introduced in [11] as well

as the KFDD-based synthesis method as introduced in [9]. The respective synthesis approach can be selected on the right-hand side of the enlarged item. Additionally, the reordering strategy and whether complement edges should be applied or not can be specified. Optionally, the applied DD structure can be displayed. After the item has been processed, the enlarged item reports the run-time needed to perform the synthesis.

- *ESOP Synthesis*
  This item provides the ESOP-based synthesis method inspired by the concepts of [2]. This approach can be configured according to the options summarized in Section C.6. After the item has been processed, the enlarged item reports the run-time needed to perform the synthesis.

- *Transformation-based Synthesis*
  This item provides the transformation-based synthesis method inspired by the concepts of [7] as well as the corresponding synthesis with output permutation method as introduced in [12]. The respective synthesis approach can be selected in the pull-down menu (in case of synthesis with output permutation additionally the optimization criteria can be defined). Furthermore, it can be specified whether bi-directional synthesis should be applied or not. After the item has been processed, the enlarged item reports the run-time needed to perform the synthesis.

- *Exact Synthesis*
  This item provides the exact synthesis method as introduced in [3]. It can be specified whether incremental SAT techniques should be applied or not. Furthermore, the maximum number of gates to be considered can be defined. After the item has been processed, the enlarged item reports the run-time needed to perform the synthesis.

- *Embedding*
  This item provides a simple embedding method. Embedding needs to be processed in order to transform a PLA function into a Truth Table. Optionally, the name of the garbage outputs can be defined. After the item has been processed, the enlarged item reports the run-time needed to perform the embedding.

Optimization

- *Adding Lines Optimization*
  This item provides the adding lines optimization method as introduced in [8]. It requires to define the number of lines that should be added. After the item has been processed, the enlarged item reports the run-time needed to perform the optimization.

- *Line Reduction*
  This item provides the circuit line reduction method as introduced in [15]. This

approach can be configured according to the options summarized in Section D.2. After the item has been processed, the enlarged item reports the run-time needed to perform the optimization.

- *Window Optimization*
  This item provides the window optimization method as introduced in [10]. This approach can be configured according to the options summarized in Section D.1. After the item has been processed, the enlarged item reports the run-time needed to perform the optimization.

Optimization

- *Equivalence Checking*
  This items provide the SAT-based equivalence checker as introduced in [13]. It gets two circuits and returns "equivalent" if both circuits realizing the same function. The equivalence checker supports different configurations of constant inputs and garbage outputs in the considered circuits. After the item has been processed, the enlarged item reports the run-time needed to perform the equivalence check.

- *Sequential Simulation*
  This item provides a simulation engine. It gets a circuit and a Pattern. After the item has been processed, the enlarged item displays a waveform illustrating the simulation. The usage of the simulation item is explicitly illustrated by means of a tutorial-video at the *www.revkit.org* website.

Helper Functions

- *Comparator*
  This item gets two circuits and passes the better one along depending on criteria which can be define in the enlarged item.

4.2. RevKit Viewer

In order to display more complex circuits (e.g. hierarchical circuits), a special GUI called *RevKit Viewer* can be utilized. The RevKit Viewer can be started using

```
> ./tools/viewer.py --filename circuit.real
```

or just by

```
> ./tools/viewer.py
```

without specifying a circuit to open. A circuit can be opened in the GUI using a corresponding menu entry. The user interface of the RevKit Viewer is shown in Figure 1. It shows the circuit specified in *examples/hierarchies.real*.

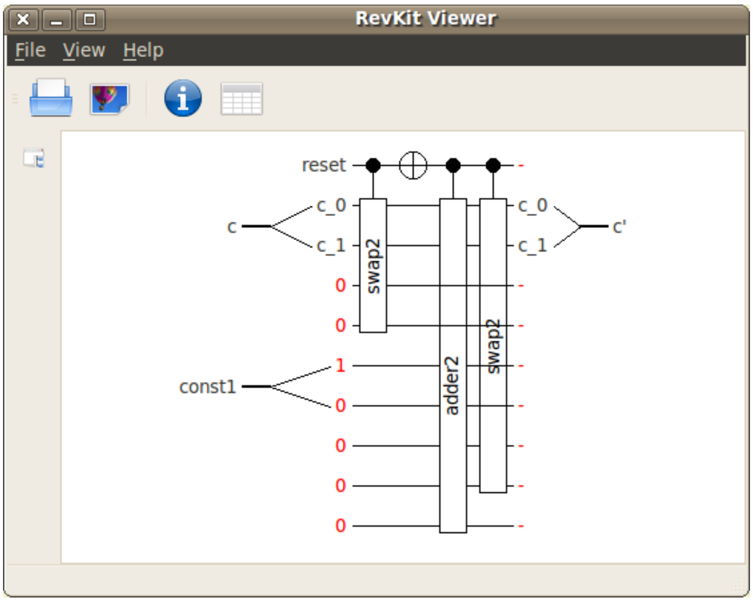


Figure 1: RevKit Viewer

In the following the functionality of the RevKit Viewer is described by outlining its menu actions. For some menu entries corresponding tool buttons in the tool-bar are available.

File ▶ Open Opens a new circuit into the viewer. This replaces an already opened circuit.

File ▶ Save as Image Saves the circuit as a PNG or JPG image.

File ▶ Save as LaTeX Saves the LaTeX code to draw the circuit.

File ▶ Quit Closes the viewer.

View ▶ Circuit details Shows details about the circuit, i.e. different cost metrics.

View ▶ View truth table Calculates and displays the fully specified truth table of the circuit. Depending on the size of the circuit, this can take some time.

View ▶ View partial truth table Calculates and displays the partial truth table, i.e. an optimized truth table omitting constant inputs and garbage outputs.

Help ▶ About Shows information about the viewer.

Zooming into the Circuit

You can zoom into and out of the circuit by placing the mouse over the view area and then move the mouse wheel.

Browsing Hierarchical Circuits

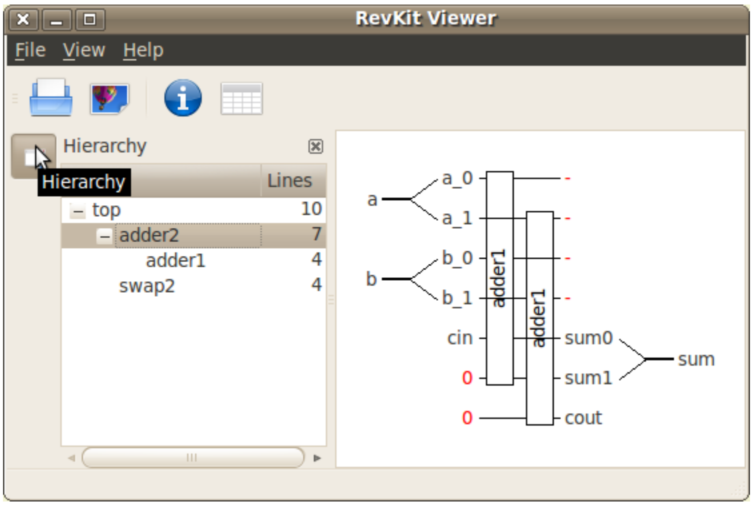


Figure 2: Navigating through modules in the RevKit Viewer

In hierarchical circuits (using RevLib 2.0 modules), the structure of the modules can be displayed by clicking on the *Hierarchy* button on the left tool-bar in the viewer. This opens a dock window containing a tree showing the hierarchy. Double clicking on a hierarchy opens the respective circuit in the view area.

Besides that, modules can also be opened by double clicking on the respective module gate in the circuit.

Displaying Annotations

Annotations are be displayed by means of tool-tips of the respective gate. Simply place the mouse over the gate and the annotation will appear.

LaTeX Export

The RevKit viewer also provides a shortcut for export LaTeX code to produce images of the circuit. Therefore, just click on the view area using the right mouse button. This opens a context menu providing a button for this action. Clicking on this button copies the LaTeX code to the clipboard.

4.3. RevKit Shell Tools

Besides the GUIs, RevKit also provides shell scripts of the most important functionalities (e.g. for the supported synthesis, optimization, or verification approaches). These are written in Python and are available in the homonymous folder *tools*. For example, to apply a given RevLib specification *function.spec* to the exact synthesis approach and to store the result in *circuit.real*, the following command has to be invoked:

```
> ./tools/exact_synthesis.py --filename function.spec --realname circuit.real
```

There are further options which can be passed to the *exact_synthesis.py* tool. They can be listed using the *help* option:

```
> ./tools/exact_synthesis.py --help
```

For each approach implemented in RevKit a corresponding script is available in the *tools* folder. Call them using the *help* option to learn more about their usage.

5. First Steps Using Python-Scripts

To enable command line usage, all functions of RevKit are also exposed as a Python library. In this section, the syntax of that library is illustrated by means of small examples and applications. Using this as a basis, more complex applications can be created in a similar fashion. Section 6 provides more advanced application scenarios which can be realized using RevKit together with Python.

In order to use the RevKit framework in Python, just enter *ipython* followed by

```
from revkit import *
```

into a shell. In the following the usage of the respective RevKit commands is introduced. To get an overview of all available commands, type

```
revkit_commands()
```

into the python shell.

5.1. Creating a circuit

How to create a circuit using RevKit is described by means of *Multiple Control Toffoli gates* (MCT) in the following. Therefore, four steps are performed:

1. Importing the the revkit module,
2. declaring a circuit including 3 lines,
3. adding the respective Toffoli gates to the circuit, and
4. printing the circuit in ASCII format to the standard output[3].

This can be performed using the following Python code:

```
#!/usr/bin/python
from revkit import *

circ = circuit( 3 )

append_toffoli( circ, [2], 1 )
append_toffoli( circ, [0, 1], 2 )
append_toffoli( circ, [1, 2], 0 )
append_toffoli( circ, [0, 1], 2 )
append_toffoli( circ, [2], 1 )

print circ
```

[3]Note that there is a special function print_function, which can be used to print the circuit to the standard output accepting more options to change the appearance. More information can be get from the reference in the remainder of this document

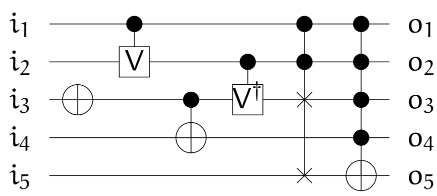
The second parameter of the append_toffoli function gives thereby a list of indices denoting the control line locations, while the last parameter gives the index of the target line. All lines are thereby counted starting with 0, whereby 0 denotes the top line.

5.2. Adding Gates

After getting to know about adding gates in general, in the following example, a circuit is created with different methods, i.e. using different gate types and positions where to insert the corresponding gate. Therefore,

1. An empty circuit with 5 lines is created,
2. Names for the input and output signals of the circuit are set.
3. A CNOT Gate with control at line 2 (counted from 0) and target at line 3 is added,
4. A V Gate (control on 0, target on 1) is prepended (added in the front of the circuit),
5. A Fredkin Gate with controls on 0 and 1 and targets 2 and 4 is appended at the end of the circuit,
6. A V+ Gate is inserted before the second gate (second parameter) with control on 1 and target on 2,
7. A NOT Gate is prepended at the beginning of the circuit,
8. A Toffoli gate with controls on 0, 1, 2, and 3 and target on 4 is added at the end of the circuit,
9. The $\LaTeX\,2_{\varepsilon}$ code for drawing the circuit (using Ti*k*Z) is printed. Thereby the width between gates is adjusted.

This leads to the following circuit:




```
#!/usr/bin/python
from revkit import *

circ  =  circuit ( 5 )
circ .inputs = [ "i_1", "i_2", "i_3", "i_4", "i_5" ]
circ .outputs = [ "o_1", "o_2", "o_3", "o_4", "o_5" ]

```

```
append_cnot( circ, 2, 3 )
prepend_v( circ, 0, 1 )
append_fredkin( circ, [0, 1], 2, 4 )
insert_vplus( circ , 2, 1, 2 )
prepend_not( circ, 2 )
append_toffoli( circ , [0, 1, 2, 3], 4 )

print create_image( circ , elem_width = 0.75 )
```

5.3. Reading and Writing a Circuit from a File

Instead of manually creating circuits, RevKit also supports circuit descriptions given in the RevLib format (see [14] for more information on RevLib and the supported formats). The following example demonstrates how a circuit given in this format can be imported, modified, and finally re-stored in a file. More precisely, the following code shows how

1. an empty circuit is created,
2. the *RevLib* realization file is parsed,
3. the circuit is modified (here, the gates are simply reversed), and
4. the circuit is re-stored to another file.

```
#!/usr/bin/python
from revkit import *

circ = circuit ();
read_realization( circ , "circuit . real")
reverse_circuit ( circ )
write_realization ( circ , "circuit -copy.real")
```

5.4. Iterating through the Gates of a Circuit

Having a circuit available, RevKit provides functions in order to work with it. As an example, the following code shows how to iterate through the gates of circuit using the Python for ... in loop. For each gate the number of its control lines is printed to the standard output. Therefore,

1. an empty circuit is created,
2. a *RevLib* file is parsed,
3. every gate is traversed from left to right, and

4. for each gate the number of its control lines is printed.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
read_realization( circ, "circuit.real")

for g in circ.gates:
    print "Gate has", g.num_controls, "controls."
```

5.5. Calling an approach

Having the basic functionality introduced so far, the main purpose of RevKit is to use the approaches e.g. for synthesis, optimization, verification, etc. This is exemplarily described in the following by means of the transformation based synthesis method (originally introduced in [7]).

In general, all approaches can be invoked using a generic signature of the respective functions. Usually, the first parameter denotes thereby the variable to which the result should be assigned (e.g. in the case of a synthesis approach, a variable representing the generated circuit). The following parameters denote all data, which might be required by the respective approach. In the case of the transformation based synthesis, this is a truth table description of the function to be synthesized. Finally, optional parameters can be delivered. If not, these parameters are initialized with default values. In case of a successful run, the return value of the respective functions is a *dictionary* (Python type dict) containing statistical data collected by the algorithm. Otherwise, the return value is a string containing an error message.

The following code shown the call of the transformation based synthesis with default parameters only.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
spec = binary_truth_table()
read_specification( spec, "function.spec")

transformation_based_synthesis( circ, spec )
```

The function has an optional parameter bidirectional, which enables a special configuration of the approach (see Section C.3 for more details). By default, this option is enabled (i.e. the respective parameter is set to True). However, as the following code shows, this configuration can be easily modified.

```
#!/usr/bin/python
from revkit import *

circ = circuit ()
spec = binary_truth_table ()
read_specification ( spec, 'function.spec' )

transformation_based_synthesis( circ, spec, bidirectional = False )
```

A more detailed documentation of all parameters (also denoted by *settings*) can be found in the last sections of this manual (e.g. in case of the transformation based approach in Section C.3). These parameter always can be applied in every order after the mandatory parameters.

Besides the settings, there are also statistical variables, denoted by *statistics* in the following. In the case of the transformation based algorithm, the only statistical information is the run-time. In the following example, this statistic should be printed after the execution of the approach. The value is thereby assigned to a Python dict variable. It can be accessed by the name of the statistical parameter which are specified in the documentaion for each approach as well. However, in the case the execution of the approach fails, a string containing an error message is returned and, thus, the statistical values cannot be accessed. Thus, we have to check first whether the algorithm succeeded.

```
#!/usr/bin/python
from revkit import *

circ = circuit ()
spec = binary_truth_table ()
read_specification ( spec, "function.spec" )

r = transformation_based_synthesis( circ, spec, bidirectional = False )

if type(r) == dict:     # Success
    print "Runtime", r["runtime"], "seconds."
else:                   # Fail
    print r
```

5.6. Displaying a Circuit

RevKit contains some basic GUI functionality (see also Section 4.2). As described above, a circuit can be printed to the standard output using Python's print or the print_circuit command. Additionally, RevKit provides functions to visualize a circuit. The following example demonstrates how to use the GUI functions. Therefore,

1. a circuit is read from a RevLib realization file,
2. the GUI is initialized (needs to be done only once per session),
3. a window (represented by the variable `w`) displaying the circuit is created and shown, and
4. an input pattern is assigned to the circuit in order to simulate it (in this example, it is assumed that the circuit has three lines).

```
#!/usr/bin/python
from revkit import *

circ = circuit()
read_realization( circ, "circuit.real" )

init_gui()

w = display_circuit( circ )
w.simulate( [1,0,0] )
```

It is possible to zoom in and out into the circuit using the mouse wheel. Further, clicking the right mouse button on the viewer opens a context menu. This menu provides an action to copy the LaTeX code to draw the circuit to the clipboard.

5.7. Miscellaneous

In this section, some tips in the usage of RevKit with the *IPython* interpreter are given. As already mentioned above, all available data structures and commands can be listed by entering

```
revkit_commands()
```

into the Python interpreter. To get the synopsis and some documentation of a command, the name of the command followed by a question mark can be entered, e.g.

```
swop?
```

Entering two question marks will print out the Python source code implementation of the command:

```
swop??
```

6. Advanced Examples

6.1. A Stand-alone Program

This section illustrates how to build a stand-alone program (also denoted as *tool*) using RevKit in combination with Python. As an example, the transformation-based synthesis approach is used.

First, the Python header is set up, and the revkit as well as the sys libraries are loaded. The latter is used for accessing the command line parameters.

```
#!/usr/bin/python
from revkit import *
import sys
```

Note that the sys library is not loaded into the global namespace. Now, the program options are set up. Therefore, different methods are available (for a comprehensive overview, see the reference in the remainder of this user guide). In the following, we need a parameter for providing a specification file to read from and a parameter for providing a realization file to write to. Furthermore, two user defined options are given. The first one is used to enable an ASCII print out and the second one to choose if the bidirectional approach for the transformation based synthesis should be used or not.

```
opts = program_options()
opts.add_read_specification_option() \
    .add_write_realization_option() \
    .add_option( "print", "prints the circuit" ) \
    .add_option( "bidirectional", True, "Bidirectional approach" )
```

The methods for adding program options can be added successively, but note that in the end of each line a backslash has to be written, since there is no end of statement character in Python. The parameter for controlling the bidirectional flag comes with a default value, i.e. True.

After all parameters are set up, they have to be parsed and checked if they are entered correctly. This is done with the methods parse and good, respectively.

```
opts.parse( sys.argv )

if not opts.good():
    print opts
    exit ()
```

The parameter for parse is sys.argv, i.e. the argument values from the command line. It checks, if the names of the parameters have been correctly entered and if values for all mandatory parameters have been provided. If the method good fails, a string for the usage of the program options is printed to the standard output and the program quits.

For the synthesis function, an empty circuit and binary truth table is required. The truth table should be parsed from the given program option.

```
14 circuit circ;
15 binary_truth_table spec;
16
17 read_specification( spec, opts.read_specification_filename() )
```

After that, we are ready to call the synthesis function. The parameter to enable or disable the bidirectional approach is thereby directly taken from the program options using the [] operator. Since we defined a default value for this parameter, it is assured that it yields a valid value. The key is the same string which was given by the call of add_option in line 8.

```
18 res = transformation_based_synthesis( circ, spec, \
19                                        bidirectional = opts["bidirectional"] )
```

The result of the algorithm is saved in the variable res. As mentioned above, if the algorithm failed, res is a string. Otherwise, it is a Python dictionary (dict) with statistical information. Thus, first it is checked, whether the algorithm succeeded. If not, the error message is printed and the program quits.

```
20 if type(r) == str:
21     print r
22     exit()
```

However, if the functional call succeeded, first it should be checked, whether the circuit should be printed to the standard output. This can be controlled using the program option print. Whether a program option is set or not can be checked with the method is_set.

```
23 if opts.is_set( "print" ):
24     print circ
```

Then, it should be checked whether the circuit realization should be dumped to a RevLib file. Since a predefined method of program_options was used to add this option, there exists a predefined option for checking and reading the value of that option as well.

```
25 if opts.is_write_realization_filename_set():
26     write_realization( circ, opts.write_realization_filename() )
```

Finally, statistical information of the circuit as well as of the synthesis process (e.g. the run-time)is printed to the standard output.

```
27 print_statistics( circ, res["runtime"] )
```

A. Core Data Structures

A.1. Gate

The class gate represents a gate in a circuit. It is a collection of control and target lines. Furthermore, a distinct type is set to each gate. Usually, gates are added using helper functions, e.g. append_toffoli.

Constructors

gate() Initializes an empty gate

Properties

controls Line iterator, allows the use of a for ... in loop (*read-only*)

targets Line iterator, allows the use of a for ... in loop (*read-only*)

size Size of the gate, which is the sum of number of control lines and target lines (*read-only*)

num_controls Number of control lines (*read-only*)

num_targets Number of target lines (*read-only*)

type Type of the gate, which can be gate_type.toffoli, gate_type.peres, gate_type.fredkin, gate_type.peres, gate_type.v, gate_type.vplus, and gate_type.module

module_name If the gate is a module, this returns the name of that module (*read-only*)

module_reference If the gate is a module, this returns the circuit it refers to (*read-only*)

Methods

add_control(l) Adds a control at line l

remove_control(l) Removes the control at line l

add_target(l) Adds a target at line l

remove_target(l) Removes the control at line l

A.2. Circuit

A circuit is the central data structure in the RevKit framework. It can be seen as a container of lines. Furthermore, it has properties for meta-data information such as names of the inputs, declaration of constant inputs, etc. A sub-circuit is also a circuit, shares the same data structure, and, thus, the same properties and operations. It is created with the subcircuit constructor.

Constructors

circuit() Initializes an empty circuit with 0 lines

circuit(*n*) Initializes an empty circuit with *n* lines

subcircuit(*base*, *from*, *to*) Initializes a sub-circuit with *base* as circuit basis, including the gates *from* to *to*, where *to* is excluded

subcircuit(*base*, *from*, *to*, *filter*) Initializes a sub-circuit with *base* as circuit basis, including the gates *from* to *to*, where *to* is excluded. Furthermore, the lines are restricted to the indices in the list *filter*

Properties

lines Number of lines

num_gates Number of gates (*read-only*)

gates Gate iterator, allows the use of a for ... in loop (*read-only*)

rgates Reverse gate iterator, allows the use of a for ... in loop (*read-only*)

inputs Input labels

outputs Output labels

constants Determines constant inputs, i.e. a list which assigns the values True, False, or None to each input

garbage Determines garbage outputs, i.e. a list which assigns the values True or False to each output

circuit_name Name of the circuit

filter The filter is a list [*s*,*f*]. In the case the circuit is a sub-circuit and restricted by its lines, then *s* is the number of lines of the base circuit and *f* is the set of lines present in the sub-circuit. Otherwise, *s* is 0 and *f* is empty (*read-only*)

offset In case the circuit is a sub-circuit it returns the offset, i.e. the index of the sub-circuit's first gate in the base circuit (*read-only*)

Methods

append_gate(*g*) Appends gate *g*

prepend_gate(*g*) Prepends gate *g*

insert_gate(*n*, *g*) Inserts gate *g* in front of the gate at position *n*

remove_gate_at(*n*) Removes the gate at position *n*

is_subcircuit() Returns whether circuit is a sub-circuit or not

inputbuses() Returns the input buses of the circuit (as bus_collection)

outputbuses() Returns the output buses of the circuit (as bus_collection)

statesignals() Returns the state signals of the circuit (as bus_collection)

add_module(*name*, *circ*) Adds the circuit *circ* as module named *name* to the circuit. This does not add a gate, but only the reference in the header of the circuit

modules() Returns a dictionary that maps a module name to its reference as circuit

annotation(*g*, *key*, *default_value*) Returns the value of the annotation called *key* of gate *g*. If no such annotation exists, *default_value* is returned instead

annotations(*g*) Returns a dictionary with all annotations, where the name of the annotation (*key*) maps to the value.

annotate(*g*, *key*, *value*) Annotates gate *g* with an annotation called *key* having the value *value*.

[*n*] **Accessor** Gets the *n*-th gate, counting from 0 (*read-only*)

Example

Two different methods for iterating through the gates.

```python
#!/usr/bin/python

circ = circuit()
read_realization( circ, 'circuit.real' )

for g in circ:
  print g.num_controls()

for i in range(0, circ.num_gates):
  print circ[i].num_controls()
```

A.3. Buses

As mentioned in the above section, the buses of a circuit, e.g. the inputbuses, refer to a bus_collection. This data structure handles the creation and the access of the buses and is described in this section.

Methods

find_bus(*line_index*) Returns the name of the bus where the line at *line_index* belongs to, if it belongs to a bus

has_bus(*has_bus*) Returns whether the line at *line_index* belongs to a bus

signal_index(*line_index*) Returns the index of a signal relative to its bus

empty() True, if and only if no bus exists in this collection. . .

[*name*] **Accessor** Returns all signals belonging to the bus with the name *name* (*read-only*)

A.4. Truth Table

As truth table, the user interface of RevKit provides a binary_truth_table containing Boolean values only.

Constructors

binary_truth_table() Initializes an empty binary truth table, working on the values True, False, and None (*don't care*)

Properties

entries Entry iterator, allows the use of a for ... in loop (*read-only*)

num_inputs Number of input variables. The value initially is 0 and is determined after the first call of add_entry() (*read-only*)

num_outputs Number of output variables. The value initially is 0 and is determined after the first call of add_entry() (*read-only*)

permutation Current output permutation, i.e. a list with the numbers from 0 to $n-1$, where n is the number of primary outputs. The permutation can also be changed with permute()

inputs Input labels

outputs Output labels

constants Determines constant inputs, i.e. a list which assigns the values True, False, or None to each input

garbage Determines garbage outputs, i.e. a list which assigns the values True or False to each output

Methods

add_entry(*in*, *out*) Adds an entry with inputs *in* and outputs *out*. The first call determines the number of input and output variables. Afterwards, the size of *in* and *out* must be conform to them

clear() Clears the truth table, including all meta-data and number of inputs and outputs

permute() Permutes the output variables. Returns False when no more new permutation can be set

Example

Iterating through the entries of a specification.

```
#!/usr/bin/python

spec = binary_truth_table()
read_specification( spec, "function.spec" )
```

```
for entry in spec.entries:
    print entry[0], "maps to", entry[1]
```

A.5. Pattern

This class offers read-only access to a simulation file, that can be read with read_pattern.

Constructors

read_pattern() Initializes an empty pattern file

Properties

initializers Returns a dict of initializers, specified by the `.init` command (*read-only*)

inputs Returns a list of input signal names, specified by the `.inputs` command (*read-only*)

patterns Returns a list of input sequences, specified in the simulation file (*read-only*)

B. Core Functions

B.1. Basic Functions

Version

revkit_version()

Returns the current RevKit version as string.

Adding Circuits

append_circuit(*circ*, *src*, *controls* = [])

Appends the circuit *src* to the circuit *circ* controlled by the control lines in *controls*.

prepend_circuit(*circ*, *src*, *controls* = [])

Inserts the circuit *src* at the beginning of circuit *circ* controlled by the control lines in *controls*.

insert_circuit(*circ*, *pos*, *src*, *controls* = [])

Inserts the circuit *src* before gate with index *pos* of the circuit *circ* controlled by the control lines in *controls*. The index is counted from 0.

Adding Gates

append_toffoli(*circ*, *controls*, *target*)

Appends the Toffoli gate with control lines in the list *controls* and target line on *target* to the circuit *circ*.

append_fredkin(*circ*, *controls*, *target1*, *target2*)

Appends the Fredkin gate with control lines *controls* and target lines *target1*, *target2* to the circuit *circ*.

append_peres(*circ*, *control*, *target1*, *target2*)

Appends the Peres gate with control line *control* and target lines *target1*, *target2* to the circuit *circ*.

append_cnot(*circ*, *control*, *target*)

Appends the CNOT gate with control line *control* and target line *target* to the circuit *circ*.

append_not(*circ*, *target*)

Appends the NOT gate with target line *target* to the circuit *circ*.

append_v(*circ*, *control*, *target*)

> Appends the V gate with control line *control* and target line *target* to the circuit *circ*.

append_vplus(*circ*, *control*, *target*)

> Appends the V+ gate with control line *control* and target line *target* to the circuit *circ*.

append_module(*name*, *controls*, *targets*)

> Appends the module gate named *name* with control lines *controls* and target lines *targets*. The module has to be added to the circuit before calling this function.

prepend_toffoli(*circ*, *controls*, *t*

> Prepends the Toffoli gate with control lines *controls* and target line *target* to the circuit *circ*.

prepend_fredkin(*circ*, *controls*, *target1*, *target2*)

> Prepends the Fredkin gate with control lines *controls* and target lines *target1*, *target2* to the circuit *circ*.

prepend_peres(*circ*, *control*, *target1*, *target2*)

> Prepends the Peres gate with control line *control* and target lines *target1*, *target2* to the circuit *circ*.

prepend_cnot(*circ*, *control*, *target*)

> Prepends the CNOT gate with control line *control* and target line *target* to the circuit *circ*.

prepend_not(*circ*, *target*)

> Prepends the NOT gate with target line *target* to the circuit *circ*.

prepend_v(*circ*, *control*, *target*)

> Prepends the V gate with control line *control* and target line *target* to the circuit *circ*.

prepend_vplus(*circ*, *control*, *target*)

> Prepends the V+ gate with control line *control* and target line *target* to the circuit *circ*.

prepend_module(*name*, *controls*, *targets*)

> Prepends the module gate named *name* with control lines *controls* and target lines *targets*. The module has to be added to the circuit before calling this function.

insert_toffoli(*circ*, *n*, *controls*, *t*

Inserts the Toffoli gate with control lines *controls* and target line *target* to the circuit *circ*.

insert_fredkin(*circ*, *pos*, *controls*, *target1*, *target2*)

Inserts the Fredkin gate with control lines *controls* and target lines *target1*, *target2* to the circuit *circ* at position *pos*.

insert_peres(*circ*, *pos*, *control*, *target1*, *target2*)

Inserts the Peres gate with control line *control* and target lines *target1*, *target2* to the circuit *circ* at position *pos*.

insert_cnot(*circ*, *pos*, *control*, *target*)

Inserts the CNOT gate with control line *control* and target line *target* to the circuit *circ* at position *pos*.

insert_not(*circ*, *pos*, *target*)

Inserts the NOT gate with target line *target* to the circuit *circ* at position *pos*.

insert_v(*circ*, *pos*, *control*, *target*)

Inserts the V gate with control line *control* and target line *target* to the circuit *circ* at position *pos*.

insert_vplus(*circ*, *pos*, *control*, *target*)

Inserts the V+ gate with control line *control* and target line *target* to the circuit *circ* at position *pos*.

insert_module(*name*, *pos*, *controls*, *targets*)

Inserts the module gate named *name* with control lines *controls* and target lines *targets* at position *pos*. The module has to be added to the circuit before calling this function.

Circuit Lines

add_line_to_circuit(*circ*, *input*, *output*, *is_control* = None, *is_garbage* = False)

Adds a line to the circuit.

control_lines(*g*)

Returns a list of the control lines of *g*.

target_lines(*g*)

Returns a list of the target lines of *g*.

find_non_empty_lines(*circ_or_gate*, *begin* = None, *end* = None)

Returns the non empty lines in a circuit(range) or gate. The first parameter can be a circuit or a gate. If the first paramater is a circuit, then the gates can be selected by a range from *begin* to *end* (exclusive).

find_empty_lines(*circ_or_gate*, *begin_or_line_size* = None, *end* = None)

Returns the empty lines in a circuit, a circuit range, or a gate. The first parameter can be a circuit or a gate. If the first paramater is a circuit, then the gates can be selected by a range from begin (*begin_or_line_size* parameter) to *end* (exclusive). If the first parameter is a gate then the second parameter is used to specify the number of lines in the gate.

Copying, Modifying and clearing circuits

clear_circuit(*circ*)

Clears the circuit *circ* completely, i.e. gates, lines, and meta-data are deleted. The object is still valid.

circuit_to_truth_table(*circ*, *spec*)

Generates the truth table for the circuit *circ*.

copy_circuit(*src*, *dest*)

Copies all relevant data including lines, gates, and meta-data from *src* to *dest*.

copy_metadata(*base*, *circ*, *copy_inputs* = True, *copy_outputs* = True, *copy_constants* = True, *copy_garbage* = True, *copy_name* = True, *copy_inputbuses* = True, *copy_outputbuses* = True, *copy_statesignals* = True, *copy_modules* = True)

Copies data from a specification or circuit *base* including inputs, outputs, garbage lines and constant lines to the circuit *circ*.

reverse_circuit(*src* [, *dest*])

Reverses the circuit *src* and write the result into *dest*, if given. Otherwise the circuit is reversed in-place.

expand_circuit(*sub*, *circ*)

Expands the sub-circuit *sub* by the lines of its base circuit and copies the result to *circ*.

Truth Table Information and Modification

fully_specified(*spec*)

Returns True, if *spec* is a fully specified truth table. Otherwise False.

extend_truth_table(*spec*)

Removes the Don't Cares Values of a binary truth table *spec*.

Simulation

create_simulation_pattern(*p*, *circ*)

> Creates simulation pattern to be used with sequential_synthesis from simulation file *p* according to circuit *circ*.

Hierarchies and Modules

flatten_circuit(*base*, *circ*)

> Flattens the circuit *base* and stores an equivalent circuit with no modules in *circ*.

circuit_hierarchy(*circ*)

> Returns a hierarchy_tree of the circuit based on the modules, and sub-modules, ...

A hierarchy_tree has the following methods:

Methods

- **root**() Returns the root node of the tree
- **node_name**(*node*) Returns the name of *node*
- **node_circuit**(*node*) Returns the circuit *node* is referring to
- **children**(*node*) Returns the children of *node*
- **parent**(*node*) Returns the parent of *node*
- **size**() Returns the size of the tree, i.e. the number of nodes

B.2. Input/Output

Creating Images

create_image(*circ*, *generator* = create_tikz_settings(), *elem_width* = 0.5, *elem_height* = 0.5, *line_width* = 0.3, *control_radius* = 0.1, *target_radius* = 0.2)

> Creates an image from *circ* and prints out the code to generate the image, e.g. LaTeX. The target code can be specified using the *generator* parameter. In the default case, the output is Ti*k*Z code for LaTeX. Another possible generator is create_pstricks_settings. Furthermore layout options can be specified with the remaining parameters.

Printing a circuit to console

print_circuit(*circ*, *print_inputs_and_outputs* = False, *print_gate_index* = False, *control_char* = '*', *line_char* = '-', *gate_spacing* = 0, *line_spacing* = 0)

> Prints the circuit *circ* as an ASCII representation to the console. The remaining parameters can adjust the appearance.

Printing statistics

print_statistics(*circ*, *runtime* = -1.0, *main_template* = '...', *runtime_template* = '...')

> Prints statistics of *circ* to the console. If *runtime* is not -1.0 it is printed as well. For more information about the templates, we refer to the corresponding entry in the API of the developers' documentation.

Reading and writing circuits and specifications

read_realization(*circ*, *filename*)

> Read-in routine for *RevLib* realization files. The circuit *circ* has to be empty.

write_realization(*circ*, *filename*, *version* = '2.0', *header* = 'This file has been generated using RevKit ... (www.revkit.org)')

> Dumps the circuit *circ* as *RevLib* realization file called *filename*.

read_specification(*spec*, *filename*)

> Read-in routine for *RevLib* specification files. The binary truth table *spec* has to be empty.

write_specification(*spec*, *filename*, *version* = '2.0', *header* = 'This file has been generated using RevKit ... (www.revkit.org)', *output_order* = [])

> Dumps the binary truth table *spec* as *RevLib* specification file called *filename*. Using *output_order* the order of the outputs can be changed. If specified, the length of the list has to match the number of outputs and all indices must be contained in the list.

read_pattern(*p*, *filename*) Read-in routine for a simulation file in *filename* to *p*.

read_pla(*spec*, *filename*, *extend* = True) Read-in routine for *PLA* specification files. The binary truth table *spec* has to be empty. The PLA gets extended using extend_truth_table automatically. This behavior can be disabled by setting extend to False.

write_blif(*circ*, *filename*, *tmp_signal_name* = 'tmp', *blif_mv* = False)

> Dumps the circuit *circ* as *BLIF* circuit to a file called *filename*.

write_verilog(*circ*, *filename*, *propagate_constants* = True)

> Dumps the circuit *circ* as *Verilog* circuit to a file called *filename*. If *propagates_constants* is set to True, all constants signals are omitted in the resulting circuit and evaluated implicitly. Otherwise, for each constant signal a Verilog variable is created.

B.3. Utilities

Cost Functions

costs(*circ*, *cost_function*)

Returns the costs for the circuit *circ* base on the costs function *cost_function*, which can be either gate_costs(), quantum_costs(), transistor_costs(), or line_costs().

Program Options

Constructors

program_options() Initializes an instance of type program_options which has initially only the *help* option.

Methods

add_costs_option() Adds an option *costs* to specify a cost function.

add_read_specification_option() Adds a mandatory option *filename* to specify a *RevLib* specification to read from. If this method was called, add_read_realization_option cannot be called anymore.

add_read_realization_option() Adds a mandatory option *filename* to specify a *RevLib* realization to read from. If this method was called, add_read_specification_option cannot be called anymore.

add_write_realization_option() Adds an option *realname* to specify a *RevLib* realization to read to.

add_numeric_option(*name*, *description*) Adds an option getting a numeric value without a default value having the name *name* and a description *description*.

add_double_option(*name*, *description*) Adds an option getting a floating number value without a default value having the name *name* and a description *description*.

add_option(*name*, *description*) Adds an option getting a string value without a default value having the name *name* and a description *description*.

add_option(*name*, *default_value*, *description*) Adds an option with a default value. The corresponding type can be determined from the default value, which can be either numeric or a string.

costs() Returns the selected costs function, if a respective option was added.

good() Evaluates to True, iff all mandatory options were specified and the *help* option was not requested.

is_set(*name*) Returns True, if the option with name *name* was set as argument.

parse(*arguments*) Parses the program arguments, usually in sys.argv.

read_realization_filename() Value of the *filename* (as realization) option, if specified.

read_specification_filename() Value of the *filename* (as specification) option, if specified.

write_realization_filename() Value of the *realname* option, if specified.

is_write_realization_filename_set() Returns True, iff the *realname* option is set as an argument.

[*name*] **Accessor** Returns the value of the option with name *name*, if specified (*read-only*)

C. Synthesis

C.1. Synthesis with Boolean Decision Diagrams

This algorithm implements the BDD based synthesis approach based on [11]. It supports complemented edges, different re-ordering strategies and the generation of both, Toffoli and elementary quantum gates.

The function representation can be read from a BLIF or PLA file-name. Thereby the extension is used to determine the file type, so it has to be ensured that a BLIF file has the extension *.blif and a PLA file has the extension *.pla, respectively.

Synopsis

```
bdd_synthesis(circ, filename[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

filename A file which contains a function described as PLA or BLIF

Settings for the algorithm:

complemented_edges Specifies whether complemented edges should be used by the BDD. The default value is True.

reordering The reordering strategy for choosing the variable ordering. The default value is 4.

dotfilename If this string is specified, i.e. if it is not empty, then a graph representation of the BDD in DOT format is written to that file-name.

infofilename If this string is specified, i.e. if it is not empty, then information about the generated BDD are dumped to that file-name.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

node_count Number of nodes of the generated BDD

Example

The following example creates a circuit using the BDD synthesis approach and dumps the BDD as a graph. Using dot, the graph can be displayed with the command
`cat /tmp/test.dot | dot -Tpng | display`

```
#!/usr/bin/python
from revkit import *

circ = circuit()
```

```
bdd_synthesis(circ, 'function.pla', dotfilename = '/tmp/test.dot')
```

C.2. Synthesis with Kronecker Functional Decision Diagrams

This synthesis approach constructs KFDDs from a given functional representation in PLA or BLIF and constructs a reversible circuit by constructing cascades for every node type as proposed in [9]. Thereby, re-ordering strategies as well as different decomposition types can be used.

Synopsis

```
kfdd_synthesis(circ, filename[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

filename A file which contains a function described as PLA or BLIF

Settings for the algorithm:

default_decomposition The default decomposition type (Shannon = 0, positive Davio = 1, negative Davio = 2) used when initially constructing the KFDD. The default value is 0.

reordering The reordering strategy for choosing the variable ordering. The default value is 0.

sift_factor Sets a factorial limit for the growth during a siftprocess cause although the outcome will be improved, during sifting the KFDD might explode if not kept at bay. Suggested values are in between 2 and 3. The default value is 2.5

sifting_growth_limit This parameter (possible values are 'r' for relative and 'a' for absolute) determines whether the given sift-factor should be treated as relative or absolute growth limit. In the case of a relative treatment, after each repositioning of a sifting variable the comparison size for the growing is actualized. In the case of an absolute treatment, the comparison size is the intial size of the KFDD for the complete sifting process. The default value is 'a'.

sifting_method Sets the kind of choice for the next sifting candidate. Possible values and their meaning are listed in the following table:

Method	Description
'r' (Random)	The random selection was introduced for comparison reasons.
'i' (Initial)	The sifting variables are chosen in the order given before the sifting procedure starts.
'g' (Greatest)	Chooses the variable in the level with the largest number of nodes.

'l' (Loser first)	Although the complete sifting process will reduce the number of DD-Nodes (or at least keep the same size if no improvement can be done) after each repositioning of a sifting variable there will occasionally be some levels that grow. The loser first strategy chooses the next sifting candidate as the variable in the level with the least increase in size.
'v' (Verify)	Calculates the number of node eliminations due to the reduction rules of OKFDDs if a variable is repositioned in a specific level. It then chooses the best position according to the highest count result.

The default value is 'v'.

dotfilename If this string is specified, i.e. if it is not empty, then a graph representation of the KFDD in DOT format is written to that file-name.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

node_count Number of nodes of the generated BDD

Example

The following example synthesizes a circuit using the KFDD synthesis approach. The negative Davio decomposition is used as the default in the construction process.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
kfdd_synthesis( circ, 'function.pla', default_decomposition = 2 )
```

C.3. Transformation-based Synthesis

This transformation based synthesis algorithm is based on [6]. The idea is to traverse the truth table rows from top to bottom and add gates to the circuit to obtain the identity. In the paper, two strategies were proposed, a basic approach adding gates in the end of the circuit and a bidirectional approach also adding gates in the beginning which can lead to fewer costs. Both approaches are implemented in this algorithm.

Synopsis

```
transformation_based_synthesis(circ, spec[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

spec A fully specified binary truth-table which is basis for the synthesis algorithm

Settings for the algorithm:

- **bidirectional** Determines whether the bidirectional approach should be used or not. The default value is True

Statistical information for the algorithm:

- **runtime** Run-time used by the synthesis algorithm

Example

The following example synthesizes two reversible circuits using the function described in the file *function.spec*. First, the basic approach is applied and afterwards the bidirectional extension is enabled.

```
circ1 = circuit()
circ2 = circuit()

spec = binary_truth_table()

read_specification(spec, 'function.spec')

# bidirectional approach
transformation_based_synthesis(circ1, spec)

# no bidirectional approach
transformation_based_synthesis(circ2, spec, bidirectional = False)
```

C.4. Reed Muller Spectra-based Synthesis

This transformation based synthesis algorithm is based on [5]. The idea is to traverse the truth table rows from top to bottom and add gates to the circuit to obtain the identity. In the paper, two strategies were proposed, a basic approach adding gates in the end of the circuit and a bidirectional approach also adding gates in the beginning which can lead to fewer costs. Both approaches are implemented in this algorithm. The algorithm is very similar to the transformation-based synthesis approach, however, instead of being based on the truth table in internally computes the Reed Muller Spectra and works on that representation.

Synopsis

```
reed_muller_synthesis(circ, spec[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

spec A fully specified binary truth-table which is basis for the synthesis algorithm

Settings for the algorithm:

bidirectional Determines whether the bidirectional approach should be used or not. The default value is True

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

Example

The following example synthesizes two reversible circuits using the function described in the file *function.spec*. First, the basic approach is applied and afterwards the bidirectional extension is enabled.

```
circ1 = circuit()
circ2 = circuit()

spec = binary_truth_table()

read_specification(spec, 'function.spec')

# bidirectional approach
reed_muller_synthesis(circ1, spec)

# no bidirectional approach
reed_muller_synthesis(circ2, spec, bidirectional = False)
```

C.5. Exact Synthesis using Boolean Satisfiability

Synthesizes a minimal circuit (with respect to the number of gates) using the SAT-based exact synthesis approach as presented in [3].

Synopsis

```
exact_synthesis(circ, spec[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

spec A fully specified binary truth-table which is basis for the synthesis algorithm

Settings for the algorithm:

- **solver** The solver to be used in the approach. The default (and currently only available) value is MiniSAT.
- **max_depth** The maximal considered circuit depth. The default value is 20.

Statistical information for the algorithm:

- **runtime** Run-time used by the synthesis algorithm

Example

In the following example, a circuit is synthesized using Boolean Satisfiability.

```
#!/usr/bin/python
from revkit import *

spec = binary_truth_table()
circ = circuit()

read_specification( spec, "function.spec" )
exact_synthesis( circ, spec )
```

C.6. ESOP-based Synthesis

This algorithm implements the ESOP based synthesis approach as introduced in [2]. The basic approach, where each input signal requires to line for its positive and negative polarity version, can be enabled by setting the setting separate_polarities to True. If one line is used for both polarities, which is the default case, a functor can be specified to reorder the cubes in order to minimize inverter gates. Two functors are provided which are, no_reordering which keeps the initial order from the truth table, and weighted_reordering which is proposed in [2] as reordering strategy.

Synopsis

```
esop_synthesis(circ, filename[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

filename A file which contains a function described as ESOP cubes

Settings for the algorithm:

separate_polarities If True, the basic approach using two circuit lines for each input variable is used. Furthermore, in that case, no reordering functor has to be specified. The default value is False.

reordering Function for reordering the cubes to obtain a better result by using less NOT gates. The default value is weighted_reordering with default values.

garbage_name A string for the name of the garbage outputs which are possible created during embedding. The default value is 'g'.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

C.7. Truth Table Embedding

This algorithm takes an irreversible (incompletely) specified truth table, for example using read_pla and embeds it into a reversible specification. Thereby necessary garbage and constant lines are added. The function is always embedded using the 0 values of the constant lines and the method which is used is the "Greedy Method" applying possible assignments by the minimal hamming distance.

Synopsis

```
embed_truth_table(spec, base[, ...])
```

spec A truth table which will be created by this algorithm. Can be the same as base.

base The base truth table which is irreversible.

Settings for the algorithm:

garbage_name A string for the name of the garbage outputs which are possible created during embedding. The default value is 'g'.

output_order The initial has a number of output variables, say n, the initial order of them is $[0, \ldots, n-1]$, i.e. the i-th variable is initially in the i-th column. However, with embedding garbage lines are possibly added, say l garbage lines. Usually, the garbage lines are *appended* to the output columns, i.e. the initial order of the output variables will not change. To change this behavior a list of indices can be specified. The list must have n different elements with the values from 0 to $(n+g-1)$ or the list is empty meaning that the output order will remain the same. The default value is the empty list.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

Example

In this example the AND function is specified manually and then embedded to be reversible. Finally the reversible specification is synthesized using the transformation based synthesis.

```python
#!/usr/bin/python
from revkit import *

spec = binary_truth_table()
spec.add_entry( [False, False], [False] )
spec.add_entry( [False, True], [False] )
spec.add_entry( [True, False], [False] )
```

```
spec.add_entry( [True, True], [True] )

embed_truth_table( spec, spec )

circ = circuit ()

transformation_based_synthesis( circ, spec )
```

C.8. Synthesis with Output Permutation

This is an implementation of the SWOP (Synthesis with Output Permutation) synthesis approach as introduced in [12]. Thereby it is generic and can be used with every truth table based synthesis approach, which gets a circuit and a truth table as parameters.

Synopsis

```
swop(circ, spec[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm

spec A fully specified binary truth-table which is basis for the synthesis algorithm

Settings for the algorithm:

- **enable** This parameter enables the output permutation. Thus, when this parameter is False, the algorithm behaves the same as calling the chosen synthesis algorithm once. Therewith, embedding a synthesis algorithm in the swop algorithm enables three configurations: no swop (enable is False), heuristic and exhaustive (enable is True in combination with the exhaustive parameter). The default value is True.
- **exhaustive** If this parameter is True, then all permutations are checked, otherwise the a good permutations is heuristically determined by sifting the permutations. The complexity of the SWOP algorithm (not considering the used synthesis approach) is $\mathcal{O}(2^n)$ if this parameter is True, and $\mathcal{O}(n^2)$ if this parameter is set to False. The default value is False.
- **synthesis** A functor to the default synthesis approach which is used. The functor is of type truth_table_synthesis_func. The default value is transformation_based_synthesis_func().
- **cost_function** A pointer to a cost function, which is used is criteria to minimize the circuit. The default value is gate_costs().
- **stepfunc** A function which gets called after each iteration of the SWOP algorithm. The functor is of type swop_step_func. The default value is an empty function.

Statistical information for the algorithm:

- **runtime** Run-time used by the synthesis algorithm

Example

In the following example the SWOP synthesis is used with a modified transformation based synthesis (using the non bidirectional approach) and a step function, which counts the number of iterations in a global variable named counter.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
spec = binary_truth_table()

counter = 0

read_specification(spec, 'function.spec')

tbs = transformation_based_synthesis_func(bidirectional = False)

def step():
    global counter
    counter += 1

swop(circ, spec, synthesis = tbs, stepfunc = swop_step_func.from_callable(step))
print counter, 'iterations were performed'
```

C.9. Quantum Decomposition

This algorithm decomposes a reversible circuit into a quantum circuit based on the work of [1] and [4]. The resulting circuits do not necessarily coincide with the quantum costs calculated by `quantum_costs()`, since some further optimizations are not considered yet.

Synopsis

```
quantum_decomposition(circ, base[, ...])
```

circ An empty circuit, which will be filled with quantum gates by the algorithm.

base The base circuit, containing reversible gates which needs to be decomposed. This circuit will not be changed by the algorithm.

Settings for the algorithm:

helper_line_input In some cases a helper line is introduced by the algorithm (see above). This string specifies the input name for the helper line. The default value is 'w'.

helper_line_output In some cases a helper line is introduced by the algorithm (see above). This string specifies the output name for the helper line. The default value is 'w'.

gate_decomposition This parameter is a `gate_decomposition_functor` which decomposes a single gate and adds it to the quantum circuit. This fuctor is called by the algorithm for every gate. The default value is `standard_decomposition`, which implements the above described decomposition techniques.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

Example

The following example decomposes the Toffoli gate as its quantum cascade and writes it to another realization file.

```
#!/usr/bin/python
from revkit import *

circ = circuit (3)
append_toffoli(circ, [0,1], 2)

quancirc = circuit ()
quantum_decomposition(quancirc, circ)
```

```

write_realization(quancirc, 'circuit.real')
```

D. Optimization

D.1. Window Optimization

This algorithm implements the window optimization approach as presented in [10]. The implementation is very generic and depends heavily on the functors defined in settings.

In a loop, a new window is selected using the select_window setting, and in case a window was found, the optimization approach using the optimization setting is applied.

The resulting new window is compared to the extracted one using the cost metric defined in the cost_function setting.

Synopsis

```
window_optimization(circ, base[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm by optimizing base.

base The base circuit which should be optimized.

Settings for the algorithm:

- **select_window** A functor which selects the window which should be considered for local optimization. The default value is shift_window_selection_func with default parameters.
- **optimization** A functor which optimizes the window. The default value is resynthesis_optimization_func with default parameters.
- **cost_function** A pointer to a cost function, which is used is criteria to minimize the circuit. The default value is gate_costs().

Statistical information for the algorithm:

- **runtime** Run-time used by the synthesis algorithm

Example

In this circuit a circuit is read from a realization file and afterwards first optimized using shift window selection with a window length of 7 and an offset of 3. Finally, the circuit is again optimized using the line window selection scheme and quantum costs as cost criteria.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
read_realization(circ, 'circuit.real')
```

```

opt_circ1 = circuit ()
window_optimization(opt_circ1, circ, \
     select_window = shift_window_selection_func(window_length = 7, offset = 3))

opt_circ2 = circuit ()
window_optimization(opt_circ2, opt_circ1, \
     select_window = line_window_selection_func(), cf = quantum_costs())
```

D.2. Line Reduction

This algorithm implements the approach presented in [15]. Windows are found and re-synthesized such that an output of that window is always returning a constant value, so that it can be used as replacement for another constant input line, often introduced by hierarchical synthesis methods.

Synopsis

```
line_reduction(circ, base[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm by optimizing base.

base The base circuit which should be optimized.

Settings for the algorithm:

- **max_window_lines** Number of lines the selected windows can have initially. The default value is 6.
- **max_grow_up_window_lines** When the truth table is not reversible, obtained by a window with max_window_lines lines, then the number of lines can be increased up at most this value. The default value is 9.
- **window_variables_threshold** The possible window inputs are obtained by simulating its *cone of influence*. It is only simulated if the number of its primary inputs is less or equal to this value. The default value is 17.
- **simulation** Simulation function used to simulate values inside the windows and inside the *cone of influence*. The default value is simple_simulation_func().
- **window_synthesis** Functor used to re-synthesize the window. It only has to embed and synthesize the window. It is preferred to use embed_and_synthesize, whereby the parameters can be adjusted to use different synthesis algorithms. The default value is embed_and_synthesize() with default parameters.

Statistical information for the algorithm:

- **runtime** Run-time used by the synthesis algorithm
- **num_considered_windows** Number of windows, which were considered in total.
- **skipped_max_window_lines** Number of skipped windows due to maximum number of allowed primary inputs to be simulated, see window_variables_threshold.
- **skipped_ambiguous_line** Number of skipped windows due to irreversible specification.
- **skipped_no_constant_line** Number of skipped windows in the case that no constant line can be found for a garbage line.
- **skipped_synthesis_failed** Number of skipped windows in the case that the synthesis of the window failed.

Example

First the line are reduced using the standard settings, meaning that the transformation based synthesis is exploited. Afterwards, line reduction is applied using the exact synthesis. To keep the number of window lines small when using the exact synthesis approach, the value for max_grow_up_window_lines is adjusted.

```
#!/usr/bin/python
from revkit import *

circ = circuit()
read_realization(circ, 'circuit.real')

lr_circ1 = circuit()
line_reduction( lr_circ1, circ)

lr_circ2 = circuit()
window_optimization(lr_circ2, circ, \
    max_grow_up_window_lines = 6, \
    window_synthesis = embed_and_synthesize( synthesis = exact_synthesis_func() ) )
```

D.3. Adding Lines Optimization

This algorithm implements the approach presented in [8]. Gates sharing the same subset of control lines are determined with the aim to replace these control lines with an additional line in order to reduce quantum costs.

Synopsis

```
adding_lines(circ, base[, ...])
```

circ An empty circuit, which is filled with gates by the algorithm by optimizing base.

base The base circuit which should be optimized.

Settings for the algorithm:

additional_lines Number of additional lines to be added to the circuit. The default value is 1.

Statistical information for the algorithm:

runtime Run-time used by the synthesis algorithm

Example

In this example, the additional lines optimization approach is applied with two additional lines.

```python
#!/usr/bin/python
from revkit import *

circ = circuit()
read_realization(circ, 'circuit.real')

circ_optimized = circuit()
adding_lines(circ_optimized, circ, additional_lines = 2)
```

E. Version History

- RevKit 1.3 (published April 2013)
  - [C++] The RM Spectra synthesis algorithm introduced in [MDM:07] has been added.
  - [C++] Verification/Simulation: Recursive simulation of modules has been added.
  - [C++] Synthesis: A function transposition_to_circuit has been added which creates a circuit realizing a certain transposition.
  - [C++] Synthesis: A synthesis approach has been added based on consecutive applications of transposition_to_circuit.
  - [C++] New options in write_blif to distinguish state signals and to keep name of constant lines have been added.
  - [Helpers] The helpers scripts are now integrating new algorithms into the Python bindings.
  - [Python] GUI Changes: Snap Items to Items have been added.
  - [Build] The installing and compilation process has been re-organized and unitized (see README for details).
  - [Build] New scripts have been added allowing for the individual compilation of the entire toolkit and its individual algorithms/implementations.
  - [Build] New clean scripts have been added.
  - [Build] A symbol link for python has been added which is used by all Python-scripts.
  - [Build] RevKit is now compatible with the recent boost-library, i.e. compilation errors with (new) Linux distributions have been fixed.
  - [C++] BUGFIX: Equivalence checking is now compatible with new gcc-compilers.
  - [C++] BUGFIX: The order of targets in a Peres gate is now respected.
  - [C++] BUGFIX: The underflow in embed_truth_table with functions that have more outputs than inputs has been fixed.
  - [C++] BUGFIX: Small bugfixes in core/circuit have been performed.
  - [C++] BUGFIX: Several further bugs have been fixed.
  - [Python] BUGFIX: Wrong function name in embed_truth_table tool has been fixed.
- RevKit 1.2.2
  - [C++] BUGFIX: Make .variables optional when parsing *.spec files
  - [C++] BUGFIX: Use generic Python library for building Python bindings

- RevKit 1.2.1
  - [Python] BUGFIX: GUI crash on Ubuntu versions older than 11.04 has been fixed.
- RevKit 1.2 (published May 2011)
  - [Python] The RevKit Graphical User Interface has been added (see User Documentation, Section 3.1 or the tutorial videos at www.revkit.org).
  - [C++] An algorithm for the simulation of sequential circuits has been added.
  - [C++] RevLib 2.0: Support of simulation files.
  - [Python] Extended zooming capabilities have been added (see the status bar of the RevKit Viewer).
  - [Python] The module 'revkitmath' for matrix manipulation has been added.
  - [C++] Support of buses and BlifMV in write_blif.
  - [C++] Bus information can be copied in copy_metadata.
  - [C++] Settings for copy_metadata have been added enabling to select which data should be copied.
  - [C++] Copying of hierarchical information is now configurable in flatten_circuit.
  - [C++] BUGFIX: The properties class (used for algorithms) has been re-implemented (without changing the interface).
  - [C++] BUGFIX: A problem with constant inputs and garbage outputs in the equivalence checker has been fixed.
  - [C++] BUGFIX: The costs calculation for hierarchical circuits has been fixed.
  - [C++] BUGFIX: The problem of too many items in read_pla when there were more than one space between columns has been fixed.
- RevKit 1.1.1 (published February 2011)
  - [Python] BUGFIX: The costs calculation for hierarchical circuits in the RevKit Viewer has been fixed.
  - [C++] BUGFIX: A missing case for Fredkin gate synthesis has been added in write_verilog.
  - [Python] BUGFIX: It is now possible to select a synthesis method in the line_reduction tool script.
  - [C++] BUGFIX: A wrong pathname for testcase has been fixed in the tutorial of the developer's documentation.
  - [C++] BUGFIX: The timeout for the line_reduction synthesis algorithms has been fixed.
  - [C++] BUGFIX: A wrong return value in target_lines and control_lines has been fixed.

- [C++] BUGFIX: Wrong output names and number of output signals in write_blif have been fixed.
- [Python] In KFDD-based synthesis, sifting instead of exact ordering is used as default.
- [C++] BUGFIX: A wrong variable name in testcase script has been fixed.
- [C++] BUGFIX: A missing variable reference in bus_collection has been fixed.
- [C++] BUGFIX: A wrong reference type in python binding for circuit::circuit_name has been fixed.

- Version 1.1 (published December 2010)
  - [C++] The adding lines optimization method introduced in [MWD:2010] has been added (see User Documentation, Section D.3).
  - [C++] The visualization of circuits has been improved (see User Documentation, Section 3.2).
  - [C++] RevLib 2.0: Support of hierarchical circuitry (i.e. modules, flatten_circuit).
  - [C++] RevLib 2.0: Support of input and output buses and state signals.
  - [C++] RevLib 2.0: Support of annotations.
  - [C++] RevLib 2.0: Support of quotes in input and output names.
  - [C++] A new IO-function "write_verilog" has been added which generates a Verilog-Code from a given circuit.
  - [C++] An offset calculation of quantum_costs has been added which can be applied to determine hypothetical costs
  - [C++] An active control concept has been introduced in circuit class.
  - [C++] Gates can be accessed by the index in circuit class.
  - [C++] A return value has been added for the "add_line_to_circuit"-function.
  - [C++] Timer: It is now possible to use system time instead of user time.
  - [Python] A "size()"-method has been added in the bitset class.
  - [C++] BUGFIX: A bug in the quantum cost calculation of Fredkin gates has been fixed.
  - [C++] BUGFIX: Fixed "write_realization" and "write_simulation" to compile in Mac OS.
  - [C++] BUGFIX: Fixed warnings to compile in Mac OS.
  - [Helpers] BUGFIX: Fixed functor name.

- Version 1.0.1 (published October 2010)
  - [Build] BUGFIX Installing python bindings is now possible on 64-bit machines

  – [C++] BUGFIX Bug in quantum decomposition fixed (thanks to Gerhard W. Dueck for reporting this error)

- Version 1.0 (published July 2010)

F. Acknowledgments

We are indebted to the following people for providing tools and/or support, which significantly helped us developing RevKit:

- Aaron Lye: For updating and maintaining the toolkit.
- D. Michael Miller: For his help in implementing the adding lines optimization method and many useful comments.
- Eleonora Schönborn and Bastian Blachetta: For their help in implementing the RevLib simulation file format.
- Fabio Somenzi: Author of the BDD-Package *CUDD*, which is used by RevKit
- Wolfgang Günther: Author of a parser for CUDD, which is used by RevKit
- Niklas Een and Niklas Sörensson: Authors of the SAT solver *MiniSAT*, which is used by RevKit
- Andreas Hett, Harry Hengster, and Bernd Becker: Co-authors of the OKFDD-Package PUMA, which is used by RevKit

References

[1] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *The American Physical Society*, 52:3457–3467, 1995.

[2] K. Fazel, M. Thornton, and J. Rice. ESOP-based Toffoli gate cascade generation. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 206 –209, 2007.

[3] D. Große, R. Wille, G. W. Dueck, and R. Drechsler. Exact multiple control Toffoli network synthesis with SAT techniques. *IEEE Trans. on CAD*, 28(5):703–715, 2009.

[4] D. Maslov and G. Dueck. Improved quantum cost for n-bit Toffoli gates. *Electronics Letters*, 39(25):1790 – 1791, 11 2003.

[5] D. Maslov, G. W. Dueck, and D. M. Miller. Techniques for the synthesis of reversible Toffoli networks. *ACM Trans. on Design Automation of Electronic Systems*, 12(4), 2007.

[6] D. Miller, D. Maslov, and G. Dueck. A transformation based algorithm for reversible logic synthesis. In *Design Automation Conference, 2003. Proceedings*, pages 318 – 323, 2-6 2003.

[7] D. M. Miller, D. Maslov, and G. W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Design Automation Conf.*, pages 318–323, 2003.

[8] D. M. Miller, R. Wille, and R. Drechsler. Reducing reversible circuit cost by adding lines. In *Int'l Symp. on Multi-Valued Logic*, 2010.

[9] M. Soeken, R. Wille, and R. Drechsler. Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition. In *Int'l Design and Test Workshop*, pages 143–148, 2010.

[10] M. Soeken, R. Wille, G. W. Dueck, and R. Drechsler. Window optimization of reversible and quantum circuits. In *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010.

[11] R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *Design Automation Conf.*, pages 270–275, 2009.

[12] R. Wille, D. Große, G. Dueck, and R. Drechsler. Reversible logic synthesis with output permutation. In *VLSI Design*, pages 189–194, 2009.

[13] R. Wille, D. Große, D. M. Miller, and R. Drechsler. Equivalence checking of reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 324–330, 2009.

[14] 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.

[15] R. Wille, M. Soeken, and R. Drechsler. Reducing the number of lines in reversible circuits. In *Design Automation Conf.*, 2010.