CirKit Documentation

Release

Mathias Soeken

Aug 22, 2017

Contents:

CHAPTER 1

Installation

Requirements and dependencies

The following software is required in order to build CirKit

- git
- cmake (at least version 3.0.0)
- g++ (at least version 4.9.0) or clang++ (at least version 3.5.0)
- boost (at least version 1.56.0)
- GNU MP, and its C++ interface GMP++
- GNU readline

Installing dependencies in Ubuntu Linux

In *Ubuntu* the packages can be installed with:

```
sudo apt-get install build-essential git g++ cmake libboost-all-dev libgmp3-dev␣
↪libxml2-dev zlib1g-dev lapack openblas
```

Installing dependencies in Arch Linux

In *arch* the packages can be installed with:

```
sudo pacman -S base-devel git g++ cmake boost boost-libs gmp libxml2 zlib lapack
```

Installing dependencies in Mac OS

In *Mac* it's recommended to use Homebrew to install the required packages:

```
brew install boost cmake gmp readline git lapack openblas
```

Build and run CirKit

After extracting or cloning CirKit perform the following steps:

```
mkdir build
cd build
cmake ..
make external
make cirkit
```

CirKit can be executed with:

```
build/programs/cirkit
```

Build and Run RevKit

After extracting or cloning CirKit perform the following steps:

```
mkdir build
cd build
cmake -Denable_cirkit-addon-reversible=ON -Denable_cirkit-addon-formal=ON ..
make external
make revkit
```

RevKit can be executed with:

```
build/programs/revkit
```

Troubleshooting

No recent cmake version

If for some reason only older versions of `cmake` are available on the system, you can install cmake using `utils/tools.py install cmake` directly from CirKit. Afterwards update the `$PATH` variable by typing `export PATH=<path-to-cirkit>/ext/bin:$PATH`.

Libraries not found

It's best to set the environment variable `CIRKIT_HOME` to the directory of CirKit using:

```
export CIRKIT_HOME=<full-path-to-cirkit>
```

or some other command depending on the user's shell. Also, sometimes depending libraries are not found, then run:

```
export LD_LIBRARY_PATH=$CIRKIT_HOME/ext/lib:$LD_LIBRARY_PATH
```

in Linux or:

```
export DYLD_LIBRARY_PATH=$CIRKIT_HOME/ext/lib:$DYLD_LIBRARY_PATH
```

in Mac OS.

No recent Boost version

There is a manual way to install Boost and CirKit's package manager can help. Before installing CirKit run:

```
mkdir build
./utils/tools.py install boost
cd build
cmake -DBoost_NO_SYSTEM_PATHS=TRUE -DBOOST_ROOT:PATHNAME=`pwd`/tools/boost_1_63_0/ ..
make external
make cirkit
```

Of course, one can add further options to the `cmake` command in the fourth line, e.g., to build RevKit.

CHAPTER 2

Getting Started

Running CirKit

CirKit has a command-line interface. After calling *build/programs/cirkit* a shell prompt is printed to the screen. In order to see all available commands type `help` . This prints a list with all commands together with a short description for each of them. To see more details about a command and its usage call the command together with the `-h` flag. For example `read_aiger -h` prints options to read an AIGER file.

CirKit can be called in three different modes:

1. **Interactive mode:** This is the default interactive mode that is described above.
2. **Bash mode:** `-c` In this mode, commands are given to CirKit as command line arguments, e.g.,:

   ```
   build/programs/cirkit -c "read_aiger file.aig; cone -o y; simulate -at; quit"
   ```

   This example reads an AIGER from *file.aig*, reduces the network to the cone of output *y*, simulates the resulting network as truth table and quits. By adding the flag `-e` additionally each command is printed to the screen before execution, e.g.,:

   ```
   build/programs/cirkit -ec "read_aiger file.aig; cone -o y; simulate -at; quit"
   ```

   Note that single character command options (that start eith with a single `-`) can be concatenated, e.g., `-e -c` can be written as `-ec`.
3. **Batch mode:** `-f` In this mode, commands are read line-by-line from a file, e.g.,:

   ```
   build/programs/cirkit -f command_file
   ```

   This mode also accepts the `-e` flag to print each command before execution. It is possible to comment some commands in the command file by starting a line with a # character.

Some of the main commands are:

Command	Description
`alias`	Creates an alias
`help`	Shows all available commands
`quit`	Quits CirKit
`set`	Sets (internal) environment variables

Stores

Shared data in CirKit such as circuits or truth tables are stored in *stores* and commands can access the data from them. Each data structure has its own store and each store can hold more than one element. For example, there are seperate stores for truth table, AIGs, and BDDs. Call `store -h` to see all available stores. Each store comes with its own *command flag* to access it, e.g., `-a` for AIGs and `-t` for truth tables. Although a store can hold more than one element, it is not necessary and possible to specify which store element to access. Instead each store indivually has a pointer to the *current store element* and commands always access this one. In order to access a different store element, one can change the pointer using the command `current`. For example, `current -a 1` will set the pointer in the AIG store to the element with index 1 (which is the second element in the store).

Example

```
read_aiger file1.aig
read_aiger file2.aig
store --show -a
read_aiger -n file3.aig
store --show -a
current -a 0
store --clear -a
```

First *file1.aig* is read into the AIG store. The second command reads *file2.aig* and **overrides** the current entry. Overriding the current store element is the default behavior of most commands. The current content of a store can be displayed with `store --show -a` or `store -a` as a short-hand. In the third line of the example, one AIG is in the store. When passing the flag `-n` to `read_aiger` as in the fourth command a new entry is added to the store and the current index is updated to the new entry, i.e., at this time the AIG store contains two elements with the current element being the second one (index 1). With `current -a 0` the current index is reset to 0 and `store --clear -a` clears the store from all AIGs.

For many commands it is clear which store they access and it's not necessary to specify the store. There are some generic commands which work on all data structures and require to pass the store access flag, e.g., the command `store`. The generic commands are:

Command	Description
`convert`	Converts store elements into other types, e.g., AIGs to BDDs
`current`	Changes the current store pointer
`print`	Prints a textual ASCII representation of the current store element
`ps`	Prints statistical information about the current store element
`show`	Visualizes the current store element (writes to a *dot* file)
`store`	Shows and clears elements from the store

Logging

Passing `-l file.log` to cirkit creates a log file of the session. This option is particularly useful in batch mode. The log file contains a JSON array with an entry for each command. Each entry contains at least the full command that was run and the time at which the command was started to execute. Some commands write additional data into the log file. For example, `ps -a` writes number of inputs, outputs, and AND gates of an AIG, and `quit` writes several information about the computer on which CirKit has been executed. Being a JSON array, the log file can be easily parsed as many programming languages have a JSON library.

Some helper functions to parse the log file and, e.g., create ASCII tables from them can be found in *utils/experiments.py*. Further, the Python program *utils/extract_script.py* extracts a CirKit script file from the the log that can be run in batch mode. This can be helpful when logging an interactive session and then rerunning the commands:

```
$ cirkit -l session.log
cirkit> read_aiger file.aig
cirkit> ps -a
...
cirkit> quit
$ utils/extract_script.py session.log > session.cs
$ cirkit -f session.cs
```

For performing experimental evaluations, the following workflow is suggested. Create two Python programs (or any other programming language) called *make_script.py* and *make_table.py*. The program *make_script.py* writes a CirKit script. The program *make_table.py* reads the log file created for the script and prints out a table:

```
$ ./make_script.cs experiments.cs
$ cirkit -f experiments.cs -l experiments.log
$ ./make_table.cs experiments.log
```

Aliases

The command `alias` allows to create aliases, which are shortcuts to commands or sequences of commands. The best place for aliass is the init file *alias* located in the directory that is specified in the `$CIRKIT_HOME` environment variable. It is recommended to set `$CIRKIT_HOME` to the root directory of CirKit. Examples for entries in an alias file are::

```
alias e2t "convert --expr_to_tt"
```

The alias command gets two arguments, the *key* and the *value* that is used for substituion. If the key or the value contain a space they need to be put into quotes, and internal quotes need to be escaped.

Note that they key can be any regular expression with capture groups and that the value is a formatted string that can contain placeholders for each capture string: `%1%` for the first capture group, `%2%` for the second one and so on. Note that the `%` sign needs to be escaped. A more complex example is an alias to read a Verilog file into an AIG using ABC::

```
alias "abc_verilog (.*)" "abc -c \"%%read %1%; %%blast\""
```

This will translate, e.g., the command `abc_verilog file.v` into:

```
abc -c "%read file.v; %blast"
```

Since the key is any regular expression, we can create aliases which are very expressive. The alias:

```
alias "(\\w+) > (\\w+)" "convert --%1%_to_%2%"
```

allows, e.g., to convert a truth table into an AIG using `tt > aig`. Putting everything together we can write scripts in CirKit such as:

```
abc_verilog file.v
aig > bdd
bdd -c
```

which reads a Verilog file into a CirKit AIG using ABC's API, then converts the AIG into a BDD and finally computes the characteristic function of the BDD.

Aliases are also useful inside scripts when they are only required locally. Consider, e.g., one wants to convert several truth tables into AIGs, optimize them, and then write them into a file. A script for this task could look as follows:

```
alias "tt_aig_prog ([01]+)" "tt %1%; tt > aig; abc -c &dc2; ps -a; write_aiger %1%.aag"
tt_aig_prog 11101000
tt_aig_prog 01011101
tt_aig_prog 0110
tt_aig_prog 1001100111010111
tt_aig_prog 1101110011000000
```

CHAPTER 3

Data Structures

CirKit (and RevKit) provide the analysis and manipulation of several data structures. These data structures are explained in this section. As described above, instances of these data structures are stored in individual stores. Not all data structures are available in both CirKit and RevKit. The following table gives an overview over the existing data structures, their access option for the store, and their availability in CirKit and RevKit.

Data structure	Access option	CirKit	RevKit
Truth table	`-t --tt`		
Expression	`-e --expr`		
And-inverter graph (AIG)	`-a --aig`		
Majority-inverter graph (MIG)	`-m --mig`		
XOR majority graph (XMG)	`-x --xmg`		
Binary decision diagram (BDD)	`-b --bdd`		
Reversible circuit	`-c --circuit`		
Reversible specification	`-s --spec`		
BDD of a characteristic reversible function (RCBDD)	`-r --rcbdd`		

Truth tables

Truth tables are bitstrings of length $2^k$ and represent Boolean functions over $k$ variables. The most significant bit is the first bit in the bitstring. For example, to load a truth table that represents the AND function $a land b$, type `tt 1000`. We assume that the least significant variable is $a$, then $b$, then $c$, and so on. The truth tables for $a$, $b$, and $c$ are therefore `10`, `1100`, and `11110000`. In order to meet size requirements, truth tables can be extended. If, e.g., `1011` is the current truth table in store, the command `tt -e 3` extends the truth table to be defined over 3 variables, yielding `10111011`.

On can convert truth tables into AIGs using `convert --tt_to_aig`. This will construct an AIG in a very naïve way by constructing each minterm explicitly and then ORing them all. Conversely, one can obtain truth tables from AIGs using simulation. For this purpose use the command `simulate` with the flags `-a` to simulate from AIGs, `-t` to simulate to truth tables, and `-n` to store the simulatuion results. The following example illustrates the usage for the *c17* benchmark from the ISCAS benchmark suite. It also employs NPN canonization on the resulting truth tables using the command `npn`.

Example

```
cirkit> read_aiger c17.aig
cirkit> simulate -atn
[i] G16 : 1011100011111000101110001111100010111000111110001011100011111000␣
↪(B8F8B8F8B8F8B8F8)
[i] G17 : 0011001111111111001100001111000000110011111111110011000011110000␣
↪(33FF30F033FF30F0)
[i] runtime: 0.00 secs
cirkit> store -t
[i] truth tables in store:
     0: 1011100011111000101110001111100010111000111110001011100011111000
  *  1: 0011001111111111001100001111000000110011111111110011000011110000
cirkit> current -t 0
cirkit> npn -t --approach 0
[i] run-time: 0.89 secs
[i] NPN class for 1011100011111000101110001111100010111000111110001011100011111000 is␣
↪0000000000000000000000000000111111110000111100001111111111111111
[i] - phase: 1001010 perm: 5 4 1 3 0 2
cirkit> current -t 1
cirkit> npn -t --approach 0
[i] run-time: 0.89 secs
[i] NPN class for 0011001111111111001100001111000000110011111111110011000011110000 is␣
↪0000000000000000000000001111111100001111000011110000111111111111
[i] - phase: 1001010 perm: 5 0 1 2 4 3
```

The current truth table in the store corresponds to the last output of the AIG. Notice that truth table simulation only scales for AIGs with a small number of inputs. One can obtain a truth table from an expression using `convert --expr_to_tt` or its alias `expr > tt`.

Some truth table related commands are:

Command	Description
`tt`	Load and modify truth tables
`npn`	NPN canonization (exact and heuristic)
`convert --tt_to_aig`, Alias: `tt > aig`	Convert truth table to AIG
`convert --expr_to_tt`, Alias: `expr > tt`	Convert expression to truth table
`simulate -atn`	Simulates AIGs as truth table and stores them
`simulate -mtn`	Simulates MIGs as truth table and stores them

Expressions

Expressions provide an easy way to enter Boolean functions into CirKit. The expressions are multi-level expressions that can contain constants (`0`, `1`), Boolean variables (`a`, `b`, `c`, ...), inversion (`!`), binary AND (`()`), binary OR (`{}`), binary XOR (`[]`), and ternary MAJ (`<>`). The whole syntax is given as follows:

```
expr ::= 0 | 1 | var | ! expr | ( expr expr ) | { expr expr } | [ expr expr ] | <␣
↪expr expr expr >
var ::= a | b | c | ...
```

Note that `a` always corresponds to the least significant bit, `b` to the second least significant bit, and so on. Expressions can be loaded into its store (access flag `-e`) using the command `expr`. Expressions can be used as starting point to create truth tables (`expr > tt`) or binary decision diagrams (`expr > bdd`) for simple functions and avoid to create a file. The following example illustrates its usage.

Example

```
cirkit> expr (ab)
cirkit> expr > tt
cirkit> print -t
1000
cirkit> expr !{ac}
cirkit> expr > tt
cirkit> print -t
00000101
cirkit> expr {{(ab)(ac)}(bc)}
cirkit> expr > tt
cirkit> print -t
11101000
cirkit> expr
cirkit> expr > tt
cirkit> print -t
11101000
```

Note that when loading `!{ac}` the resulting truth table represents a 3-variable Boolean function which does not functially depend on the value for `b`. The last two examples are both Boolean expressions for MAJ, the majority-of-three function.

Some commands related to expressions are:

Command	Description
`expr`	Load expressions
`convert --expr_to_tt`, Alias: `expr > tt`	Convert expression to truth table
`convert --expr_to_bdd`, Alias: `expr > bdd`	Convert expression to binary decision diagram

And-inverter Graphs (AIG)

Loading an AIG into CirKit

There are several ways to load an AIG into CirKit. If the AIG is represented as AIGER file with extension `*.aig` if in binary format and `*.aag` if in ASCII format, one can use the command `read_aiger` to parse the file and create an AIG in the store. If already an AIG is in the store, it will be overriden, unless one calls `read_aiger -n`. If the AIG is represented in Verilog such that ABC's command `%read` is able to parse it, one can use `read_verilog -a` to read the Verilog file, convert it into an AIG and put it into the store. Also BENCH files can be read into AIGs with the command `read_bench`. The command `tt > aig` allows to translate the current truth table into an AIG. Internally, ABC's API will be used for that purpose and the AIG is optimized using `dc2`.

This summary lists commands to load AIGs into CirKit:

Command	Description
`read_aiger`	Read AIG from binary or ASCII AIGER file
`read_verilog -a`	Read AIG from Verilog file (using ABC's `%read` command)
`read_bench`	Read AIG from BENCH file
`convert -tt_to_aig`, Alias: `tt > aig`	Convert truth table to AIG

Manipulating the AIG

ABC is a powerful tool for AIG optimization and manipulation and using the tight integration of CirKit with ABC using the command `abc`, it is very easy to use ABC to optimize AIGs in CirKit directly. Hence, few commands in

CirKit exist to perform AIG optimization, but mainly utility commands.

This list some commands in CirKit to manipulate an AIG:

Command	Description
`cone`	Extracts AIG based on output cones
`cuts -a`	Performs cut enumeration
`propagate`	Propagates constants through inputs
`rename`	Renames inputs and outputs
`shuffle -a`	Shuffles I/O of an AIG
`strash`	Strashes an AIG (removes dangling nodes)
`unate`	Computes unateness properties and functional dependencies of the AIG

Writing an AIG

AIGs can be written into AIGER files using `write_aiger` or into Verilog files using `write_verilog -a`.

This summary lists commands to write AIGs:

Command	Description
`write_aiger`	Write AIG to ASCII file (ASCII if suffix is `.aag`)
`write_verilog -a`	Write AIG to Verilog file

CHAPTER 4

Integration

CirKit offers functions to interact with certain tools. This section shows with which tools CirKit interacts well and gives some illustrative examples.

ABC

CirKit is tightly integrated with ABC. ABC can be accessed as a subshell inside CirKit with the command `abc`. If an AIG is present in CirKit's AIG store, it will be copied to the ABC subshell and available in the &-space (ABC9 commands). Furthermore, when leaving the ABC subshell using `quit` ABC's AIG will be copied back to CirKit and replace the current AIG in the store (unless `abc` is called with the option `-n`). The following example illustrates this interaction in which an AIG is copied to ABC in order to optimize it and then copied back:

```
cirkit> read_aiger c432.aig
cirkit> ps -a
[i]                 c432: i/o =      36 /       7  and =      136  lev =   25
cirkit> abc
UC Berkeley, ABC 1.01 (compiled Apr 22 2016 19:45:32)
abc 01> &ps
c432     : i/o =     36/      7  and =     136  lev =   25 (19.14)  mem = 0.00 MB
abc 01> &dc2
abc 01> &ps
c432     : i/o =     36/      7  and =     123  lev =   25 (19.14)  mem = 0.00 MB
abc 01> quit
cirkit> ps -a
[i]                 c432: i/o =      36 /       7  and =      123  lev =   25
cirkit> quit
```

CHAPTER 5

RevKit

This page lists some RevKit specific documentation. Many parts of this documentation were contributed from Aldo Sayeg.

Input and output

In general, in order to create an instance of a data structure in RevKit we need to load the data structure from a file in a format that describes the data structure. Expressions and truth tables are the only structure that can be created inside of Revkit by using the `expr` and the `tt` command, respectively.

The data structures can often also be written to the same file format. The following table lists all of the data structures' read and write commands used in RevKit along with a short decription.

Data structure	Format	Read command	Write command	Description
Reversible circuits	REAL	`read_real`	`write_real`	Reversible circuit representation using different gates as basis. Part of RevLib supported formats.
Reversible specification	SPEC	`read_spec`	`write_spec`	Truth table of a reversible circuit. Part of RevLib supported formats.
Binary decision diagram	PLA	`read_pla`	`write_pla`	Sum of products representation of a Boolean function.
AND-inverter graph	AIGER	`read_aiger`	`write_aiger`	Format developed for the AIGER utilities.
AND-inverter graph	Verilog	`read_verilog -a`	`write_verilog -a`	Parses whatever can be read with ABC‘s `%read` command.
AND-inverter graph	Bench	`read_bench`		Format developed for traditional circuits as part of ABC.
XOR-majority graph	Verilog	`read_verilog -x`	`write_verilog -x`	Simple single-statement assignment Verilog file.

Adding a command to RevKit

This tutorial explains how to integrate a new simple command into RevKit. As an example, a command called *unopt* is implemented, that copies gate in a reversible circuit without modifying the functionality.

CirKit provides utility scripts in order to create new files easily. We create a file for the new command using the following script:

```
./utils/make_src_file.py cli/commands/unopt reversible
```

Note, that the first argument is the path to the filename without the *src/* in the beginning and without an extension in the end. Two files, a header and a source file, are created. The second parameter ensures that the files are created for the RevKit add-on. The third parameter is optional and can have a name of the file author. If not specified, the name is fetched from the user's git configuration.

The header file in *addons/cirkit-addon-reversible/src/cli/commands/unopt.hpp* contains already some skeleton code and we extend it as follows (all comments are omitted in the code):

```
// unopt.hpp
#ifndef CLI_UNOPT_COMMAND_HPP
#define CLI_UNOPT_COMMAND_HPP

#include <cli/cirkit_command.hpp>

namespace cirkit
{

class unopt_command : public cirkit_command
{
public:
  unopt_command( const environment::ptr& env );
  rules_t validity_rules() const;

protected:
  bool execute();

private:
  unsigned copies = 1u;
};

}

#endif
```

We define a command with the base class `cirkit_command`. It is important that the class name is `unopt_command` to be used in later code that makes use of macros and relies on some naming conventions. Two methods need to be implemented, the constructor that will set up the arguments which can be passed to the command, and `execute` which executes the code and calls our algorithm. We also implement the method `validity_rules` to ensure that the store contains at least one reversible circuit when calling the command. More details on how to write commands can be found in the abc_cli example program.

```
// unopt.cpp
#include "unopt.hpp"

#include <alice/rules.hpp>
#include <cli/reversible_stores.hpp>
#include <core/utils/program_options.hpp>
```

```
#include <reversible/target_tags.hpp>
#include <reversible/functions/copy_metadata.hpp>

namespace cirkit
{

unopt_command::unopt_command( const environment::ptr& env )
  : cirkit_command( env, "unoptimize circuits" )
{
  opts.add_options()
    ( "copies,c", value_with_default( &copies ), "number of gate copies" )
    ;

  add_new_option(); /* adds a flag --new, or -n that can be used to add a new
                       store entry instead of overwriting it */
}

command::rules_t unopt_command::validity_rules() const
{
  return {has_store_element<circuit>( env )};
}

bool unopt_command::execute()
{
  auto& circuits = env->store<circuit>(); /* access store with reversible circuits */

  /* reference to current circuit, and new circuit with same properties */
  const auto& circ = circuits.current();
  circuit circ_new;
  copy_metadata( circ, circ_new );

  for ( const auto& g : circ )     /* iterate through the gates */
  {
    circ_new.append_gate() = g;    /* copy existing gate */
    if ( is_toffoli( g ) )         /* some more copies, if gate is Toffoli */
    {
      for ( auto i = 0u; i < 2u * copies; ++i )
      {
        circ_new.append_gate() = g;
      }
    }
  }

  extend_if_new( circuits ); /* extend store by empty element if --new option is set␣
↪*/
  circuits.current() = circ_new;

  return true; /* always return true */
}

}
```

The function should always return `true`.

We are almost done. Next, we add the command to the RevKit executable. For this purpose, open the file *addons/cirkit-addon-reversible/programs/reversible/revkit.cpp* and add the following header, where other headers are included:

```
#include <cli/commands/unopt.hpp>
```

And then add the command in the same style as other commands are added using:

```
ADD_COMMAND( unopt );
```

That's it. We rebuild RevKit with:

```
make -C build revkit
```

and then call it to try out the new command:

```
revkit> read_spec -p "0 4 2 1 0 3 7 5"
revkit> tbs
[i] run-time: 0.00 secs
revkit> ps -c
Lines:        3
Gates:        7
T-count:      21
Logic qubits: 4
revkit> unopt
revkit> ps -c
Lines:        3
Gates:        21
T-count:      63
Logic qubits: 4
```

Exercises

Here are some suggestions for exercises (with a difficulty estimation from 0–50) to extend the add-on.

1. [25] Copy all gates which are self-inverse in this manner based on a syntactic comparison.

CHAPTER 6

Indices and tables

- genindex
- search