CIRCA Documentation

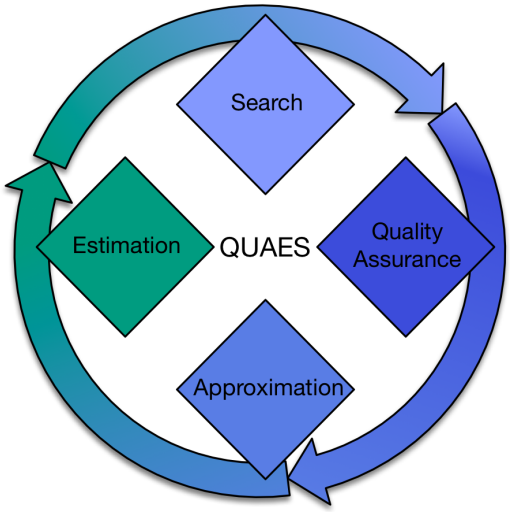
Release 0.9

Linus Witschen

Sep 10, 2019

Contents:

Note: This is a WIP Beta version. All content is subject to change.



CHAPTER 1

About CIRCA

CIRCA is a framework for the automatic generation of approximate circuits. CIRCA is designed to be a flexible yet consistent framework in which the processing tasks in an approximation process are clearly separated from each other. In this way, CIRCA provides a general and modular design, facilitates the implementation of new functionality, and fosters comparative studies as well as the evaluation of new techniques against existing ones.

CIRCA comprises of three stages: the *input* stage, the *QUAES* stage, and the *output* stage. The *input* and the *output* stage frame the *QUAES* stage - the main approximation process - to ensure compatibility to tools invoked before or after the approximation process. An interface for external tools provides compatibility to other tools during the approximation process, e.g. *Yosys*.

The *QUAES* stage targets search-based approximation processes. Thus, we modeled the approximation process as an iterative search problem. The *QUAES* stage, i.e., the approximation process itself, comprises four processing blocks: **Qu***ality Assurance*, **A***pproximation*, **E***stimation*, and **S***earch Space Exploration*. Each block fulfills one specific task and is communicating with other blocks via well-defined interfaces. The employed functionality of each stage or block during an approximation process is configured using a *configuration file*. The configuration file allows the swift exchange of functionality and enables the comparison of different approximation techniques, search methods, or validation techniques against each other, encouraging the Approximate Computing community to compare their approaches fairly against each other.

Note: This is a WIP Beta version. All content is subject to change.

1.1 Getting Started with CIRCA

1.1.1 Introduction[1]

CIRCA's architecture is shown in the *overview* figure. We target search-based approximation approaches with different approximation techniques, and formal verification as well as testing for circuit validation. The framework is divided

[1] From L. Witschen, T. Wiersema, H. Ghasemzadeh Mohammadi, M. Awais and M. Platzner, "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation," Presented at the Third Workshop on Approximate Computing (AxC '18) 2018, Bremen, Germany, 2018.

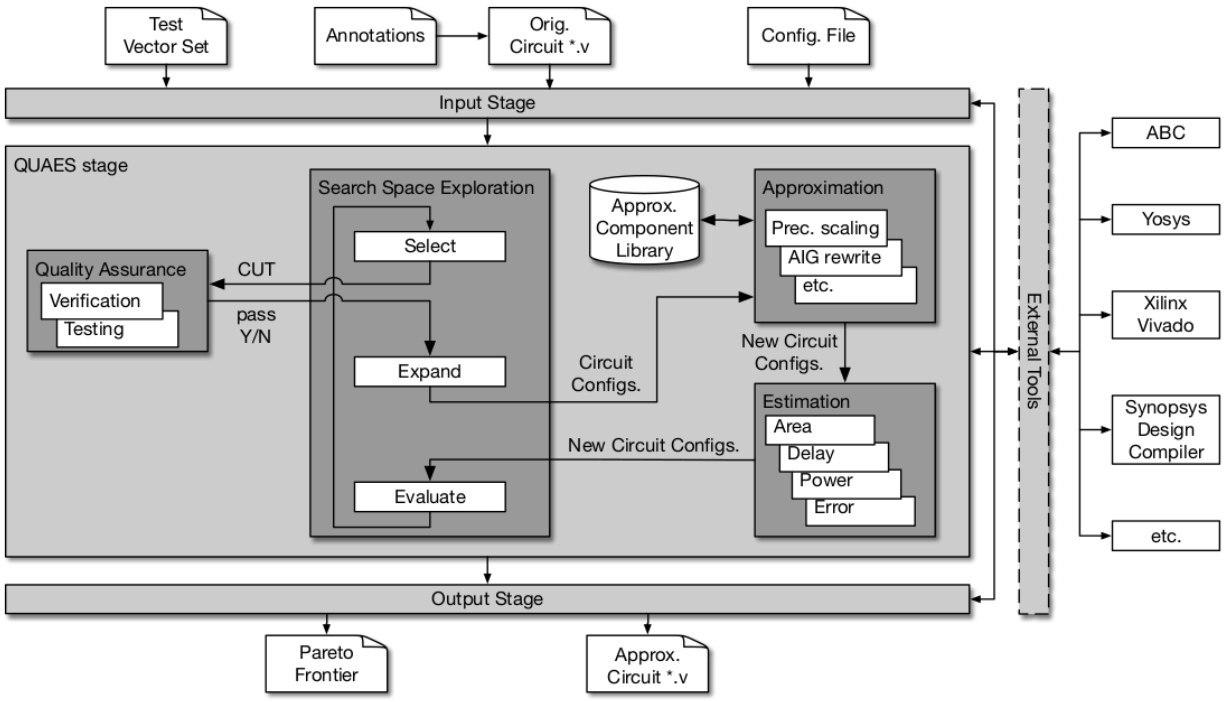


Fig. 1: Overview of CIRCA's components and structure

into three stages: the *input* stage, the *QUAES* stage, and the *output* stage. The input stage processes user-provided information, which in any case comprises a *configuration file* and a description of the original, exact circuit. The configuration file defines the employed functionality in the *QUAES* stage and specifies the target metric as well as the quality constraints for the approximated circuits. The circuit description may contain annotations, added, for example, by the user to mark subcircuits amenable to approximations. Since our framework supports also testing as quality assurance technique, the input can include a test vector set.

The main stage for generating approximate circuits is the QUAES stage. The split of the corresponding functionality into the four processing blocks **Qu***ality Assurance*, **A***pproximation*, **E***stimation* and **S***earch Space Exploration* (**QUAES**) is key to achieving a modular and extensible architecture. We denote the subcircuits in the input Verilog code subject to approximations as *candidates*, and the different approximated versions of these candidates as *variants*. The overall circuit with instantiated variants for the candidates is called a *circuit configuration*.

The structure of the search space exploration is kept rather general, and thus, sufficiently flexible to support a wide range of search algorithms. The search space is iteratively explored by creating and visiting configurations, which form the nodes of the search space, by applying the three steps *Select*, *Expand*, and *Evaluate*.

The evaluation step takes a set of configurations and determines estimated values for parameters of interest. Typically these parameters cover metrics related to area, delay, and power but can also include estimates of the error metrics. The actual estimation functions are encapsulated in the estimation block to clearly separate estimation from the search space exploration.

The select step receives a set of configurations, each with an annotated vector of estimated parameters, and selects the next configuration to be further considered, i.e., to be expanded. This selection relies on a search heuristic. The selected configuration is subsequently validated. To this end the configuration, now called *Circuit-under-Test* (CUT), is sent to the quality assurance block. This block either employs formal verification or testing to decide if the CUT satisfies the quality constraints and passes the check or not. If a CUT fails quality assurance, the select step will, depending on the configuration file, either abort the search or pick the next best configuration for validation. In the latter case, the search terminates if there are no more valid configurations.

If a CUT passes validation, the configuration and thus the search space is expanded by creating a number of new configurations. For creating new configurations the approximation block applies certain approximation techniques. Depending on the actual configuration of the framework, one or more candidates of the configuration can be approximated and one or more approximation techniques can be applied.

The approximation block accesses a library of approximated subcircuits, which is beneficial for two reasons: First, it is rather likely that one circuit component will be approximated for several times. This happens when the overall circuit contains identical candidates, e.g., multiple occurrences of an 8-bit unsigned adder. Storing the approximated versions of such components and retrieving them the next time can greatly save computations. Second, there are already libraries available of approximated components which can then be leveraged by CIRCA.

During the QUAES stage, all validated configurations are stored. The output stage performs post-processing on these configurations to return either the best approximated circuit, e.g., a circuit that respects the error constraints and has minimal area or energy, or a Pareto-filtered set of circuits.

1.1.2 Installation

Prerequisites

- Have Python 3.6.0 or newer installed
- Whatever ABC and Yosys require (see respective website)
- Go to `circa` base directory

On first installation

- Compile ABC (for troubleshooting see: ABC documentation or ABC GitHub):

```
cd abc/
make -j 8
cd ..
```

- Compile Yosys (for troubleshooting see: Yosys website or Yosys GitHub):

```
cd yosys/
make -j 8
cd ..
```

Before each terminal session

Source CIRCA settings to set up CIRCA's environment:

```
cd circa/
source settings.sh
cd ..
```

Try out demo

(Not required for installation)

Small sequential circuit with two modules as approximation candidates and a configuration file that sets the approximation method to precision scaling:

```
cd demo/anno_gradient/
# Calling CIRCA: circa <cfg. file> <input design *.v>
circa example.cfg test_seq.v
```

1.1.3 Next Steps

Now that you have successfully installed and tested CIRCA, you can continue by reading the *user guide* to learn how to use CIRCA to generate approximate circuits yourself, or you can go on to the *developer guide* and learn everything you need to design and implement your own CIRCA components.

For *more information*, visit the CIRCA website or git repository.

1.1.4 Licensing

CIRCA is free software licensed under the MIT license.

1.1.5 Citation

Please cite the following reference when publishing results obtained with CIRCA:

L. Witschen, M. Awais, H. Ghasemzadeh Mohammadi, T. Wiersema and M. Platzner, CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation," Microelectronics Reliability, (99):277-290, Elsevier, August 2019.

Note: This is a WIP Beta version. All content is subject to change.

1.2 User Guide

This section will teach you how to use CIRCA for approximate circuit generation from a user's perspective. You will learn to configure and run CIRCA's approximation process in the following sections:

- *Basic usage*
- *Configuring CIRCA*
- *Built-in component implementations*
- *Example configuration*

1.2.1 Basic usage

Note: Make sure you run `source circa/settings.sh` before each terminal session.

After installation, you can run the `circa` command from any directory you like. Simply type

```
circa your_config_file.cfg your_input_design.v
```

into the console and the approximation process will start, generating logs and output files along the way. (*Note that the input design's directory is usually used as a base directory for output generated during CIRCA's approximation process.*)

The command takes two arguments: the configuration file followed by the input design. The config. file defines the particular approximation process. In general, the config. file contains information about the input design and specifies the employed implementation in a stage or processing block, respectively, and its parameter set. To edit the config., it is generally a good idea to start with a copy of an existing config. file (for example, the file provided in the demo) and only edit the settings you want to change, leaving the rest of the file unchanged. The following sections will give you an in-depth overview of *what* you can configure in the config. file and how it will affect the framework's behavior. To learn about the config.'s structure and *how* to edit the settings, go to the *configuration chapter*.

In the current implementation, the input design argument is a Verilog description of the circuit you want to approximate (note that CIRCA is not limited to only Verilog, it is up to now just the format we use). It must have a top-level module whose name must be specified in the configuration file. In general, the input design must comply to the standard Verilog syntax. Some FrontEnd implementations, for example, state exceptions to this rule, since they may support or even require code annotations, violating the Verilog standard. Such exceptions, however, will always be described in the documentation to make the users aware.

Some implementations also may require certain limitations of the input design regarding language constructs which cannot be processed (see *Standard FrontEnd* as an example).

1.2.2 Configuring CIRCA

As mentioned in the *Getting Started* section, CIRCA distributes the approximation process on several independent and interchangeable components which form the framework's three stages *input* (or *FrontEnd*), *output*, and *QUAES*. While the *input* and *output* stages are each handled by a single component, the *QUAES* stage comprises of four components, one for each processing block (*Quality Assurance*, *Approximation*, *Estimation*, and *Search*).

Using the configuration file, each method employed in these six components can easily be parameterized or exchanged. In general, each method provides its own set of parameters, which is usually not shared among different methods. The following sections will describe universally shared parameters and the parameters of the built-in methods.

Approximation candidate settings

The approximation process heavily relies on the detection and processing of *approximation candidates*, the parts of the input circuit subject to approximations. The configuration of the processing of the candidates impacts the quality of the approximated outcome directly. By default, there are two parameters which determine how the approximation of a candidate is carried out: the *approximation method* and the *quality constraints*.

The *approximation method* specifies the algorithm or technique that is used to perform the approximation in the *Approximation* block. For example, one of CIRCA's standard approximation methods is *precision scaling* (*PS*). It approximates a candidate by replacing bits of the output vector with constant logic 1s or 0s and thereby rendering all logic behind the replaced bits obsolete. A synthesis tool is then able to optimize the resulting circuit and reduce its area, delay, power consumption etc. If multiple approximation methods are assigned to a candidate, they will be handled separately to make the approximation results of different methods comparable.

The *quality constraints* assigned to a candidate specify a boundary for the approximation methods. We denote quality constraints assigned to a candidate as *local quality constraints* - or the reciprocal, *local error bounds*. These local quality constraints allow CIRCA to control the quality of a candidate, or more precisely, its particular variant. Being able to control the quality of a variant, enables CIRCA to expand the search space in a controlled manner. The local error bounds of a candidate represent the maximal accepted error of a candidate. The local error bounds applied to the candidate's variants are in the range of no error (representing the original implementation of the candidate) and the local error bounds of the candidate. For example, a valid local worst-case error bound for a candidate with an 8-bit

output (representing unsigned integer values) would be *255* since this expresses the maximal possible output value of the candidate. Any error bound larger than *255* would be meaningless.

Quality can be measured in several different *error metrics*. Standard error metrics of CIRCA are, for example, *worst-case* and *bit-flip* errors (*WC* and *BF*). The worst-case error of an approximated variant is the maximum numerical difference between its output and the output of the original variant, which is always used as the point of reference. The bit-flip error is the maximum Hamming distance between the original and the approximated output vectors. To be considered valid, an approximated variant must satisfy all quality constraints specified for the candidate.

The approximation process works by gradually increasing the local error bounds, i.e., the error bounds of each candidate. The rate at which the error bounds are increased can be specified via a `step` parameter. The bounds will not be increased beyond the maximum error value, specified by the `bound` parameter.

There are many different ways to specify approximation methods and quality constraints for the candidates. They depend on the used component implementations and may include default settings for all candidates, specific settings for a candidate provided via annotations in the input design, and global quality constraints assigned to the whole circuit.

Working with the configuration file

The configuration file is a simple text file which is divided into *sections*. To parse a config. file, CIRCA employs Python's configparser module, which also dictates its structure. Each section has a section header, indicated by square brackets encapsulating the name of the section `[section name]`. The section header is followed by a list of *options* and their *values*, which are assigned using a =. The file can be commented by using a # to make the configuration parser ignore the rest of the line. A simple example for the configuration file syntax would be

```
[Section1]
Option1 = Value1
Option2 = Value2_1, Value2_2

[Section2]
# This section has no options

# [Section3 (will be ignored)]
```

The option values are arbitrary strings, but for most options, only certain values can be interpreted. The set of possible option values is documented in the component implementation's documentation or directly in the configuration file.

Configuration sections

CIRCA's configuration file has seven sections, each having one mandatory option by default.

The section `General` is used for providing information about the input design that is not specific to one of the components or cannot be acquired from the input design automatically (yet). The section contains the mandatory option `TopModule` to which the name of the input design's top-level module must be assigned.

The other six sections, named `Input`, `Search`, `Estimation`, `QualityAssurance`, `Approximation`, and `Output`, each refer to one of the stages or processing blocks. For each of these sections, the parameter `Method` has to be defined. The value of this parameter determines the employed method for the corresponding stage or processing block, respectively. Additional parameters may follow the `Method` parameter to parameterize the employed method further. Examples for additional options, their possible values, and their effects can be found in the *next section*.

Standardized quality constraint syntax

CIRCA defines a standardized syntax for the specification of quality constraints and approximation methods which is used by the built-in component implementations and recommended to be used for custom implementations as well. This syntax specification applies to the values of the `QualityConstraints` options of the sections `[QualityAssurance]` and `[Approximation]` in the configuration file and to the candidate annotations in the input design if the built-in implementations for the *QualityAssurance*, *Approximation* or *Input* component `AnnotatedCandidates` are used.

The general syntax is a simple list of elements separated by `;`, like for examle

```
element1;element2;element3
```

Whitespace before and after the `;` is ignored.

Each element can be a specifier for either an *error metric* or an *approximation method*. Error metric specifiers have the format

```
<ID>:bound=<bound>,step=<step>
```

where `<ID>` is the unique identifier of the error metric, `<bound>` is the maximum upper bound for this metric and `<step>` is the step size, i.e., the amount by which the error bound is increased in one step. Note that both the `bound` and `step` parameters are not always required (e.g. the `[QualityAssurance]` section does not need a step size) and can be omitted in such a case. Also, their order can be changed and whitespace around the `:`, = and `,` is ignored. Multiple occurrences of a parameter will at least generate a warning message and may abort the approximation process.

Example error metric specifier:

```
WC:bound=16,step=2
```

This expression specifies a worst-case error with a maximum upper bound of 16 and upper bound increment of 2 in each step.

Approximation method specifiers have the format

```
AppMethods:<ID1>,<ID2>,<ID3>
```

where `<ID1>`, `<ID2>` and `<ID3>` are unique identifiers of approximation methods. Approximation method specifiers are usually only found in candidate annotations.

Example approximation method specifier:

```
AppMethods:PS,AIG
```

This expression specifies the approximation methods *Precision Scaling* and *AIG-Rewriting*.

Here are a few examples for complete valid quality constraint specification expressions:

```
# Bit-flip error with max. upper bound 4 and default step size,
# worst-case error with auto-detected maximum bound and step size 5
# and only precision scaling as approximation method
BF:bound=4;WC:step=5;AppMethods:PS

# Worst-case error with max. bound 10 and step size 1
# and AIG-Rewriting and Precision Scaling as approx. methods
WC:step=1,bound=10;AppMethods:AIG,PS

# Approximation method AIG-Rewriting
# and error metrics worst-case and bit-flip,
```

(continues on next page)

(continued from previous page)

```
# both with default step size and maximum upper bound
AppMethods:AIG;WC;BF
```

1.2.3 Built-in component implementations

The following list leads to a documentation of CIRCA's six, method-specific config. sections. For each section, you can find a brief description of the component's tasks, the currently implemented methods, and the documentation of the available parameters for each method.

Input component

- **Input component**
- *Quality assurance component*
- *Approximation component*
- *Estimation component*
- *Search component*
- *Output*

Component description

The *input* component handles primarily the detection of approximation candidates, initialization of data structures, and generation of files which are later used by the other stages. It reads the input design and prepares the file for approximation, for example, by removing FrontEnd-specific annotations.

The standard FrontEnd implementation is called `AnnotatedCandidates`, which is also the value to assign to the `Input` section's `Method` option to use this implementation.

AnnotatedCandidates

The `AnnotatedCandidates` FrontEnd relies on annotations in the Verilog input design, marking Verilog modules as approximation candidates. To mark a module for approximation, a `Key`, specified in the config. file, must be inserted in the input file between the module keyword and the module name. Additional parameters, following the `Key` and framed by <<< and >>>, can be used to specify the applied approximation methods and quality constraints for the particular module or candidate. The additional parameter annotation uses the *default CIRCA parameter syntax*.

Example module definition with `Key = <<<APPROX_CAND>>>`:

```
module <<<APPROX_CAND>>>:<<WC:bound=20,step=2;appMethods:PS>>> Adder(in,
↪cout);
// Module content...
endmodule
```

Configuration options

Method AnnotatedCandidates

Key The keyword used to mark a module as approximation candidate. Can have any string value which does not appear anywhere else in the input design file. Recommended value is `<<<APPROX_CAND>>>`.

Notes and restrictions

- Candidates are generated from *Verilog modules*, not *instances*. Thus, parameters set for specific instances of the Verilog module are *ignored*. You may use parameters in the module; however, do not change these parameters for the instances.
- The size of all primary input and output signals of the top-level module of the circuit and all marked modules must be an *absolute literal integer*, i.e., the size must be 1 or specified as `[msb:lsb]` where `msb` and `lsb` are integer numbers, *not parameters, localparams, or expressions* which resolve to an integer. Example:

```
input clk;          // size 1, ok
input [3:0] in1;    // size 4, ok
input [4-1:0] in2; // expression, not ok
parameter W = 3;
output [W:0] out;   // parameter, not ok
```

Back to the User Guide

Quality assurance component

- *Input component*
- **QualityAssurance component**
- *Approximation component*
- *Estimation component*
- *Search component*
- *Output*

Component description

The *QualityAssurance* component is responsible for the validation of approximated circuits, i.e., checking if the circuits resulting from the approximation process satisfy the specified quality constraints. Generally, we distinguish between testing-based approaches and formal approaches. Testing-based approaches evaluate the quality of a circuit by applying a set of input vectors (or input sequences for sequential circuits). However, testing-based approaches are usually not able to adopt all possible input vectors; thus, the error of a circuit (or its quality) can only be estimated. Formal verification methods, on the other side, are able to provide a mathematical prove to guarantee that the circuit adheres to the specified error bounds. This guarantee, however, often comes at costs of higher runtimes.

CIRCA supports testing-based approaches as well as formal methods. Currently, however, only formal methods are provided: `abc_dprove`, `abc_pdr`, and `yosys_ic3`. The three approaches utilize the inductive solvers of the external programs ABC and Yosys.

abc_dprove, abc_pdr, and yosys_ic3

CIRCA's standard quality assurance implementations proceed in the same way to verify the quality of a circuit: they generate a so-called *sequential quality constraint circuit* (*SQCC*) (see *figure below (left)*) and apply a formal verification method on it, i.e., ABC's dprove, ABC's PDR, or Yosys' IC3.

The SQCC comprises of the original input circuit, the circuit-under-test (*CUT*), the quality evaluation circuit (QEC), and three configurable blocks. The QEC compares the output of the original circuit and the CUT (see *figure below (right)*). Error bounds encoded in the QEC evaluate the error of the CUT, e.g., worst-case error (WC), and the results of all evaluations are OR-ed to produce a single output bit, the *error'* signal. The *error'* signal signals an error bound violation if at least one error bound is violated. The three configurable blocks (*start sequence*, *Capture block*, and *Output block*) enable CIRCA to support different circuit types, e.g., streaming or run-to-completion. The Start sequence block can implement a commonly used startup protocol (a reset signal followed by a start signal) to reduce the runtime of the verification by reducing the number of possible states and state transitions. The Capture block ensures that only the valid output signals of the circuits are evaluated by the QEC. The output signal of the QEC is only made present at the output of the SQCC, if the signal is valid. This is ensured by the Output block.

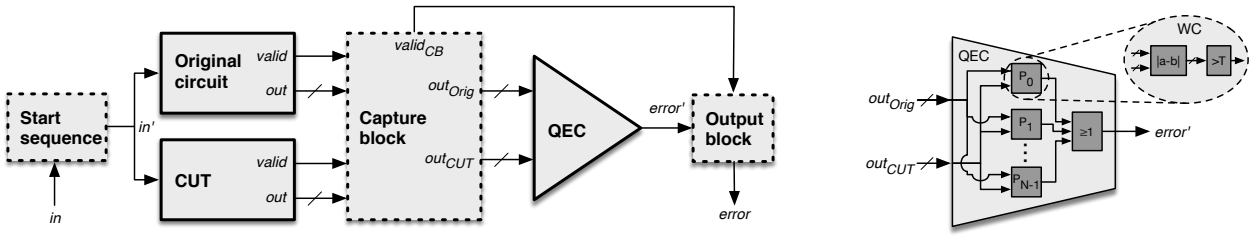


Fig. 2: Overview of CIRCA's sequential quality constraint circuit (*SQCC*) (left) and a detailed view of the quality evaluation circuit (*QEC*).

(Open the PDF here if it is not displayed.)

For more information, please read the paper on CIRCA.

Configuration options

The three implementations use the SQCC, and thus, share the same parameter set.

`Medthod` `abc_dprove`, `abc_pdr`, or `yosys_ic3`

`QualityConstraints` The global error bounds and steps which will be applied to the whole circuit. At least one error metric must be listed and each error metric must have a `bound` specified. The `step` parameter is optional and the error metric's default step size will be used if it is left out.

The syntax of quality constraint specifications is described in the *user guide's subsection about the configuration file*. All error metrics mentioned there are supported.

`CircuitType` The circuit type of the input circuit. Supported circuit types are `combinational`, `run_to_completion` and `streaming`.

`OutputSignal` The case-sensitive name of the circuit's primary output signal. Any string that is a valid signal name in Verilog is permitted.

`OutputIsSigned` Set to `1` if the circuit's primary output signal is signed, and to `0` if it is not.

`ValidSignal` (optional for each `CircuitType` except `run_to_completion`) The case-sensitive name of the circuit's signal that signals valid data at the output.

`OutputSignalConcat` (optional) If the circuit's output signal is a concatenation of multiple signals, you can specify a list of *msb-lsb* pairs. The pairs are separated by `,` with optional whitespace. A pair-entry must contain two integer numbers separated by any non-integer character other than `,`.

Example:

```
OutputSignalConcat = [31:16], [15:0]
```

`ResetSignal` (optional) The case-sensitive name of the circuit's reset signal.

`ResetSignalIsHighActive` (mandatory if `ResetSignal` is specified) Set to `1` if the circuit's reset signal is high-active and to `0` if it is low-active.

`StartSignal` (optional) The case-sensitive name of the circuit's start signal.

`StartSignalIsHighActive` (mandatory if `StartSignal` is specified) Set to `1` if the circuit's start signal is high-active and to `0` if it is low-active.

Notes and restrictions

- The circuit type, the name of the output signal, and the information whether it is signed or not must be supplied in the configuration file.
- If the circuit type is `run_to_completion`, the circuit must have a valid signal. This is an unsigned output signal of size 1 which assumes the logic value `1` as soon as the circuit's computation is completed. Its name must be specified in the configuration file.

Back to the User Guide

Approximation component

- *Input component*
- *Quality assurance component*
- **Approximation component**
- *Estimation component*
- *Search component*
- *Output*

Component description

In the configuration file, the user specifies at least one approximation technique, used to approximate the candidates. CIRCA currently provides two techniques: precision scaling (PS) and approximation-aware AIG rewriting (AIG). For each specified approximation technique, CIRCA instantiates an *approximator* and assigns the approximators to the corresponding candidates. During the approximation process, the *Approximation* component calls the candidate's approximator to generate new variants.

Additionally to the approximation techniques, the user has to specify *local quality constraints*, i.e., local error metrics and local error bounds, and may specify the step-size in which the error bounds are increased in the next iteration.

Precision scaling

The precision scaling approximation method generates approximated variants of a candidate by replacing bits of the candidate's output vector with logic 1s or 0s, starting from the LSB. This causes the logic behind the replaced bits to become obsolete; thus, enabling an external synthesis tool to remove it and thereby reduce the circuit's area, delay, power consumption, etc.

Note: Precision scaling is inherently better compatible with the *bit-flip* error metric than with *worst-case*, because the increasing significance of the replaced bits leads to an exponential increase of the absolute error, making the variant generation for all worst-case error values in-between the exponential steps effectively unnecessary.

Thus, in order to significantly reduce runtime overhead, it is highly recommended to use BF as local error metric.

AIG rewriting*[0]

*

Approximation-aware AIG rewriting operates on the And-Inverter-Graph representation of a circuit. The technique identifies the critical paths, ranks the nodes on the critical paths by their cut size, and iteratively substitutes the nodes by logical *0*. In this way, the computational delay, the area, as well as the power consumption can be reduced. To check the variant's compliance with its quality constraints, an approximation miter formed (a circuit similar to the *SQCC*) and a SAT solver is employed.

Note: This technique is restricted to combinational candidates.

Configuration options

`Method` `precision_scaling` or `aig_rewriting` or both separated by a `c`

`QualityConstraints` The default local quality constraints for each candidate which got no constraints assigned in the input stage. The format of the specification is CIRCA's *standard quality constraint syntax*.

Both the `bound` and the `step` parameter are optional. The maximum `bound` can be determined automatically for each candidate and each error metric has a default `step` value. At least one error metric must be specified. All error metrics are supported unless restricted by a candidate's approximation method.

Notes and restrictions

- The approximators specified in the config. file are the default approximators. Some front-ends allow the specification of candidate-specific configurations, including candidate-specific approximators which override the default settings.
- Approximation method implementations can specify a set of restrictions for the quality constraints of their candidates. There may be incompatibilities in some configurations of approximation methods and error metrics.

[0] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. 2016. Approximation-aware rewriting of AIGs for error tolerant applications. In Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD '16). ACM, New York, NY, USA, Article 83, 8 pages. DOI: https://doi.org/10.1145/2966986.2967003

In such a case, the quality constraints will be adjusted accordingly, if possible, and a warning message will be generated. If the adjustments lead to an invalid configuration, e.g., no error metrics left, the approximation process will not be started and CIRCA will terminate with an error message.

Back to the User Guide

Estmation component

- *Input component*
- *Quality assurance component*
- *Approximation component*
- **Estimation component**
- *Search component*
- *Output*

Component description

The *Estimation* component utilizes synthesis tools to synthesize the approximated circuits, i.e., a particular circuit configuration. From the synthesis reports, CIRCA extracts the circuit parameters of interest and attaches them to the circuit under investigation. In the search block, the circuit's statistics is used to evaluate the approximated circuit, for example, with a heuristic function. In this way, the different circuits generated throughout the approximation process can be ranked to find the best-suited circuit for the user. Common target metrics are *hardware area*, *circuit delay*, and *power or energy consumption*.

abc_if

CIRCA's standard Estimation implementation is `abc_if`. The implementation utilizes *ABC*'s `if` command to perform a technology mapping to FPGA 4-LUTs. The reported circuit statistics are stored in a file in the directory *approx_circuits*. From these files CIRCA extracts the statistics of each circuit, i.e., area, delay, and power dissipation due to switching.

Configuration options

```
Method abc_if
```

Note: ABC's if command provides some configuration parameters. `abc_if`, however, does not provide these parameters in the configuration file. If you need to adjust the parameters, e.g., the number of the input of the LUTs, feel free to modify the code and extend the configuration possibilities.

Back to the User Guide

Search component

- *Input component*
- *Quality assurance component*
- *Approximation component*
- *Estimation component*
- **Search component**
- *Output*

Component description

The *Search* component specifies the algorithm which is used to navigate the search space. Its primary functions are *selecting* the circuit configuration to be validated next, *expanding* the search space by defining new circuit configurations to be generated, and *evaluating* the results of the approximation process by analyzing the circuit statistics generated by the *Estimation component*.

CIRCA's Search implementations are:

- *gradient descent (or hill climbing)*,
- *Simulated Annealing*,
- *Monte Carlo Tree Search*, and
- *Jump Search*.

Gradient Descent Search

Description

The `gradient_descent` Search implementation uses a circuit configuration's *area* value and models it as a function of the configuration's current error bounds and approximation method. With this model of the search space in mind, a modified gradient descent search algorithm is utilized to approach a minimum of that function. The starting point for the search is the original circuit.

To expand the search space, all *neighbors* of the current circuit configuration are generated. The neighbors of a configuration differ from the current configuration in either the approximation method or one error metric with an increased error bound.

The neighbors are then categorized into three groups by their *area* values: One group for circuits with *lower area* than the current circuit (*negative gradient*), one group for circuits with *the same area* (*zero-gradient*) and one group for circuits with *higher area* (*positive gradient*). The circuits from the first group with *the lowest area* are appended to a list of configurations to inspect later. More configurations, also from the other groups, are added to that list according to an *Effort* parameter set in the configuration file.

Depending on the configuration, either the best circuit from the list or a random circuit is then used as the new current circuit configuration for the next iteration. This cycle continues until the list is empty. This is the case if a local area minimum has been reached or the quality constraints allow no further area improvement.

Notes and restrictions

- This Search implementation considers exclusively the *area* values of the circuits and thus only leads to reliable area improvements.

- The *Effort* parameter can be used to configure how many circuit configurations are inspected. The options vary from a straight-forward continuous area reduction to an exhaustive brute-force style search. This may lead to significant differences in runtime between the *Effort* levels.

Configuration options

Method `gradient_descent`

Effort The effort level of the search, determines how many circuit configurations will be inspected. Possible values are integers `0` to `6`:

- **0** Only negative gradients are inspected
- **1** All negative and 5 randomly chosen 0-gradients are inspected
- **2** All negative and 20 randomly chosen 0-gradients are inspected
- **3** All negative and randomly chosen 25% of 0-gradients are inspected
- **4** All negative and randomly chosen 50% of 0-gradients are inspected
- **5** All non-positive gradients are inspected
- **6** Take all gradients into account (pos., neg., and zero)

SelectStrategy (optional) Specifies whether the best (lowest gradient) or a random circuit configuration from the list will be used as starting point for the next iteration. Possible values are

- **best** (default) Take the best configuration
- **random** Take a random configuration

Simulated Annealing

Description

Initially, the `simulated_annealing` search starts with the original circuit and a temperature of *1.0*. During the search a node, i.e., a circuit, is randomly selected. Depending on the `acceptance` function, the selected node is either *accepted* or *rejected*. A node which improves the target metric is always accepted. A node which worsens the target metric may be accepted under the probability of

$$random[0.0, 1.0) < e^{-\frac{\Delta D}{T}}.$$

ΔD represents the change in the target metric and T the current temperature. If a node is accepted, its quality is verified. If the node is valid, the node is expanded, i.e., its neighboring nodes are generated, and its neighbors represent then the open-list. If the node is being rejected or invalid, another node from the open-list is randomly selected.

For each temperature T, the *equilibrium* has to be reached, i.e., *equilibrium* number of iterations have to be performed, before the temperature T is lowered by

$$temperature = temperature * \alpha.$$

After the minimal temperature T_{Min} is reached, the search terminates.

Notes and restrictions

- The search uses an initial temperature of *1.0*.

Configuration options

Method `simulated_annealing`

TMin The minimal temperature at which the search terminates. Must be ≥ 0.

Alpha Rate at which the temperature is reduced after equilibrium is reached. Must be ≥ 0.

Equlibrium Number of iterations performed at each temperature.

Heuristic Specifies the target metric directly or uses a heuristic to express the merit of the node. The value is used to compute ΔD to determine whether a node is accepted or not.

Area (default) The circuit's hardware area is considered as target metric. For the computation of ΔD the relative hardware is used: $\Delta D = \frac{area_{new} - area_{old}}{area_{old}}$.

Monte Carlo Tree Search (MCTS)

Description

Monte Carlo tree search (MCTS) is a stochastic best-first search technique that exploits the generality of random sampling to build an efficient search algorithm. In each iteration, MCTS performs four main steps: selection, expansion, simulation and update. The selection step always starts from the root node of the search tree and attempts to reach at a leaf node while using the UCT score of each nodes as a heuristics in the path.

$$UCT(node_i) = \frac{W_i}{V_i} + C\sqrt{\frac{\ln(V_N)}{V_i}}$$

where W_i is the reward value of the $node_i$, and V_i is the number of visits of the $node_i$. Moreover, V_N shows the number of visits for the parent of $node_i$, and C represents the exploration constant which trades off between exploration and exploitation in the search tree.

The expand step adds new children nodes to the search tree. The simulate step then evaluates the node's statistics and then calculates the weighted reward for the node based on the following formula:

$$Reward_{node} = \alpha * area\_savings + \beta * delay\_savings + \gamma * power\_savings$$

This completes one iteration of the MCTS. The search terminates when the allocated search budget to MCTS is exhausted.

Configuration options

Method `mcts`

Budget Number of iterations for the search.

Scaler Constant value used in the UCT formula (usually ranges [0-2.5]).

Jump Search (JS)

Description

The Jump Search search method seeks to minimize the runtime of the approximation process by minimizing the number of expensive synthesis and verification steps. JS operates in two phases:

Phase 1 (path planning): Determine a path through the search space, starting from the original circuit to the boundaries of the search space. In phase 1, the AxCs found, are being evaluated using the Figure of Merit (FOM):

$$FOM(AxC) = \sum_{cands \in AxC} \sqrt{\frac{Err(c)}{MaxE(c)}} \times \frac{if_{Area}(c)}{|if_{Err}(c)| + 1}$$

The FOM takes into account the normalized error of the candidates within the AxC to balance the selection, as well as two impact factors, if_{err} and if_{area}. The two impact factors of each candidate are being determined in a pre-processing step. Employing the FOM to evaluate the AxCs makes any invocation of synthesis or verification obsolete, and thus, significantly reduces the runtime of the approximation process.

Phase 2: JS makes the assumption that the deeper a node is on the path, the more aggressive are the applied approximation to that AxC, and thus, the larger are the expected savings. In phase 2, employing synthesis and verification is inevitable; however, to minimize the number of invoked synthesis and verification steps, a binary search is utilized, finding the deepest, valid node on the path.

Note: For a more detailed description, please see our paper presented at the 4th AxC workshop at the DATE'19 conference and our paper presented at the GLSVLSI'19 conference.

- L. Witschen, H. Ghasemzadeh Mohammadi, M. Artmann, and M. Platzner, "Jump Search: A Fast Technique for the Synthesis of Approximate Circuits," Presented at the Fourth Workshop on Approximate Computing (AxC '19) 2019, Florence, Italy, 2019.
- L. Witschen, H. Ghasemzadeh Mohammadi, M. Artmann, and M. Platzner, Jump Search: A Fast Technique for the Synthesis of Approximate Circuits, In the proceedings of the Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), 2019, Tysons Corner, VA, USA. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3299874.3317998.

Configuration options

Method `jump_search`

ImpactFactorsErr Dictionary containing the impact factors if_{err} of each candidate. Example:

```
ImpactFactorsErr = {'Cand_Adder_0': 0.16, 'Cand_Mult_0': 0.4, 'Cand_
↪Adder_1': 0.914}
```

ImpactFactorsArea Dictionary containing the impact factors if_{area} of each candidate. Example:

```
ImpactFactorsArea = {'Cand_Adder_0': 0.25, 'Cand_Mult_0': 0.412, 'Cand_
↪Adder_1': 0.09}
```

Back to the User Guide

Output component

- *Input component*
- *Quality assurance component*

- *Approximation component*
- *Estimation component*
- *Search component*
- **Output component**

Component description

The *output* component takes care of post-processing and filtering the approximation results. It can also be used to do cleanup work, i.e. remove directories and files which were used during the approximation process but shall not remain afterwards.

The standard *output* implementation of CIRCA is called `output_best_area`.

Functionality

The `output_best_area` implementation copies all approximated and validated circuits to the *output* directory and prints statistics about the whole process to the console. These include a list of all approximated circuits, the smallest area, the original circuit's area and the numbers of validated and valid circuits.

Notes and restrictions

- This *output* implementation was designed to work best with the `gradient_descent_search` implementation, or *Search* implementations focusing on the circuit's *area* in general.

Config options

`output_best_area` does not require additional configuration.

Back to the User Guide

1.2.4 Example configuration

This is an example walkthrough of the configuration process to give you an idea of the steps required for making CIRCA work with your own input design.

Let's assume you have a run-to-completion circuit `my_circuit.v` with a top-level module called `TOP_MOD` and three other modules called `MOD_1`, `MOD_2` and `MOD_3`, which are instantiated in the top module. The goal is to generate an approximate version of the circuit with minimal area, but its output must not deviate from the original circuit's output by an absolute value of more than 42.

```
module TOP_MOD (/*portlist...*/);
// Many MOD_1 instances
// One MOD_2 instance
// One MOD_3 instance
// ...
endmodule

// MOD_1, MOD_2 and MOD_3 definitions
```

All three modules shall be considered as approximation candidates, but `MOD_1` is special in that it has far more instances than the other two. You decide that it should not be approximated as far as the other modules and may only have a maximum bit-flip error of 2. However, multiple approximation methods should be used on it, while `MOD_2` and `MOD_3` should have AIG-rewriting as their only approximation method. They may also have a worst-case error as high as necessary to achieve a good area.

The circuit is quite large, but you have a powerful system and some time, so you are ready to accept a longer runtime in exchange for better results. However, since you want to test CIRCA's accuracy, you decide to choose a medium-effort search strategy and compare the results to a brute-force search later.

Preparing the circuit

To assign different approximation methods and quality constraints to the modules, you may have to use annotations in the input design. The `AnnotatedCandidates` FrontEnd implementation offers exactly the required functionality and is applicable to the circuit, because all approximation candidates are modules and all primary input and output signals have a static size.

To mark the modules as candidates, you decide to use the recommended annotation key:

```
// ...
module <<<APPROX_CAND>>> MOD_1 (/*ports*/);
// MOD_1 content
endmodule
// ...
module <<<APPROX_CAND>>> MOD_2 (/*ports*/);
// MOD_2 content
endmodule
// ...
module <<<APPROX_CAND>>> MOD_3 (/*ports*/);
// MOD_3 content
endmodule
```

The candidate `MOD_1` needs very specific settings, while `MOD_2` and `MOD_3` can have the same, so their settings can be specified as default settings in the configuration file. After adding the approximation method and quality constraints for `MOD_1`, its definition looks like this:

```
// ...
module <<<APPROX_CAND>>>>>><BF:step=1,bound=2;appMethods:PS,AIG>>> MOD_1 (/
↪*ports*/);
// MOD_1 content
endmodule
// ...
```

The circuit is now ready for approximation.

Note: The specification of candidate-specific setting, such as for `MOD_1` in the example above, is optional. If no parameters are specified, the configuration specified in the configuration file is used.

Preparing the configuration file

Because you never used CIRCA before, you make a copy of the supplied example configuration and adjust each section if necessary:

The `General` Section must specify the name of the top module, so it looks like this:

```
[General]
TopModule = TOP_MOD
```

Since you decided to use the `AnnotatedCandidates` FrontEnd implementation, its name must be specified as value of the `Method` option. The annotation key must be the same as the one in the input design, which is already the case:

```
[Input]
Method = AnnotatedCandidates
Key = <<<APPROX_CAND>>>
```

As search method with adjustable effort level, `gradient_descent` is suitable. To get a medium effort, you decide to use effort level 4. To test CIRCA's accuracy, the `SelectStrategy` should be left at the default value `best`

```
[Search]
Method = gradient_descent
Effort = 4
SelectStrategy = best
```

The standard Estimation implementaion `abc_if` provides a good area estimation, so this section can be left unchanged:

```
[Estimation]
Method = abc_if
```

For the `QualityAssurance` section, you remember the global quality constraint of a maximum worst-case error of 42. You don't know the difference between the standard methods, so you decide to leave it at `abc_dprove`. You also have to look up the name of the output signal and specify the circuit type. Since it is `run_to_completion`, the name of the valid-signal must be specified aswell. The circuit has no start- or reset-signal, so these options are omitted:

```
[QualityAssurance]
Method = abc_dprove
QualityConstraints = WC:bound=42
OutputSignal = out
OutputIsSigned = 0
CircuitType = run_to_completion
ValidSignal = valid
```

In the `Approximation` section, you specify the approximation method and quality constraints for the modules `MOD_2` and `MOD_3`: they were supposed to use AIG-rewriting and have maximum worst-case error if necessary. Since this makes the `bound` parameter unnecessary, it can be omitted. However, the step size should be small in order to get accurate results:

```
[Approximation]
Method = AIG
QualityConstraints = WC: step = 1
```

The `Output` implementation doesn't really matter for your purpose, so you keep the standard value:

```
[Output]
Method = output_best_area
```

Finally, the input files are ready for the approximation process:

`my_circuit.v`:

```
module TOP_MOD (/*ports*/);
// Many MOD_1 instances
// One MOD_2 instance
// One MOD_3 instance
// ...
endmodule
// ...
module <<<APPROX_CAND>>>><<<BF:step=1,bound=2;appMethods:PS,AIG>>> MOD_1 (/
↪*ports*/);
// MOD_1 content
endmodule
// ...
module <<<APPROX_CAND>>> MOD_2 (/*ports*/);
// MOD_2 content
endmodule
// ...
module <<<APPROX_CAND>>> MOD_3 (/*ports*/);
// MOD_3 content
endmodule
// ...
```

my_config.cfg:

```
[General]
TopModule = TOP_MOD

[Input]
Method = AnnotatedCandidates
Key = <<<APPROX_CAND>>>

[Search]
Method = gradient_descent
Effort = 4
SelectStrategy = best

[Estimation]
Method = abc_if

[QualityAssurance]
Method = abc_dprove
QualityConstraints = WC:bound=42
OutputSignal = out
OutputIsSigned = 0
CircuitType = run_to_completion
ValidSignal = valid

[Approximation]
Method = AIG
QualityConstraints = WC: step = 1

[Output]
Method = output_best_area
```

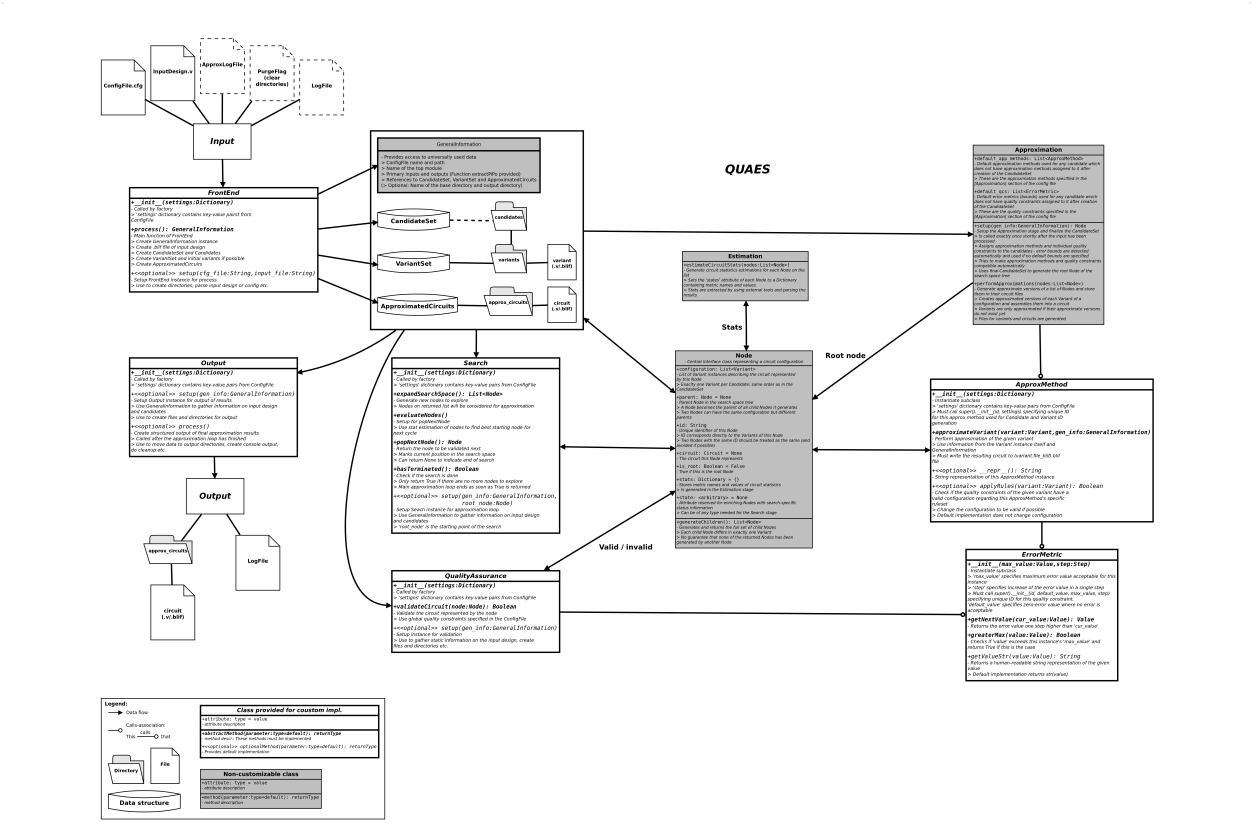
Executing CIRCA

The approximation process can now be started by navigating to the directory with the input design and the configuration file and typing

```
circa my_config.cfg my_circuit.v
```

Note: This is a WIP Beta version. All content is subject to change.

1.3 Developer Guide



This section will teach you how to develop custom component implementations and integrate them into CIRCA's approximation flow. The covered topics are:

- *Basics*
- *Important data structures and classes*
- *Exchangeable components*
- *Extensible components*

You can find more detailed technical information on the framework's modules in the *module index*.

1.3.1 Basics

CIRCA is designed to be:

- *General*: The framework should not be restricted to certain circuit types, error metrics, approximation and search techniques, or specific target technologies.
- *Modular*: The framework architecture should enable the exchange of certain processing steps without affecting other steps. Modularity is key for the evaluation and the comparison of different techniques under a consistent experimental setup.
- *Compatible*: The framework, in particular its inputs and outputs, should connect to other, widely-used academic and commercial front-end and back-end tools, e.g., tools synthesizing circuits for ASIC or FPGA technology.
- *Extensible*: The framework should facilitate the swift implementation and evaluation of new techniques.

To achieve this, CIRCA's approximation flow is handled by independent stages and processing blocks (short: components) which communicate using well- defined interfaces. Each component can be extended by custom implementations and provides a specific piece of functionality which is needed for the approximation process. Integrating your own component implementations allows you to easily test different search strategies, approximation techniques, and quality assurance methods.

The framework provides the necessary structure for the components to work independently. CIRCA utilizes data structures as well as classes which can be accessed and interpreted by all components, enabling them to work together effectively.

Component implementation

The customizable components of CIRCA are classes, implementing a specific interface. The interfaces are specified by *abstract base classes* from which the custom implementations must be derived. They were implemented using Python's Abstract Base Classes module, so their subclasses have to implement some methods in order to be instantiatable. The concept of abstract base classes ensures that new implementations provide the functions used by CIRCA in its standard approximation flow.

To create a custom component implementation, simply create a subclass of the abstract base class for that component and make sure to implement all of the base class's abstract methods (note that it is possible to derive from an already existing subclass to minimize implementation effort).

Each abstract base class provides a dictionary `SUPPORTED_SUBCLASSES`. The key corresponds to the method's name used in the configuration file, e.g., *AnnotatedCandidates*. The value is a lamda expression, calling the constructor of the corresponding implementation. Once an implementation is finished, i.e., the subclass, add the key-value pair to the dictionary to make the implementation available.

Component classification

We differentiate between two types of customizable components: *exchangeable components* and *extensible components*.

- Exchangeable components are the stages and processing blocks *Input*, *QualityAssurance*, *Estimation*, *Search*, and *Output*. Exactly one implementation of each component is used when CIRCA is executed. The used implementations have a great impact on the approximation process and on the quality of the outcome.
- Extensible components can be seen as functionality which is either used or not used in an approximation process. Extensible components include *approximators* from the *Approximation* block and *ErrorMetrics*. While the number of exchangleable components in an approximation process is restrricted to exactly one, the user has no upper limit on the number of employed extensible components.

1.3.2 Important data structures and classes

In order to implement your custom components, you need to know the data structures, classes, and interfaces you will be working with. Since most of them are already documented extensively in the code, their implementation details will be omitted here. Instead, to improve the understanding of the objects, an overview of the scope of the particular object is given from a high-level perspective.

GeneralInformation

The `GeneralInformation` class acts as a container for information used by several components and is instantiated once in the *Input* stage. After instantiation, it is passed to every other stage when they are created.

The `GeneralInformation` object's content includes paths to the configuration and input circuit files, information on the input circuit, and references to the data structures *Candidates*, *Variants*, as well as nodes and circuits.

When creating an *Input* component implementation, your job is to setup the `GeneralInformation` instance and usually also extract PI/PO (primary input/primary output) information, either manually or using the existing method. The PI/PO information is needed, for example, in the quality assurance step. For the automatic extraction of PI/PO information from Verilog code, `GeneralInformation` provides a method.

Candidates

A *candidate* is a part of the input circuit which can be replaced by an approximate component. Usually, the arithmetic units in the data path of a circuit are specified as candidates. The selection of candidates in a circuit is crucial since the selection of candidates directly influences the quality of the approximated outcome.

Candidates are represented by the `Candidate` class. Each `Candidate` instance must have a unique name and specify local quality constraints, i.e., local error metrics with error bounds. The local quality constraints of a candidate Every candidate has a reference to its original implementation (a circuit file containing only the candidate from the input circuit). From the candidate's original implementation, the candidate's variants are generated and stored in a candidate-specific directory.

A candidate can represent different parts of a circuit, e.g., a register, a wire, or an entire circuit module. Depending on the component or part of the circuit represented by the candidate, the candidate may have to be treated differently and may offer different methods. Thus, the `Candidate` class is abstract and cannot be instantiated. A specific implementation of a candidate is described in a subclass, derived from the `Candidate` class. In this way, CIRCA does not restrict candidates to a specific parts of a circuit. Furthermore, the candidate's type can be used to derive a unique name for the candidates.

All candidates are stored in a `CandidateSet`, which is a basic data structure containing the `Candidate` instances and the path to the directory in which the candidate files are stored. It is accessible through the `GeneralInformation` instance. Candidates can be added and removed by all components.

Variants

A *variant* is an unique instance of a candidate with a certain degree of approximation applied. The relation of a variant to its candidate can be compared to the relation between an object and its class. The candidate specifies the part of the input circuit to approximate, at least one approximation method and some quality constraints. The variants are the resulting circuits, each having only one approximation method and possibly adjusted quality constraints.

Variants do not need a name, since they generate a unique ID value based on their candidate's name, their approximation method, and their local quality constraints. Since the ID can become very long, a hash value is generated which can be used instead. The `Variant` instances have a reference to their candidate and to their circuit file.

`Variant` objects can be created in two ways: A `Candidate` instance can generate its own original variant, which is basically a copy of the candidate's part of the input circuit. Each `Variant` instance can then generate its "neighboring" variants by either switching the approximation method or increasing one error bound by one step. These variants are called *neigbors* or *children* of the variant generating them. The variants created this way do not have a circuit file however. They must be chosen for the *Approximation* stage where these files are created.

All variants are stored in a `VariantSet`, a data structure similar to the `CandidateSet` with a slightly different internal structure. It can also be accessed through the `GeneralInformation` instance and manipulated by all stages. Throughout the approximation process, the *Approximation* stage creates new variants and adds them to the `VariantSet`; hence, the set is continuously extended. This can be seen as an on-the-fly generation of an approximate component library. That is, if an existing approximate component library is loaded into CIRCA, it would either represent or even replace the `VariantSet`.

Note that a variant's ID (or hash) is unique. Thus, each variant is only created once in an approximation process which improves efficiency.

Nodes and Circuits

In the approximation process, approximated variants of all specified candidates are generated. Then, CIRCA replaces the original implementation of a candidate with its approximated variants to create an approximated circuit. These *approximated circuits* are represented by instances of the `Circuit` class. An instance holds information about the circuit file and the result of the verification, i.e., whether the circuit's quality has been validated by the *QualityAssurance* stage and whether the circuit has passed the check.

All `Circuit` instances are stored in an `ApproximatedCircuits` instance representing the directory in which the circuit files are stored. A circuit can be stored in Verilog and in BLIF format.

An approximated circuit is uniquely described by the combination of variants used to implement the candidates. We denote this combination as *circuit configuration*. Since the search space is modeled as a graph structure, we identify, in the context of the search, configurations with `Nodes`. A `Node` is a representation of a circuit in another context, providing different information and functionality. Once a `Node`'s circuit has been generated, the corresponding `Circuit` instance and the `Node` are connected by bidirectional references. The `Node` class specifies the interface for traversing the search space and working with circuits without having to provide detailed circuit information on this abstract level. The components use `Nodes` as input and output of their functions, i.e., as an interface.

A `Node` has a method for generating its children, similar to the `Variant` class. The children are those nodes, providing a circuit configuration with only one candidate implementing a different variant than the parent and this variant has a local error bound increased by one step compared to the parent's variant. For example, consider a node with the configuration $\{C_{1,\,WC1}; C_{2,\,WC3}\}$, $C_{1,\,WC1}$ being a candidate's variant with an worst-case error bound of *1* and $C_{2,\,WC3}$ a candidate's variant with a worst-case error bound of *3*. Then, the children of this node are: $\{C_{1,\,WC2}; C_{2,\,WC3}\}$ and $\{C_{1,\,WC1}; C_{2,\,WC4}\}$ (assuming a step-size of *1*). Note that this means that a `Node` instance could generate a child `Node` with the same configuration as its parent, by changing a candidate to its previous variant. Avoiding these circles is part of the *Search* stage component and can be accomplished using the `Node`'s `id` attribute.

There is no central data structure storing all `Nodes` as this is considered to be the *Search* component's task and many nodes are likely to be never actually used. Make sure to look into the `Node` documentation if you want to implement your own stage component.

1.3.3 Exchangeable components

Exchangeable components share many properties which must be considered when developing a custom implementation. Here you can find the features they have in common and detailed documentation of all five components.

Shared features

All abstract base classes of exchangeable components provide a `factory` function which is called for component instantiation. A `factory` function takes the config. file as an input and uses the `Method` parameter from the corresponding config. section to instantiate the correct subclass.

All supported implementations are listed in the base class's `SUPPORTED_SUBCLASSES` attribute, a dictionary storing the names of all available implementations as keys and constructor references as values, using lambda expressions. The function also extracts all key-value pairs from the component's config section and passes them to the constructor, stored in a dictionary as well.

So, when creating a new component implementation, the first step is to add a new entry to the `SUPPORTED_SUBCLASSES` dictionary

```
"MyComponent": lambda s: MyComponent(s)
```

where `MyComponent` is the name of your class and `MyComponent(s)` is the constructor of the class, receiving the key-value pairs of the config. section. Also make sure that your `__init__` method has the correct signature with only the config. option dictionary as parameter:

```
class MyComponent(ComponentABC):
    def __init__(self, settings_dict):
```

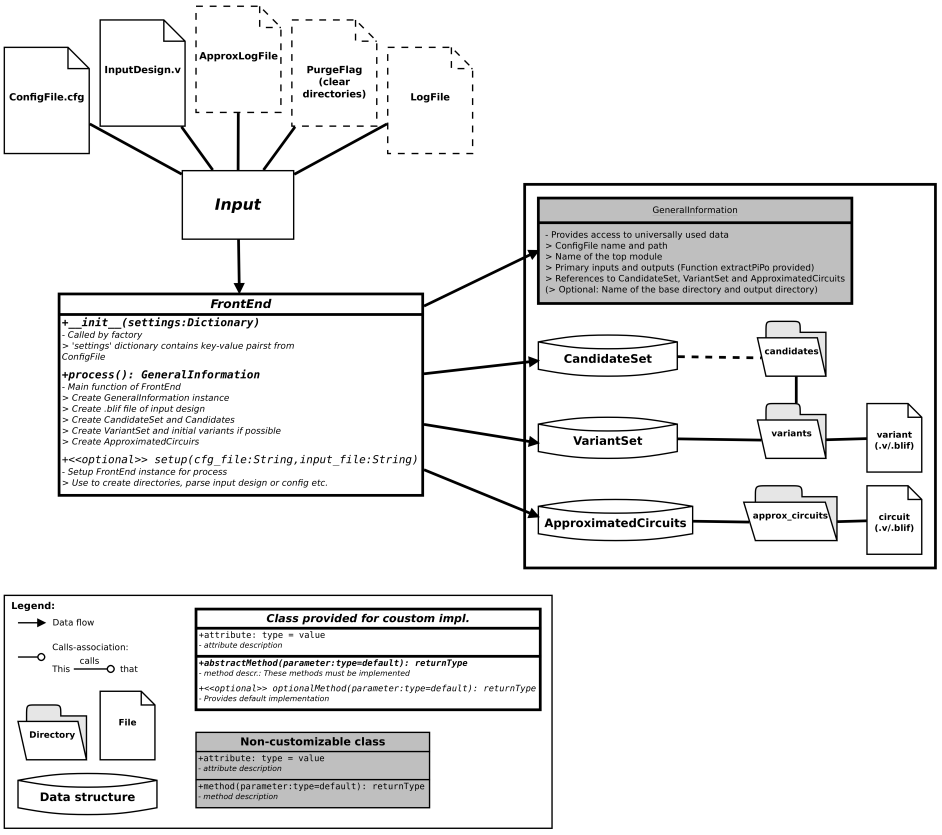
Note that `ComponentABC` can be the component's abstract base class or one of the existing subclasses, if you want to base your implementation on something more substantial.

The exchangeable components also share an optional `setup` method. The `setup` function is called immediately after instantiation of the component and can be used to prepare the component for the approximation process. The function takes - at least - the `GeneralInformation` object as input and is used to perform further setup steps.

Component details

Exchangeable components

- **Input component**
- *QualityAssurance component*
- *Estimation component*
- *Search component*
- *Output component*



Input component

The *Input* (or *FrontEnd*) component pre-processes the input circuit, e.g., removing code annotations to restore valid Verilog code, and extracts information from the circuit, e.g, candidates, to initialize required data structures, e.g., the candidate set. In other words, the *Input* component's role in the approximation process is to create a foundation on which the other components can operate.

Functions

`setup(self, cfg_file: String, input_file: String): None`

The `setup` method can be used to initialize the component by storing the configuration and input files and perhaps do some pre-processing such as validation and compatibility checking of the input circuit. The default implementation stores the files and only checks if the input circuit file name is not empty.

`process(self): GeneralInformation`

The `process` method provides the main functionality of the *Input* component. It is called exactly once at the very beginning of the approximation process and must return a `GeneralInformation` instance

which is fully set up to be used in the *QUAES* stage. It doesn't matter how a specific implementation accomplishes this goal, as long as the following post-conditions are met:

- A `GeneralInformation` instance was created
- `CandidateSet`, `VariantSet`, and `ApproximatedCircuits` instances were created and stored in the `GeneralInformation` instance
- The `CandidateSet` instance is filled with the complete set of `Candidates` to be considered in the approximation process (they don't need to have `quality_constraints` or `approx_methods` yet. In case of missing information, the default parameter will be filled in later)
- The `CandidateSet` and `ApproximatedCircuits` instances have valid `path` attributes, i.e., valid directory paths
- There is a `*.v` and a `*.blif` file storing the original circuit, where `*` is the original circuit's name
- Following attributes of the `GeneralInformation` instance are set correctly:

  `top_module` The name of the original circuit's top level module

  `pis` and `pos` Lists of `Signal` instances representing the original circuit's primary input and output signals (can be extracted from Verilog code automatically with the provided function `extractPiPo()`)

  `base_directory` (Optional, default is the directory containing the input file) The root directory under which all directories and files of the framework will be stored

  `output_dir` (Optional, default is the directory containing the input file) The directory in which all output data will be stored

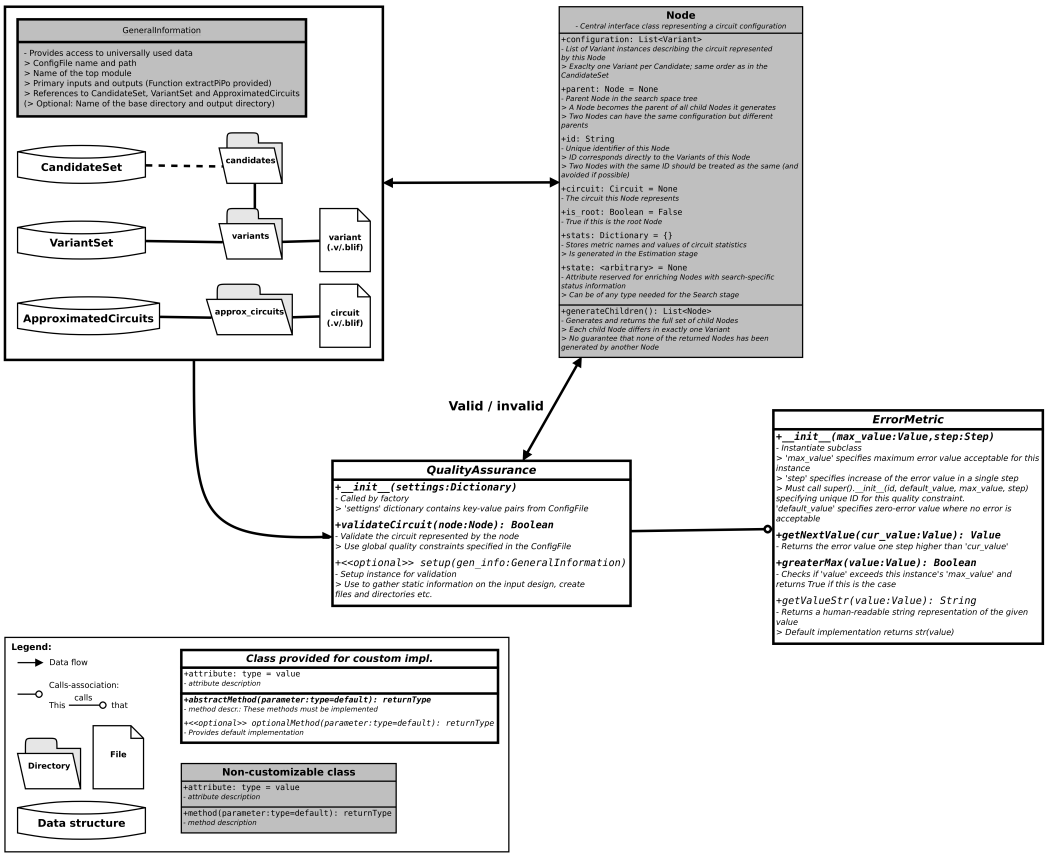
The hardest part of this is probably creating the `CandidateSet`, since the other tasks can be easily done once the set has been created. However, extracting the candidates from the original circuit in a particular manner is the main argument for creating a custom `FrontEnd` subclass, so this is the part on which you should focus.

The framework and the classes themselves provide many handy utility functions and sensible default values, so make sure to have a look at existing `FrontEnd` implementations and the other classes' documentation before you start writing your code.

Back to the Developer Guide

Exchangeable components

- *Input component*
- **QualityAssurance component**
- *Estimation component*
- *Search component*
- *Output component*



QualityAssurance component

The *QualityAssurance* component (or stage) is responsible for validating the circuits resulting from the approximation process against global quality constraints. Its main function is simply deciding whether or not an approximated circuit satisfies all quality constraints. The result can be used in the search process to recognize boundaries in the search space or eventually terminating the search.

Functions

`setup(gen_info: GeneralInformation): None`

The `setup` method can be used to initialize this component by storing the `GeneralInformation` instance, parsing the global quality constraints from the configuration file and for creating files and directories that are needed for the validation process. The default implementation stores the `GeneralInformation` instance and creates a directory `sqcc_files` in the base directory, assuming that auxiliary circuits will be created for validation.

`validateCircuit(node: Node): Boolean`

The `validateCircuit` method validates the circuit of a given `Node` against the global quality constraints and only outputs whether the validation was successful or not. To store the result and make it

interpretable for the other components, the `Circuit` instance's `validated` attribute should be set to `True` and its `valid` attribute set to the result of the validation.

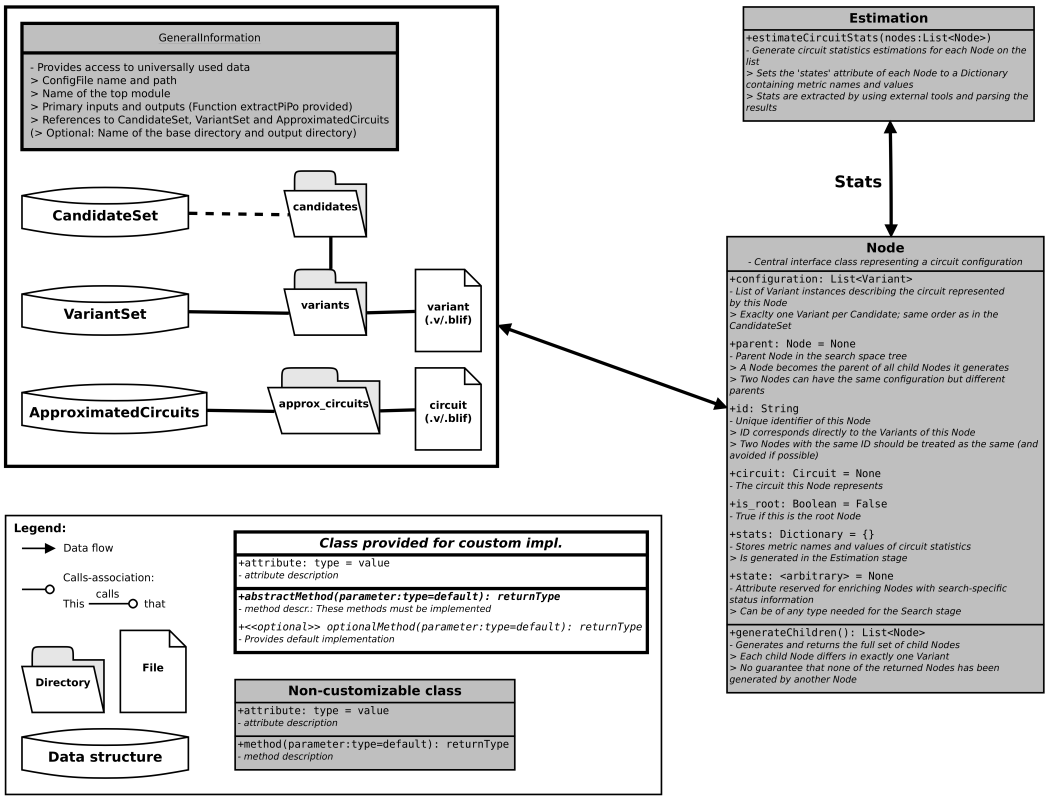
Note that this method works with a `Circuit` instance, but gets a `Node` as parameter. This is because `Node`s are the central interface for circuit representations and component interaction. A `Node` instance however stores a reference to its respective `Circuit` instance, which is accessible via `node.circuit`. This attribute will always contain a `Circuit` instance if the `Node` was put through the approximation process beforehand.

Since quality assurance techniques are generally very diverse and require a lot of setup and cleanup, this is typically the start of a toolchain or a series of helper function calls. Feel free to add as many more functions as you need to compute the result of the validation.

Back to the Developer Guide

Exchangeable components

- *Input component*
- *QualityAssurance component*
- **Estimation component**
- *Search component*
- *Output component*



Estimation component

Note: The Estimation component is not customizable yet.

The *Estimation* component is used by the framework to generate statistics of the circuits generated during the approximation process. Statistics of interest are, for example, *area*, *delay*, *power*, *energy*, and *error*. These statistics can either be directly used or incorporated into a heuristic to compare circuits with each other. This information can then guide the search to improve the target metric. CIRCA currently invokes ABC's *if* command to synthesize a circuit and to obtain estimations about *area*, *delay*, and *power dissipation due to switching*.

Functions

`estimateCircuitStats(nodes: List<Node>): None`

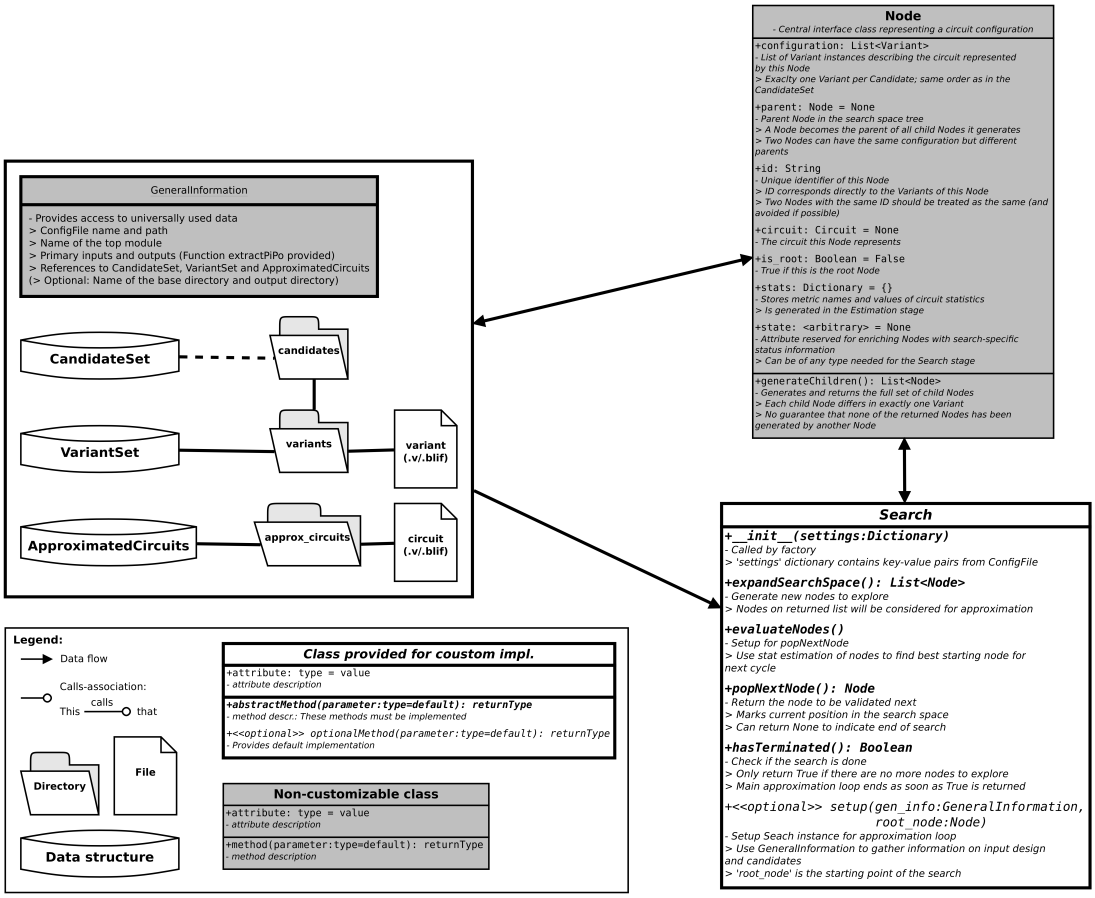
This method implements the main functionality of the *Estimation* stage. It is called each time the approximation stage yields another set of approximated circuits. As for all stages, the circuits are represented by their `Node`s, hence the parameter of the method is a list of `Node` instances.

Every `Node` has a parameter called `stats` which contains a dictionary. Initially, this dictionary is empty, and it is this method's task to fill in useful statistics in the form of key-value entries like `"statName": statValue` where `statName` is one of the supported metrics and `statValue` is its value of type `float`, or `None` if no value for this metric was found.

Back to the Developer Guide

Exchangeable components

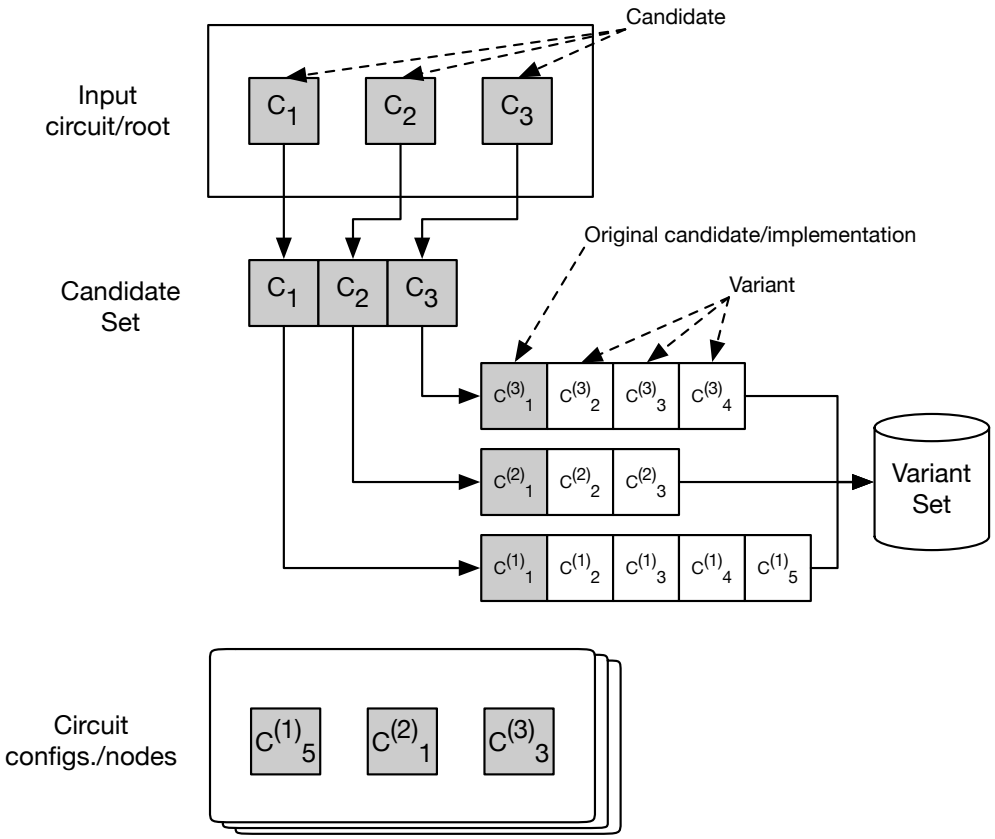
- *Input component*
- *QualityAssurance component*
- *Estimation component*
- **Search component**
- *Output component*



Search component

The *Search* component controls how the search space (modeled as a tree-like graph structure) is traversed, i.e., which nodes to approximate and validate next and when to terminate the approximation process. This also means that the *Search* component must manage already visited nodes, soon to be visited nodes, and "anomalies" in the search space, such as recurring circuit configurations, on its own.

Structure of the search space



To talk about the concepts of the search space more efficiently, we first define some terms and explain their meaning. The figure above visualizes the terms.

Candidate A *candidate* is a part of the input circuit which can be approximated independently and then inserted back into the input circuit to replace its own original variant, thereby creating an approximated version of the whole input circuit. The *set of candidates* of the input circuit is generated during the *Input* stage before the approximation process begins. For each candidate there must be at least one *approximation method* and at least one *error metric* with an upper bound specified.

Variant We denote an approximated version of a candidate as *variant*. Each variant has a unique candidate, but candidates generally have multiple variants. The original, non-approximated version of a candidate is called *original variant*. Variants are uniquely identified by the ID of their candidate in conjunction with an error bound for each error metric and exactly one approximation method.

Circuit configuration Assume the input circuit has the candidates $(C_1, C_2, \ldots, C_n)$ and each candidate C_j can be represented by one of its variants $\{V_1^{(j)}, V_2^{(j)}, \ldots, V_m^{(j)}\}$. We call the tuple of variants $\left(V_x^{(1)}, V_y^{(2)}, \ldots, V_z^{(n)}\right)$ the *configuration* of the circuit in which each candidate C_j is replaced by one of its variants $V_i^{(j)}$. A circuit configuration is unique.

Node A *node* in the search space represents a single circuit configuration $\left(V_x^{(1)}, V_y^{(2)}, \ldots, V_z^{(n)}\right)$ and is treated as a node (or vertex) in a graph. It is connected to its *parent node* and to its *child nodes* (or *children*) once they are created. The children of a node represent circuit configurations which differ from the node (their *parent*

node) in exactly one variant, i.e., one variant $V_i^{(j)}$ is replaced by another variant of that candidate C_j and all other variants remain the same. The new variant of that candidate is not arbitrary, but must have either a different approximation method or a single error bound increased by one step. Note that nodes representing the same configuration can be created by different parent nodes, so that some nodes in the search space may have multiple parent nodes (this is the reason why the search space is not really a tree). The node representing the circuit configuration with only original variants is called the *root node*. It does not have a parent node.

Nodes are represented by `Node` instances which contain all necessary information about the circuit configuration represented by the graph node. A `Node` instance can generate its children with the `generateChildren` method. These nodes are freshly generated each time the method is called and neither do they have any file representations nor are they stored in any data structure, i.e., at first, they are temporary objects iwth the minimal required information. The search can then decide whether these nodes are generated by the approximation stage or simply discarded.

Functions

`setup(gen_info: GeneralInformation, root_node: Node): None`

The `setup` method can be used to initialize the component by storing the `GeneralInformation` instance and the search space's root node. The default implementation does exactly these two things. This is where you should initialize data structures and variables based on the `CandidateSet` instance in `GeneralInformation`, which is complete at this point and defines all possible circuit configurations that can occur in the search space.

`hasTerminated(): Boolean`

This is a simple query issued at the start of every iteration. It should return `True` if and only if the search has finished and there is nothing left for the component to do. When this happens, the approximation process stops and the output is generated.

Note: Note that by implementing this method incorrectly you can run into an infinite loop, so make sure to double check your termination conditions!

`popNextNode(): Node`

The `popNextNode` method is called once at the start of every iteration and it will return one `Node` instance. The returned instance is validated by the *QualityAssurance* component. Only if it is validated successfully, the other methods of the *Search* component are called in that iteration. With this in mind, you should implement your *Search* component in a way that supports multiple consecutive calls of this function.

In the context of the search space graph, the returned `Node` marks the current position in the search space. Note that, if it is validated successfully, you can use this fact to generate new nodes from this position to explore the search space step by step.

`expandSearchSpace(): List<Node>`

The `expandSearchSpace` method is the first method which is called if a node was successfully validated. It returns a list of newly generated `Node` instances which will then be approximated and analyzed, i.e., they will receive files for their variants and circuits and their `stats` dictionaries will be filled with values for the metrics supported by the *Estimation* component.

To create a better intuition for the purpose of this method, imagine a search space tree with only the current node in it. The `expandSearchSpace` method returns a list of all hidden nodes which are possible candidates for exploration in the next iterations and marks them for the other components so they can discover them and fill them with information. In many cases, it makes sense to simply return the children of the current node.

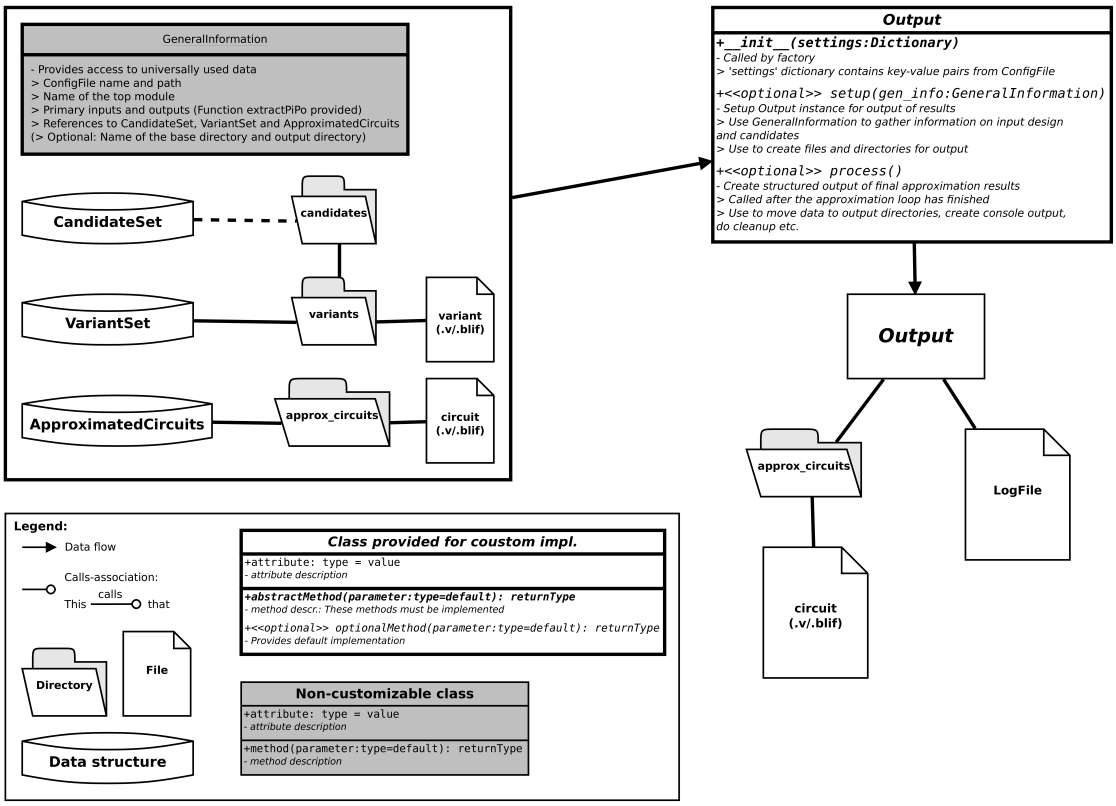
`evaluateNodes(): None`

The `evaluateNodes` method is called at the end of each iteration if the current node was validated successfully. It gives you the chance to process the results of the *Approximation* and *Estimation* stage before `popNextNode` is called again. The `Node`s that were returned by `expandSearchSpace` have now been approximated and filled with statistics to evaluate. As a rule of thumb, in this method you should create a sorted list of all `Node`s you want to validate, using the statistics generated by the *Estimation* component.

Back to the Developer Guide

Exchangeable components

- *Input component*
- *QualityAssurance component*
- *Estimation component*
- *Search component*
- **Output component**



Output component

The *Output* component is responsible for postprocessing the approximation results to create structured and/or filtered output artifacts, e.g., proviing the "best" approximated circuit according to a target metric or providing the Pareto frontier. It can also be used to clean up or remove working directories.

Functions

`setup(gen_info: GeneralInformation): None`

The `setup` method can be used to initialize this component by storing the `GeneralInformation` instance and creating output files and directories. Note that this method is called during CIRCA's start-up phase, and thus, no data structures of the `GeneralInformation` instance are final at this point.

`process(): None`

The `process` method provides the main functionality of the *Output* component. It is the last component method that is called before the framework terminates, so none of the other components will be affected if files or directories are removed or internal data structures are changed.

Back to the Developer Guide

1.3.4 Extensible components

Extensible components have very little in common since they are used for very specialized purposes and not in a generalized manner like the exchangeable stage components. They are used on the same level as many of the specialized data structures which cannot be customized (but may be made customizable in the future). You can create as many custom implementations for these as you like to enrich the framework with new possibilities and use them independently with built in or custom component implementations. However, it can be useful to create approximation methods or error metrics that are especially suitable to use in conjunction with a specific custom component.

Extensible components

- **Approximators**
- *ErrorMetrics*

Approximators

Approximators (or `ApproxMethod` instances) are service providers for the *Approximation* component. Each class represents a specific approach to generate approximated circuits based on quality constraints, e.g., precision scaling. Each *instance*, however, may implement the approach differently based on the configuration options of that approach.

When the `CandidateSet` is generated, `ApproxMethod` instances are generated for each approximation method assigned to it. During the approximation process, the candidate's variants are created by these instances.

Some approximation methods may be incompatible with certain error metrics or combinations of those. To compensate for this, approximators may adjust a variant's configuration of error metrics and bounds in a way that it can be approximated without lowering its error bounds or exceeding their maximum bounds.

Functions

`__init__(id: String, settings: Dictionary)`

> For identification and also representation of approximation methods, each `ApproxMethod` must have a unique class-level ID. The ID should not be too long (2-5 characters) for better readability of circuit configurations and log messages.
>
> The `settings` dictionary can be used to specify the approximator's mode of operation more precisely. You can define the parameters freely and then use the configuration file of the approximation process to specify the options to be used.

`approximateVariant(variant: Variant, gen_info: GeneralInformation): None`

> The `approximateVariant` method provides the main functionality of the approximator. It takes a `Variant` instance as a parameter, generates its approximated circuit and stores it, e.g., in the Verilog and/or the Blif format. The file paths are specified in the `Variant` itself and the circuit from which to start the approximation is the original variant of the candidate the new variant corresponds to.
>
> A variant is unique, i.e., this method is only called if no variant with the same error metrics and approximation method as `variant` has been generated before; otherwise, the variant can be loaded from the variant set.

applyRules(variant: Variant): Boolean

The `applyRules` method is used to ensure that a `Variant`'s configuration of error metrics and bounds is compatible with its approximation method, represented by the `ApproxMethod` instance on which this method is called. It is allowed to increase error bounds of the variant, but not to exceed the maximum bound of an error metric. Error bounds can also be ignored for certain approximation methods, but you should definitely at least generate warning messages if you do so.

The method should return `True` if the variant was successfully adjusted to the approximator's set of rules, and `False` otherwise. Typically, a `False` means that the variant is not considered in the approximation process.

__repr__(): String

The `ApproxMethod` class provides a `__repr__` method for a human-readable representation of the approximation method. In most cases, the default implementation (which outputs the class name and the instance's settings if present) should suffice.

Back to the Developer Guide

Extensible components

- *Approximators*
- **ErrorMetrics**

ErrorMetrics

Error metrics (or `ErrorMetric` instances) provide a general way to specify quality constraints. Each `ErrorMetric` represents a specific way of quantifying the errors produced by an approximated circuit in relation to the original circuit's output. Each *instance*, however, specifies a concrete maximum bound for its metric to serve as candidate-specific quality constraint.

To create an abstract interface for working with error metrics of arbitrary types, the `ErrorMetric` class provides several methods for increasing or comparing error values independently of the respective error metric's value representation.

Functions

__init__(id: String, max_value: Value, step: StepValue)

For identification and representation of different error metrics, each `ErrorMetric` must have a unique class-level ID, just like *approximators*. The `max_value` parameter specifies the maximum error bound for this instance. It has the abstract type `Value` which you can choose freely. The same applies to the `step` parameter, which defines the step size of a value increment.

Note: Note that this method must call the superclass's `__init__` method, which takes an additional argument `default_value` of type `Value`. It specifies the initial value of a new `ErrorMetric` instance.

---

getNextValue(cur_value: Value): Value

This method calculates the next error value as the current value increased by exactly one step. The value is returned, even if it exceeds the maximum error bound of this instance.

greaterMax(value: Value): Boolean

A simple query method to check whether the given error value `value` exceeds the maximum error bound `max_value` of this instance.

getValueStr(value: Value): String

Returns a string representation of the given error value. This is used for ID generation of variants and circuit configurations, so different error values should always generate different string representations. Also make sure to keep the representation as short as possible to avoid overly long IDs.

Back to the Developer Guide

1.3.5 Utils

The CIRCA framework provides some utility modules for working with the configuration file, generating log messages and accessing the file system and OS functions.

Configuration utils `cfg_util`

The `cfg_util` module provides a configuration file parser `CfgParser` to easily get option values or dictionaries of options and their values from a whole section of the config file. It also includes a parser function `parseQualityConstraintSpecifier` to parse the values from a quality constraint specification string as defined in the *user guide's subsection about the configuration file*

Logging utils `logutils`

CIRCA comes with a relatively extensive logging functionality that lets you write logging output to files or to the console with five priority levels. Log messages to the console are also colored based on their level. To include logging in a custom module, simply include this code in the module:

```
import logging
from circa.utils.logutils import IndentationAdapter
log = logging.getLogger("circa.base.<your_module_name>")
log = IndentationAdapter(log, offset=7)
```

Where `<your_module_name>` is the name you want to appear as source of the log message. You can adjust the `offset` parameter of the `IndentationAdapter` to increase or decrease the amount of indentation that is prepended to the message. For writing a log message, use one of the following functions:

```
log.debug("Message of low importance, only for debugging")
log.info("General message for the user, can be ignored safely")
log.warning("Something is not right but the process will continue")
log.error("Something unexpected happened that might lead to a crash")
log.critical("Something very bad happened and the process will be stopped")
```

Note: This is a WIP Beta version. All content is subject to change.

1.4 External programs and frameworks

1.4.1 ABC

ABC is an open-source synthesis tool developed by Berkeley Logic Synthesis and Verification Group. In CIRCA approximation techniques are implemented using ABC.

1.4.2 Yosys

Yosys is developed by Clifford Wolf from TU Vienna. In CIRCA it is used for the synthesis of Verilog files to BLIF netlists.

Note: This is a WIP Beta version. All content is subject to change.

1.5 CIRCA Source Code

1.5.1 base package

base.candidates module

class base.candidates.**Candidate**(*name*, *variants_path*, *orig_var*, *quality_constraints*, *app_methods*)

Bases: abc.ABC

Abstract Base Class for approximation candidate.

Stores all static information about a candidate in the original circuit that is to be approximated.

Parameters

- **name** (*str*) – The *Candidate*'s name used primarily for identification. Should be human-readable (e.g. module name if candidate is a module). *name* and *type* are used to create a unique identifier for the *Candidate*.
- **variants_path** (*str*) – Path to the directory in which the *Candidate*'s variant files will be stored.
- **orig_var** (Variant) – The *Candidate*'s initial instance from the input design.
- **quality_constraints** (*list* of ErrorMetric) – A list of *ErrorMetrics* specifying the error metrics and bounds to be used when approximating the *Candidate*.
- **app_methods** (*list* of ApproxMethod) – A list of *ApproxMethods* to execute the approximation of the *Candidate*.

Inheriting classes must implement the following methods:

__init__()

generateOriginalVariant()

Default implementations are provided for

hasApproxMethod()

hasErrorMetric()

getErrorMetric()

__repr__()

__str__()

It is recommended to use the default implementations unless special behavior of these methods is required.

static getTypeStr()
Returns the *Candidate* type's unique ID string.

name
The *Candidate*'s name used primarily for identification.

Type str

id
Unique identifier of the *Candidate*; is generated from type ID and *name* attribute automatically.

Type str

variants_path
Path to the directory in which the *Candidate*'s variant files will be stored.

Type str

orig_var
The *Candidate*'s initial instance from the input design.

Type Variant

quality_constraints
A list of *ErrorMetrics* specifying the error metrics and bounds to be used when approximating this *Candidate*.

Type *list* of ErrorMetric

app_methods
A list of *ApproxMethods* to execute the approximation of the *Candidate*.

Type *list* of ApproxMethod

pis
A list of the *Candidate*'s primary inputs represented by *Signal* instances.

Type 'list' of Signal

appendPi(*pi*)
Appends a *Signal* to the list of primary inputs.

Parameters **pi** (Signal) – A *Signal* instance representing a primary input of this *Candidate*.

pos
A list of the *Candidate*'s primary outputs represented by *Signal* instances.

Type 'list' of Signal

appendPo(*po*)
Appends a *Signal* to the list of primary outputs.

Parameters **po** (Signal) – A *Signal* instance representing a primary output of this *Candidate*.

generateOriginalVariant(*gen_info*)
Creates and returns the first *Variant* of this *Candidate*.

The original *Variant*'s error bounds will have the default values of this *Candidate*'s error metrics and its approximation method will be the first one on this *Candidate*'s *app_methods* list.

Note: This is an abstract method which must be implemented by inheriting classes.

Parameters **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

hasApproxMethod(*app_method_id*)
Does this *Candidate* have an *ApproxMethod* with the specified ID?

Parameters **app_method_id** (*str*) – The unique identifier of the *ApproxMethod* subclass to look for.

Returns True if an *ApproxMethod* instance with ID *app_method_id* is an element of this *Candidate*'s *app_methods* attribute, otherwise False.

getApproxMethod(*app_method_id*)
Gets the *ApproxMethod* instance with the specified ID if it exists.

Parameters **app_method_id** (*str*) – The unique identifier of the *ApproxMethod* subclass to look for.

Returns The *ApproxMethod* instance from this *Candidate*'s *app_methods* attribute with the specified ID if one exists, None otherwise.

hasErrorMetric(*err_id*)
Does this *Candidate* have an *ErrorMetric* with the specified ID?

Parameters **err_id** (*str*) – The unique identifier of the *ErrorMetric* subclass to look for.

Returns True if an *ErrorMetric* instance with ID *err_id* is an element of this *Candidate*'s *quality_constraints* attribute, otherwise False.

getErrorMetric(*err_id*)
Gets the *ErrorMetric* instance with the specified ID if it exists.

Parameters **err_id** (*str*) – The unique identifier of the *ErrorMetric* subclass to look for.

Returns The *ErrorMetric* instance from this *Candidate*'s *quality_constraints* attribute with the specified ID if one exists, None otherwise.

class base.candidates.**ModuleCand**(*name*, *variants_path*, *orig_var*, *quality_constraints*, *app_methods*)
Bases: *base.candidates.Candidate*

Candidate representation of a module.

This *Candidate* subclass represents approximation candidates specified directly by a module from the input design. Approximation of *ModuleCands* results in the replacement of the original modules by their approximated variants and is therefore especially simple and configurable.

static getTypeStr()
Returns the *Candidate* type's unique ID string.

generateOriginalVariant(*gen_info*)
Overrides the superclass method.

Both Verilog and BLIF files of the generated *Variant* are created.

class base.candidates.**CandidateSet**(*cands=None*, *path=''*)
Bases: object

Data structure to store all *Candidates*.

This class is only instantiated once and serves as universally accessible storage for the approximation candidates.

Parameters

- **cands** (*list* of *Candidate*, optional) – The initial set of *Candidates* if already available. The list may be changed later.
- **path** (*str, optional*) – The path of the directory in which the
- **files are stored. Default value is "".** (*Candidate*) –

candidates
The list in which the *Candidate* instances are stored.

Type *list* of *Candidate*

path
The path of the directory in which the *Candidate* files are stored.

Type str

append(*cand*)
Adds a *Candidate* to the set. Same semantics as `list.append()`.

Parameters **cand** (*Candidate*) – A *Candidate* instance to be added to the set.

extend(*cands*)
Adds a *list* of *Candidates* to the set. Same semantics as `list.extend()`.

Parameters **cands** (*list* of *Candidate*) – A list of *Candidate* instances to be added to the set.

remove(*cand*)
Removes a *Candidate* from the set. Same semantics as `list.remove()`.

Parameters **cand** (*Candidate*) – A *Candidate* instance to be removed from the set.

Raises `ValueError` – If *cand* is not part of the *CandidateSet*.

base.circuits module

class base.circuits.**Circuit**(*node*, *name*, *path*)
Bases: object

Represents the circuit defined by the configuration of a *Node*.

Stores file and validity information as well as a reference to the *Node* representing the circuit.

node
The *Node* representing this circuit.

Type Node

name
A string identifying the circuit uniquely. It is used to generate the circuit's internal ID using a hash function.

Type str

hash
Internal unique identification of this *Circuit*. Calculated as `hash(self.name)`.

Type str

path
The absolute path to the directory in which the circuit files shall be stored.

Type str

file_name
Name of the circuit files without extension. Generated as `self.path + "/" + self.hash`.

Type str

file_verilog
Name of the circuit's Verilog file. Generated as `self.file_name + ".v"`.

Type str

file_blif
Name of the circuit's BLIF file. Generated as `self.file_name + ".blif"`.

Type str

valid
True if and only if this *Circuit* has been validated sucessfully.

Type boolean

validated
Set to *True* after validation.

Type boolean

class base.circuits.**ApproximatedCircuits**(*path*)
Bases: object

This class holds the throught the approx. process generated approx. circuits. A list of approximated circuits, ordered by time of generation.

Parameters **path** (*str*) – The absolute path to the directory in which the approximated circuit files are stored.

path

append(*circuit*)

appendNode(*node*)
Appends the *Circuit* represented by a given *Node*.

base.information module

class base.information.**GeneralInformation**(*cfg_file*, *orig_design*)
Bases: object

Entity class holding all static general information about the circuit and references to *CandidateSet*, *VariantSet* and *ApproximatedCircuits* to be accessed by all stages.

cfg_file
The content of the used configuration file.

Type str

cfg_file_path
The absolute path to the configuration file.

Type str

orig_design
The absolute path to the input circuit file without file extension.

Type str

orig_design_path
The absolute path to the input circuit file's directory.

Type str

orig_design_verilog
The absolute path to the input circuit's Verilog file.

Type str

orig_design_blif
The absolute path to the input circuit's Blif file.

Type str

top_module
The name of the input circuit's top level module.

Type str

pis
A list of the original circuit's primary input signals.

Type *list* of `Signal`

pos
A list of the original circuit's primary output signals.

Type *list* of `Signal`

base_directory
The absolute path to the base working directory in which the input circuit is typically stored.

Type str

output_dir
The absolute path to the directory in which all output files will be stored.

Type str

cand_set
The framework's global *CandidateSet* instance in which the *Candidates* are stored.

Type `CandidateSet`

cand_set_path
The absolute path to the directory in which candidate files are stored.

Type str

variant_set
The framework's global *VariantSet* instance in which all generated *Variants* are stored.

Type `VariantSet`

variant_set_path
The absolute path to the directory in which variant files are stored.

Type str

approx_circuits
The framework's global *ApproximatedCircuits* instance in which all generated *Circuits* are stored.

Type ApproximatedCircuits

approx_circuits_path
The absolute path to the directory in which circuit files are stored.

Type str

timing
The framework's global *TimestampManager* instance that is used to store the most important events and runtimes.

Type TimestampManager

extractPiPo()
Gathers information about the input circuit's primary inputs and outputs and stores it in attributes *pis* and *pos*.

base.metrics package

base.metrics.error_metric module

class base.metrics.error_metric.**ErrorMetric**(*default_value*, *max_value*, *step*)
Bases: abc.ABC

Abstract Base Class for an error metric. Inheriting classes must implement methods:

__init__ getValueStr getNextValue getPrevValue greaterMax

Parameters

- **default_value** – The default error value of the new *ErrorMetric*.
- **max_value** – The maximum error value of the new *ErrorMetric*.
- **step** – The step size, i.e. the rate at which the error value is increased.

Note: All *value* parameters can have arbitrary types to allow for arbitrary error definitions.

---

static metricId()
Returns this *ErrorMetric*'s unique ID string.

id

default_value

max_value

step

getValueStr(*value*)
Returns a human-readable string representation of the given *value*.

getNextValues(*cur_value*)
Returns a list of all possible values resulting from increasing *cur_value* by one step.

getPrevValues(*cur_value*)
Returns a list of all possible values from which *cur_value* can be reached in one step. The list is empty if this is not possible.

greaterMax(*value*)
Returns `True` if *value* exceeds *max_value*, else `False`.

convertToValue(*value*)
Converts the given string representation of a value into the correct value type.

base.metrics.error_metric_factory module

class base.metrics.error_metric_factory.**ErrorMetricFactory**
Bases: `object`

Factory class for creating *ErrorMetric* instances.

Provides functionality to create *ErrorMetric* instances based on their unique class IDs and bound and step parameters.

SUPPORTED_ERROR_METRICS = {'BF': <function ErrorMetricFactory.<lambda>>, 'BM': <functi

static factory(*metric_id*, *bound*, *step=0*)
Creates an *ErrorMetric* instance of the type specified by *metricId*.

The object will be instantiated with the given *bound* and *step* parameters or default *step* value if *step* is not specified.

Parameters

- **metric_id** (*str*) – The unique ID string of the *ErrorMetric* subclass to instantiate.
- **bound** (*float*) – The upper bound for the new *ErrorMetric*'s value.
- **step** (*float*) – The rate at which the value is increased in each step. Default value is 0.

Returns An instance of the *ErrorMetric* subclass specified by *metric_id*.

Return type `ErrorMetric`

base.metrics.error_wc module

class base.metrics.error_wc.**WorstCaseError**(*max_value*, *step*)
Bases: `circa.base.metrics.error_metric.ErrorMetric`

static metricId()
Returns this *ErrorMetric*'s unique ID string.

getNextValues(*cur_value*)
Returns a list of all possible values resulting from increasing *cur_value* by one step.

getPrevValues(*cur_value*)
Returns a list of all possible values from which *cur_value* can be reached in one step. The list is empty if this is not possible.

getValueStr(*value*)
Returns a human-readable string representation of the given *value*.

greaterMax(*value*)
Returns `True` if *value* exceeds *max_value*, else `False`.

convertToValue(*value*)
Converts the given string representation of a value into the correct value type.

base.metrics.error_bf module

class base.metrics.error_bf.**BitFlipError**(*max_value*, *step*)
Bases: circa.base.metrics.error_metric.ErrorMetric

static metricId()
Returns this *ErrorMetric*'s unique ID string.

getNextValues(*cur_value*)
Returns a list of all possible values resulting from increasing *cur_value* by one step.

getPrevValues(*cur_value*)
Returns a list of all possible values from which *cur_value* can be reached in one step. The list is empty if this is not possible.

getValueStr(*value*)
Returns a human-readable string representation of the given *value*.

greaterMax(*value*)
Returns `True` if *value* exceeds *max_value*, else `False`.

convertToValue(*value*)
Converts the given string representation of a value into the correct value type.

base.metrics.error_bm module

class base.metrics.error_bm.**BitMaskError**(*max_value*, *step*)
Bases: circa.base.metrics.error_metric.ErrorMetric

ErrorMetric subclass that represents an error as a number of MSBs and a number of LSBs that together form a bit mask specifying which bits should be replaced with zeros during approximation. Error values are tuples `(num_msbs, num_lsbs)`.

static metricId()
Returns this *ErrorMetric*'s unique ID string.

getNextValues(*cur_value*)
Returns a list of all possible values resulting from increasing *cur_value* by one step.

getPrevValues(*cur_value*)
Returns a list of all possible values from which *cur_value* can be reached in one step. The list is empty if this is not possible.

getValueStr(*value*)
Returns a human-readable string representation of the given *value*.

greaterMax(*value*)
Returns `True` if *value* exceeds *max_value*, else `False`.

convertToValue(*value*)
Converts the given string representation of a value into the correct value type.

base.metrics.error_rel module

class base.metrics.error_rel.**RelativeError**(*max_value*, *step*)
Bases: circa.base.metrics.error_metric.ErrorMetric

static metricId()
Returns this *ErrorMetric*'s unique ID string.

getNextValues(*cur_value*)
Returns a list of all possible values resulting from increasing *cur_value* by one step.

getPrevValues(*cur_value*)
Returns a list of all possible values from which *cur_value* can be reached in one step. The list is empty if this is not possible.

getValueStr(*value*)
Returns a human-readable string representation of the given *value*.

greaterMax(*value*)
Returns `True` if *value* exceeds *max_value*, else `False`.

convertToValue(*value*)
Converts the given string representation of a value into the correct value type.

base.nodes module

class base.nodes.**Node**(*configuration*, *parent=None*, *circuit=None*, *is_root=False*, *state=None*)
Bases: object

This class represents a node in the search space tree, mainly consisting of a list holding a *Variant* for each *Candidate* in the circuit.

Parameters

- **configuration** (*list* of `Variant`) – A list containing the *Variants* of all *Candidates* in the original circuit. The order of the *Variants* must be equal to the order of the respective *Candidates* in the framework's *CandidateSet*.
- **parent** (*Node*, optional) – The *Node*'s parent *Node* in the search space, i.e. the *Node* that creates this new *Node*. Default value is `None`.
- **circuit** (`Circuit`, optional) – The actual *Circuit* represented by this *Node*. Default value is `None`.
- **is_root** (*bool, optional*) – Flag specifying whether this *Node* should be treated as the search space tree's root node. Default value is `False`.
- **state** – Custom attribute in which additional component-specific information about this *Node* can be stored, e.g. whether a *Node* has been visited during search stage.

configuration

parent

id
A string that uniquely identifies this *Node* by its configuration.

Type str

circuit

is_root

state

stats
A dictionary storing estimated or computed statistical information about this *Node*.

Type Dictionary

valid
This *Node*'s *Circuit*'s *valid* attribute. Cannot be read if this *Node* does not have a *Circuit*.

Type bool

validated
Has this *Node*'s *Circuit* been validated? Returns `False` if this *Node* does not have a *Circuit*.

Type bool

getVariantOfCandidate(*cand*)
Returns the *Variant* of a given *Candidate* in the configuration of this *Node*.

Parameters **cand** (Candidate) – The *Candidate* whose *Variant* to return.

Returns The *Variant* corresponding to *cand* if it can be found, otherwise `None`.

generateChildren()
Returns a list of all child *Nodes*. Each child differs from its parent in exactly one *Variant*. The differing *Variants* are created using the `generateChildren()` method of the *Variant*. The resulting *Variants* will generally have higher error values than the *Variants* of this *Node*.

generateNeighbors()
Returns a list of all neighboring *Nodes*. Each neighbor differs from this *Node* in exactly one *Variant*. The differing *Variants* are created using the `generateNeighbors()` method of the *Variant*. The resulting *Variants* will have different, but not necessarily higher error values than the *Variants* of this *Node*.

generateParents()
Returns a list of all *Nodes* that could yield this *Node* as child. Each parent differs from this *Node* in exactly one *Variant*. The differing *Variants* are created using the `generateParents()` method of the *Variant*. The resulting *Variants* will generally have lower error values than the *Variants* of this *Node*.

static generateNodeFromConfiguration(*gen_info*, *config_dict*)
Generates a *Node* instance where for each *Candidate*, a predetermined *Variant* configuration is used. If a configuration is not or only partially specified, the original *Variant*'s values will be used.

Parameters

- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.
- **config_dict** (Dictionary) – A dictionary containing the *Candidates*' names as keys and specification strings for the corresponding *Variants* as values.

Returns A *Node* instance where the *Variants* are configured as specified by the `config_dict`, if their *Candidate*'s name is included as key, or generated like the *Candidate*'s original *Variants*.

base.signals module

class base.signals.**Signal**(*name=''*, *size=None*, *is_signed=0*, *is_reg=0*, *high_active=1*)
Bases: object

Holds information about a signal of a circuit, mainly for input and output signals.

Parameters

- **name** (*str, optional*) – The *Signal*'s name as specified in the input design file. Default value is `""`.
- **size** (*int, optional*) – The bit width of the *Signal*. Default value is `None`.
- **is_signed** (*int, optional*) – Flag specifying whether the *Signal* is signed (`1`) or not (`0`). Default value is `0`.
- **is_reg** (*int, optional*) – Flag specifying whether the *Signal* is a register (`1`) or not (`0`). Default value is `0`.
- **high_active** (*int, optional*) – Flag specifying whether the *Signal* is active on high input (`1`) or low input (`0`). Default value is `1`.

name

size

is_signed

is_reg

high_active

base.variants module

class base.variants.**Variant**(*candidate*, *app_method*, *error_bounds*)

Bases: `object`

This class represents a specific version of an approximated candidate.

Variants are used to synthesize approximated circuits by replacing the *Candidates*' original *Variants* with approximated *Variants*.

Parameters

- **candidate** (`Candidate`) – The *Candidate* this *Variant* belongs to.
- **app_method** (`ApproxMethod`) – An *ApproxMethod* instance that is used to generate the approximated circuit file specified by this *Variant*'s quality constraints.
- **error_bounds** (`Dictionary`) – A dictionary containing *ErrorMetric* IDs as keys and error value bounds as values.

id

A unique string identifying this *Variant*. It is generated using the *Candidate*'s ID and this *Variant*'s quality constraints.

Type str

candidate

path

The absolute path to the directory in which the *Variant* files of this *Variant*'s *Candidate* are stored, i.e. the directory in which this *Variant*'s files are probably stored.

Type str

file_name

The absolute path to this *Variant*'s circuit file without file extension.

Type str

file_verilog

The absolute path to this *Variant*'s Verilog circuit file.

Type str

file_blif
The absolute path to this *Variant*'s Blif circuit file.

Type str

app_method

error_bounds

hasErrorMetric(*err_id*)
Returns whether an *ErrorMetric* with the given ID is in this *Variant*'s error bounds.

Parameters **err_id** (*str*) – The ID of the desired *ErrorMetric* type.

Returns `True` if an *ErrorMetric* of the type with the given ID `err_id` is contained in this *Variant*'s `error_bounds` attribute, otherwise `False`.

generateChildren()
Returns a list of *Variants* so that for each approximation method and error metric there is exactly one *Variant* where either exactly one error bound is increased by one step or the approximation method is different. The error bounds of the metrics are not increased beyond their maximum value and approximation methods are chosen from this *Variant*'s *Candidate*.

generateParents()
Returns a list of *Variants* so that for each approximation method and error metric there is exactly one *Variant* where either exactly one error bound is decreased by one step or the approximation method is different. The error bounds of the metrics are not decreased beyond their minimum value and approximation methods are chosen from this *Variant*'s *Candidate*.

generateNeighbors()
Returns a list of *Variants* so that for each approximation method there is exactly one *Variant* with another approximation method and for each error metric there are two *Variants* where the error bound is increased or decreased by one step. The error bounds are not increased beyond their maximum value and not decreased beyond their minimum value. Approximation methods are chosen from this *Variant*'s *Candidate*.

static generateVariantFromConfiguration(*cand*, *conf*)
Generates a *Variant* instance of a *Candidate* based on a configuration string that specifies error metric values and/or the approximation method. Each non-specified setting will be set as for the *Candidate*'s original *Variant*.

Parameters

- **cand** (`Candidate`) – The *Candidate* instance of which to create a *Variant*.
- **conf** (*str*) – A configuration string specifying the new *Variant*'s quality constraints and/or approximation methods. It consists of semicolon separated `key=value` pairs where `key` is either the key word `ApproxMethod` or the ID of an *ErrorMetric* and `value` is the ID of an *ApproxMethod* or an error value as upper bound.

Example

```
ApproxMethod=PS; BF=2
```

class base.variants.**VariantSet**(*cand_set*)
Bases: `object`

Globally accessible data structure storing all approximated *Variants*.

Parameters **cand_set** (`CandidateSet`) – The framework's final *CandidateSet* instance.

path
The absolute path to the directory in which the *Variant* files are stored.

Type str

append(*variant*)
Adds a *Variant* to the set. Same semantics as `list.append()`.

Parameters **variant** (*Variant*) – A *Variant* instance to be added to the set.

extend(*variants*)
Adds a *list* of *Variants* to the set. Same semantics as `list.extend()`.

Parameters **variants** (*list* of *Variant*) – A list of *Variant* instances to be added to the set.

findVariant(*variant*)
Returns `True` if and only if the given *Variant* is contained in this *VariantSet*.

1.5.2 ext_tools package

ext_tools.ext_tools module

class ext_tools.ext_tools.**ExtTools**(*settings*)
Bases: `object`

This class provides an interface for all stages to utilize external tools based on system configuration.

static abc_dprove(*file_in*)
Uses ABC's *dprove* method to validate a circuit.

Parameters **file_in** (*str*) – An SQCC file with the CUT already in place in Verilog or BLIF format. If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.

Returns `True` if the circuit does not violate the specified quality constraints, otherwise `False`.

Raises `ValueError` – If the input file format is incorrect.

static abc_pdr(*file_in*)
Uses ABC's *pdr* method to validate a circuit.

Parameters **file_in** (*str*) – Verilog or BLIF format. If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.

Returns `True` if the circuit does not violate the specified quality constraints, otherwise `False`.

Raises `ValueError` – If the input file format is incorrect.

static abc_if(*file_in*, *file_out*)
Uses ABC's *if* method to generate statistics about a circuit.

A stat file is created and stored as `file_out`.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format. If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.
- **file_out** (*str*) – The absolute path to the stat file in any writable format (use `*.stat` by convention).

static precisionScaling(*file_in*, *file_out*, *mask*, *tool='abc'*)
Generates an approximated circuit using precision scaling.

The `mask` parameter specifies the number of output bits to lock and their values. Returns the number of changes.

Parameters

- **file_in** (*str*) – The absolute path to the circuit file to be approximated in Verilog or BLIF format. If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **mask** (*str*) – Specifies number and kind of output bits to lock as a string containing only 1s and 0s. The length of the string must equal the output size of the input circuit.
- **tool** (*str, optional*) – Determines which external tool to use (`abc` or `yosys`). Default value is `abc`.

Note: Right now only ABC is supported for this function.

Returns The number of changes made for approximation as returned by the external tool.

Raises `ValueError` – If the specified `tool` is not supported.

static aigRewriting(*qc_file*, *file_in*, *file_out*, *effort=2*, *tool='abc'*)
Generates an approximated circuit using AIG rewriting and returns the number of changes.

Parameters

- **qc_file** (*str*) – The absolute path to the circuit file that specifies the quality constraitns to be used for approximation in Verilog or BLIF format.If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.
- **file_in** (*str*) – The absolute path to the circuit to be approximated in Verilog or BLIF format. If Verilog format is used, Yosys will be used to convert to BLIF. ABC does not support vectors and BLIF is a bit format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **effort** (*int, optional*) – Effort parameter supplied to the external tool. For ABC, it specifies the number of critical paths that are checked: 0: 1 critical path 1: All cricital paths once 2: All critical paths as long as possible
- **tool** (*str, optional*) – Determines which external tool to use (`abc` or `yosys`). Default value is `abc`.

Note: Right now only ABC is supported for this function.

Returns The number of changes made for approximation as returned by the external tool.

Raises `ValueError` – If the specified `tool` is not supported.

static synthesize(*file_in*, *file_out*, *top_module*, *f_noattr=1*, *tool='yosys'*, *cmd='techmap'*)
Synthesizes a circuit by mapping it to a specific technology and writes the resulting circuit to another file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for Yosys (`1`) or not (`0`). Default value is `1`.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.
- **cmd** (*str, optional*) – The command to be used by the used tool. Default value (for Yosys) is `techmap`.

Note: Right now only Yosys with `techmap` command is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static extractAndWriteModule(*file_in*, *file_out*, *top_module*, *module*, *f_noattr=1*, *tool='yosys'*)

Writes a new circuit file containing a module from the input circuit using the given tool.

Parameters

- **file_in** (*str*) – The absolute path to the circuit file from which to extract the module in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the top level module of the input circuit.
- **module** (*str*) – The name of the module to extract.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the result file with Yosys (`1`) or not (`0`). Default value is `1`.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static findCandidates(*file_in*, *top_module*, *file_separated_design*, *file_solutions*, *solution_name*, *pattern*, *tool='yosys'*)

Searches for PGSL pattern in the input circuit and extracts all matches to separate files.

Each match is encapsulated in a Verilog module and written to a solutions file. The modified input design is written to its own file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the top level module of the input circuit.
- **file_separated_design** (*str*) – The absolute path to the file in which the modified input design will be stored in Verilog format.

- **file_solutions** (*str*) – The absolute path to the file in which the found solutions are stored in Verilog format.
- **solution_name** (*str*) – TODO: Add arg description
- **pattern** (*str*) – The PGSL pattern used for the search.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Note: There is currently no Yosys implementation available.

static flattenDesign(*file_in*, *file_out*, *top_module*, *f_techmap=1*, *f_noattr=1*, *tool='yosys'*)
Removes hierarchical structures from a circuit design, i.e. removes modules and subcircuits.

Parameters

- **file_in** (*str*) – The absolute path to the circuit file to be flattened in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **f_techmap** (*int, optional*) – Flag specifying whether the `techmap` command of Yosys should be executed before flattening (`1`) or not (`0`). Default value is `1`.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the result circuit when using Yosys (`1`) or not (`0`). Default value is `1`.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static listModules(*file_in*, *top_module*, *select_pattern=''*, *tool='yosys'*)
Return a list with all module names occurring in a circuit design.

The module names can be selected using a string pattern.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **select_pattern** (*str, optional*) – Specifies the pattern by which module names are selected. If not empty, only names matching the pattern will be included in the result. Default value is `""`.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Returns A list containing all module names in the given circuit design matching the specified pattern.

Return type *list* of str

Raises `ValueError` – If the specified `tool` is not supported.

static renameModule(*file_in*, *file_out*, *module*, *new_module*, *f_noattr=1*, *tool='yosys'*)
Change sthe name of a module in a circuit design and writes the result into another file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **module** (*str*) – The name of the module to be renamed.
- **new_module** (*str*) – The new name for the module.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the result circuit when using Yosys (`1`) or not (`0`). Default value is `1`.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static replaceModules(*file_in*, *file_out*, *top_module*, *files_modules*, *f_noattr=1*, *tool='yosys'*)
Deletes modules from a circuit design, replaces them with circuits from other files and writes the resulting circuit to another file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the output circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the top level module in the input circuit.
- **files_modules** (*list* of *tuple*) – A list containing tuples `(file, module)` where `module` is the name of a module in the input circuit to be replaced and `file` is the absolute path to a circut file containing the replacement module.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the resulting circuit when using Yosys (`1`) or not (`0`). Default value is `1`.
- **tool** (*str*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.

Note: Right now only Yosys is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static readFiles(*files*, *file_out*, *top_module*, *tool='yosys'*, *f_check_hierarchy=False*, *f_flatten=False*, *f_noattr=True*, *f_opt=False*)

Reads a number of circuits and writes them into a single file.

Parameters

- **files** (*list* of str) – List containing absolute paths to the circuit files to read in Verilog or BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog or BLIF format.
- **top_module** (*str*) – The name of the result circuit's top level module. If *f_check_hierarchy* is set to `True`, all input files must have this top module.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.
- **f_check_hierarchy** (*bool, optional*) – Flag specifying whether each input circuit should be checked for the correct top level module (`True`) or not (`False`). Default value is `False`.
- **f_flatten** (*bool, optional*) – Flag specifying whether the result circuit should be flattened (`True`) or not (`False`). Default value is `False`.
- **f_noattr** (*bool, optional*) – Flag specifying whether the `noattr` flag should be used for writing circuits with Yosys (`True`) or not (`False`). Default value is `True`.
- **f_opt** (*bool, optional*) – Flag specifying whether the result circuit should be optimized (`True`) or not (`False`). Default value is `False`.

Note: Right now only Yosys is supported for this function.

Raises `ValueError` – If the specified `tool` is not supported.

static verilogToBlif(*file_in*, *file_out*, *top_module*, *tool='yosys'*, *f_flatten=False*)

Converts a Verilog file to BLIF format.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog format.
- **file_out** (*str*) – The absolute path to the result circuit file in BLIF format.
- **top_module** (*str*) – The name of the top-level module of the input circuit.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.
- **f_flatten** (*bool, optional*) – Flag specifying whether the result circuit should be flattened (`True`) or not (`False`). Default value is `False`.

Raises `ValueError` – If the specified `tool` is not supported.

static blifToVerilog(*file_in*, *file_out*, *top_module*, *tool='yosys'*, *f_flatten=False*, *f_wideports=True*, *f_noattr=True*)

Converts a BLIF file to Verilog format.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in BLIF format.
- **file_out** (*str*) – The absolute path to the result circuit file in Verilog format.

- **top_module** (*str*) – The name of the top-level module of the input circuit.
- **tool** (*str, optional*) – Determines which external tool to use (`yosys` or `abc`). Default value is `yosys`.
- **f_flatten** (*bool, optional*) – Flag specifying whether the result circuit should be flattened (`True`) or not (`False`). Default value is `False`.
- **f_wideports** (*bool, optional*) – Flag specifying whether it should be attempted to restore busses in the result circuit when using Yosys (`True`) or not (`False`). Default value is `True`.
- **f_noattr** (*bool, optional*) – Flag specifying whether the `noattr` flag should be used for writing the result circuit using Yosys (`True`) or not (`False`). Default value is `True`.

Raises `ValueError` – If the input or output file format is incorrect or the specified `tool` is not supported.

static ic3(*file_in*, *top_module*, *name_q*, *f_seq*)

Performs circuit validation using Yosys' `sat` command.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file containing an SQCC in Verilog or BLIF format.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **name_q** (*str*) – The parameter to be added after the `-prove` flag of the `sat` command. TODO: Add proper description here
- **f_seq** (*bool*) – Flag specifying whether the input circuit is sequential (`True`) or not (`False`).

Returns `True` if the validation was successful, `False` otherwise.

Return type bool

static compileVerilog(*source_files*, *compiled_file*, *top_module=''*)

Compiles a list of Verilog files to an executable simulation file.

The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_files** (*list* of str) – A list containing the absolute paths to all Verilog source files.
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is `*.vvp`.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises

- `ValueError` – If no source files are specified.
- `RuntimeError` – If the `iverilog` command exits with an error.

static compileVerilogFileList(*source_list_file*, *compiled_file*, *top_module=''*)

Compiles a list of Verilog files specified in a separate file to an executable simulation file.

The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_list_file** (*str*) – The absolute path to the file in which the Verilog source files are listed. This file must be a readable text file that specifies one source file per line and ends with a blank line (!).
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is `*.vvp`.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises `RuntimeError` – If the `iverilog` command exits with an error.

static compileVerilogMixed(*source_files*, *source_list_file*, *compiled_file*, *top_module=''*)

Compiles a list of Verilog files specified explicitly or in a separate file to an executable simulation file.

Source file names can be specified explicitly or by a file containing one source file name per line. The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_files** (*list* of str) – A list containing the absolute paths to Verilog source files.
- **source_list_file** (*str*) – The absolute path to the file in which other Verilog source files are listed. This file must be a readable text file that specifies one source file per line and ends with a blank line (!).
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is `*.vvp`.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises `RuntimeError` – If the `iverilog` command exits with an error.

static runCompiledVerilog(*compiled_file*, *remove_file=False*)

Runs a compiled Verilog simulation file.

Parameters

- **compiled_file** (*str*) – The absolute path to the compiled simulation file. By convention, simulation files have the format `*.vvp`.
- **remove_file** (*bool*) – Flag specifying whether the compiled file should be removed after the simulation was executed. Default value is `False`.

Raises `RuntimeError` – If the `vvp` command exits with an error.

ext_tools.abc_interface module

`ext_tools.abc_interface.`**generateStat**(*file_in*, *file_out*)

Invokes ABC and generates statistics about a circuit using the command `print_stats -lmpg`.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or Blif format.
- **file_out** (*str*) – The absolute path to the output file in any writable format.

ext_tools.abc_interface.**opt**(*file_in*, *file_out*)
Invokes ABC and optimizes a given circuit design.

Parameters

- **file_in** (*str*) – The absolute path to the circuit file to optimize.
- **file_out** (*str*) – The absolute path to the output file to which the optimized circuit will be written.

Note: If a hierarchy is found in the design it will be flattened, i.e., only the top module remains. This is important if, in later steps, modules should be replaced.

---

ext_tools.abc_interface.**aigRewriting**(*qc_file*, *file_in*, *file_out*, *effort=2*)
Invokes ABC and performs AIG-Rewriting on a given circuit and conforming to given quality constraints.

Parameters

- **qc_file** (*str*) – The absolute path to the file specifying the quality constraints. Passed to the `aig_rewrite` command as first parameter.
- **file_in** (*str*) – The absolute path to the input circuit file. Passed to the `aig_rewrite` command as second parameter.
- **file_out** (*str*) – The absolute path to the output file to which the resulting circuit will be written.
- **effort** (*int, optional*) – Integer in `[0..2]` specifying the effort level to be used for approximation. Is passed to the `aig_rewrite` command as `-E` parameter. Default value is 2.

Returns The number of changes as returned by the `aig_rewrite` command.

Raises `IOError` – If ABC does not behave as expected.

ext_tools.abc_interface.**precisionScaling**(*file_in*, *file_out*, *mask*)
Invokes ABC and performs precision scaling on a given circuit design.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file.
- **file_out** (*str*) – The absolute path to the output file to which the resulting circuit will be written.
- **mask** (*str*) – A string representing the bit mask to be used for precision scaling. It is passed to the `precision_scaling` command as the only parameter and must be formatted correctly for this purpose. Ideally, the string contains only 0s and 1s and its length equals the number of primary outputs of the input circuit.

Returns

The number of changes computed as `n_start - n_end` from the numbers of starting and ending nodes returned by the `precision_scaling` command.

Raises `IOError` – If ABC does not behave as expected.

ext_tools.abc_interface.**dprove**(*file_in*)
Invokes ABC and performs temporal induction to prove sequential equivalence using the `dprove` command.

Parameters **file_in** (*str*) – The absolute path to the input circuit containing an SQCC in Verilog or Blif format.

Returns 1 if the proof was successful, 0 otherwise.

ext_tools.abc_interface.**pdr**(*file_in*)
Invokes ABC and performs temporal induction to prove sequential equivalence using the pdr command.

Parameters **file_in** (*str*) – The absolute path to the input circuit containing an SQCC in Verilog or Blif format.

Returns 1 if the proof was successful, 0 otherwise.

ext_tools.abc_interface.**runCmd**(*cmd*)
Invokes ACB and runs a command using subprocess. The command's return code is returned.

ext_tools.abc_interface.**runScript**(*file*)
Invokes ABC and runs a script file using subprocess. The script's return code is returned.

ext_tools.iverilog_interface module

ext_tools.iverilog_interface.**compileVerilog**(*source_files*, *compiled_file*, *top_module=''*)
Compiles a list of Verilog files to a file that is executable by the vvp command.

The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_files** (*list* of str) – A list containing the absolute paths to all Verilog source files.
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is *.vvp.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises

- ValueError – If no source files are specified.
- RuntimeError – If the iverilog command exits with an error.

ext_tools.iverilog_interface.**compileVerilogFileList**(*source_list_file*, *compiled_file*, *top_module=''*)
Compiles a list of Verilog files specified in a separate file to a simulation file that is executable by the vvp command.

The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_list_file** (*str*) – The absolute path to the file in which the Verilog source files are listed. This file must be a readable text file that specifies one source file per line and ends with a blank line (!).
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is *.vvp.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises RuntimeError – If the iverilog command exits with an error.

ext_tools.iverilog_interface.**compileVerilogMixed**(*source_files*, *source_list_file*, *compiled_file*, *top_module=''*)

Compiles a list of Verilog files to a file that is executable by the `vvp` command.

Source file names can be specified explicitly or by a file containing one source file name per line. The files must be in Verilog format and include all modules that are instantiated. The name of the root module can be specified. It must be specified if the design hierarchy does not have a unique top level module.

Parameters

- **source_files** (*list* of str) – A list containing the absolute paths to Verilog source files.
- **source_list_file** (*str*) – The absolute path to the file in which other Verilog source files are listed. This file must be a readable text file that specifies one source file per line and ends with a blank line (!).
- **compiled_file** (*str*) – The absolute path to the compiled and executable simulation file. Standard format is `*.vvp`.
- **top_module** (*str, optional*) – The name of the module to use as root module to execute. Only this module will be simulated when running the compiled simulation file.

Raises `RuntimeError` – If the `iverilog` command exits with an error.

ext_tools.iverilog_interface.**runCompiledVerilog**(*compiled_file*, *remove_file=False*)

Runs a compiled Verilog simulation file using the `vvp` command.

Parameters

- **compiled_file** (*str*) – The absolute path to the compiled simulation file. By convention, simulation files have the format `*.vvp`.
- **remove_file** (*bool*) – Flag specifying whether the compiled file should be removed after the simulation was executed. Default value is `False`.

Raises `RuntimeError` – If the `vvp` command exits with an error.

ext_tools.yosys_interface module

ext_tools.yosys_interface.**ic3**(*file_in*, *flags*, *top_module*, *f_call_and_remove=1*)

Creates a Yosys script that invokes Yosys and performs circuit validation using the `sat` command.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file containing an SQCC in Verilog or Blif format.
- **flags** (*str*) – String containing flag parameters for the `sat` command.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be executed and then removed (`1`) or not (`0`). This also has an effect on the return value. Default value is `1`.

Returns The exit code of the script execution if `f_call_and_remove` is set to `1`, otherwise the absolute path to the script file. The script is not executed in this case.

ext_tools.yosys_interface.**readFiles**(*files*, *file_out*, *top_module*, *f_call_and_remove=1*, *f_check_hierarchy=1*, *f_flatten=1*, *f_opt=1*, *f_noattr=1*)

Creates a Yosys script that reads a set of files into Yosys, performs a number of optional checks and optimizations and writes the resulting circuit into another file.

Parameters

- **files** (*list* of str) – A list of absolute file paths for circuit files in Verilog of Blif format to read.
- **file_out** (*str*) – The absolute path to the output file to which the resulting circuit is written in Verilog or Blif format.
- **top_module** (*str*) – The name of the top level module of the resulting circuit.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be executed and then removed (`1`) or not (`0`). Default value is `1`.
- **f_check_hierarchy** (*int, optional*) – Flag specifying whether a hierarchy check verifying that `top_module` is the top level module should be executed (`1`) or not (`0`). Default value is (`1`).
- **f_flatten** (*int, optional*) – Flag specifying whether the resulting circuit's hierarchy should be flattened (`1`) or not. Default value is `1`.
- **f_opt** (*int, optional*) – Flag specifying whether the resulting circuit should be optimized (`1`) or not (`0`). Default value is `1`.
- **f_noattr** (*int, optional*) – Flag specifying whether the `-noattr` flag should be used for writing the resulting circuit to a Verilog file (`1`) or not (`0`). Default value is `1`.

Returns The absolute path of the generated script file. Note that the file will be removed if `f_call_and_remove` is set to `1`.

`ext_tools.yosys_interface.`**verilogToBlif**(*file_in*, *file_out*, *top_module=None*, *f_call_and_remove=True*, *f_flatten=False*)

Creates a Yosys script that converts a Verilog circuit file to Blif format.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog format.
- **file_out** (*str*) – The absolute path to the output circuit file in Blif format.
- **top_module** (*str, optional*) – The name of the top level module to write to the Blif file. If set to `None`, the `auto-top` flag will be used to determine the top module name. Default value is `None`.
- **f_call_and_remove** (*bool, optional*) – Flag specifying whether the script should be executed and then removed (`True`) or not (`False`). Default value is `True`.
- **f_flatten** (*bool, optional*) – Flag specifying whether the resulting circuit in Blif format should be flattened (`True`) or not (`False`). Default value is `False`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `True`.

`ext_tools.yosys_interface.`**blifToVerilog**(*file_in*, *file_out*, *top_module*, *f_wideports=True*, *f_noattr=True*, *f_call_and_remove=True*, *f_flatten=False*)

Creates a Yosys script that converts a Blif circuit file to Verilog format.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Blif format.
- **file_out** (*str*) – The absolute path to the output circuit file in Verilog format.

- **top_module** (*str*) – The name of the top level module of the input circuit. Will be used for hierarchy check.
- **f_wideports** (*bool, optional*) – Flag specifying whether the `wideports` flag should be used for reading the Blif file (`True`) or not (`False`). Default value is `True`.
- **f_noattr** (*bool, optional*) – Flag specifying whether the `noattr` flag should be used for writing the Verilog file (`True`) or not (`False`). Default value is `True`.
- **f_call_and_remove** (*bool, optional*) – Flag specifying whether the script should be executed and then removed (`True`) or not (`False`). Default value is `True`.
- **f_flatten** (*bool, optional*) – Flag specifying whether the resulting circuit in Verilog format should be flattened (`True`) or not (`False`). Default value is `False`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `True`.

ext_tools.yosys_interface.**extractAndWriteModule**(*file_in*, *file_out*, *top_module*, *module*, *f_noattr=1*, *f_call_and_remove=1*)

Creates a Yosys script that reads a Verilog circuit file and writes a single module from that file into another Verilog file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog format.
- **file_out** (*str*) – The absolute path to the output circuit file in Verilog format.
- **top_module** (*str*) – The name of the input circuit's top level module.
- **module** (*str*) – The name of the module to extract from the input circuit.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the resulting Verilog file (`1`) or not (`0`). Default value is `1`.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be executed and then removed (`1`) or not (`0`). Default value is `1`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `1`.

ext_tools.yosys_interface.**replaceModules**(*file_in*, *file_out*, *top_module*, *files_modules*, *f_noattr=1*, *f_call_and_remove=1*)

Creates a Yosys script that reads a circuit file into Yosys, replaces a number of modules from that circuit with modules read from other circuit files and writes the result to another file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or Blif format.
- **file_out** (*str*) – The absolute path to the output circuit file in Verilog or Blif format.
- **top_module** (*str*) – The name of the top level module in the input circuit.
- **files_modules** (*list* of *tuple*) – A list containing tuples `(file, module)` where `module` is the name of a module in the input circuit to be replaced and `file` is the absolute path to a circut file containing the replacement module.
- **f_noattr** (*int, optional*) – Flag specifying whether the `noattr` flag should be used for writing the resulting circuit (`1`) or not (`0`). Default value is `1`.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be called and then removed (`1`) or not (`0`). Default value is `1`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `1`.

`ext_tools.yosys_interface.`**`flattenDesign`**(*file_in*, *file_out*, *top_module*, *f_techmap=1*, *f_noattr=1*, *f_call_and_remove=1*)

Creates a Yosys script that flattens the design hierarchy, i.e. removes modules and subcircuits, of a circuit design.

This may be needed as a design pre-processing step for ABC.

Parameters

- **`file_in`** (*`str`*) – The absolute path to the input circuit file in Verilog or Blif format.
- **`file_out`** (*`str`*) – The absolute path to the output circuit file in Verilog or Blif format.
- **`top_module`** (*`str`*) – The name of the top level module of the input circuit.
- **`f_techmap`** (*`int, optional`*) – Flag specifying whether the `techmap` command should be executed before flattening (`1`) or not (`0`). Default value is `1`.
- **`f_noattr`** (*`int, optional`*) – Flag specifying whether the `noattr` flag should be used for writing the resulting circuit file (`1`) or not (`0`). Default value is `1`.
- **`f_call_and_remove`** (*`int, optional`*) – Flag specifying whether the script should be called and then removed (`1`) or not (`0`). Default value is `1`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `1`.

`ext_tools.yosys_interface.`**`techmap`**(*file_in*, *file_out*, *top_module*, *f_noattr=1*, *f_call_and_remove=1*)

Creates a Yosys script that synthesizes a circuit using the `techmap` command.

Parameters

- **`file_in`** (*`str`*) – The absolute path to the input circuit file in Verilog or Blif format.
- **`file_out`** (*`str`*) – The absolute path to the output circuit file in Verilog or Blif format.
- **`top_module`** (*`str`*) – The name of the top level module of the input circuit.
- **`f_noattr`** (*`int, optional`*) – Flag specifying whether the `noattr` flag should be used for writing the resulting circuit file (`1`) or not (`0`). Default value is `1`.
- **`f_call_and_remove`** (*`int, optional`*) – Flag specifying whether the script should be called and then removed (`1`) or not (`0`). Default value is `1`.

Returns The absolute path of the generated script file. Note that the file will be removed if the `f_call_and_remove` flag is set to `1`.

`ext_tools.yosys_interface.`**`renameModule`**(*file_in*, *file_out*, *module*, *new_module*, *f_noattr=1*, *f_call_and_remove=1*)

Creates a Yosys script that renames a module in a circuit and writes the resulting circuit to a file.

Parameters

- **`file_in`** (*`str`*) – The absolute path to the input circuit file in Verilog or Blif format.
- **`file_out`** (*`str`*) – The absolute path to the output circuit file in Verilog or Blif format.
- **`module`** (*`str`*) – The name of the module to rename.
- **`new_module`** (*`str`*) – The new name for the module.
- **`f_noattr`** (*`int, optional`*) – Flag specifying whether the `noattr` flag should be used for writing the resulting circuit file (`1`) or not (`0`). Default value is `1`.

- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be called and then removed (1) or not (0). Default value is 1.

Returns The absolute path of the generated script file. Note that the file will be removed if the f_call_and_remove flag is set to 1.

ext_tools.yosys_interface.**generateStat**(*file_in*, *file_out*, *f_call_and_remove=1*)

Creates a Yosys script that reads a circuit and writes statistics to another file.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or Blif format.
- **file_out** (*str*) – The absolute path to the output circuit file in any writable format.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be called and then removed (1) or not (0). Default value is 1.

Returns The absolute path of the generated script file. Note that the file will be removed if the f_call_and_remove flag is set to 1.

ext_tools.yosys_interface.**listModules**(*file_in*, *top_module*, *select_pattern=''*, *f_call_and_remove=1*)

Creates a Yosys script that reads a circuit design file and lists all modules in the circuit.

The module names are stored in a list and can be filtered by a pattern passed to the select command.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog format.
- **top_module** (*str*) – The name of the top level module of the input circuit.
- **select_pattern** (*str, optional*) – The pattern by which to filter the module names. If not empty, the pattern is passed to the select command. Default value is "".
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script should be called and then removed (1) or not (0). Default value is 1.

Returns A list containing all module names in the input circuit matching the select_pattern.

Return type *list* of str

ext_tools.yosys_interface.**runScript**(*file*, *f_run_quiet=1*, *f_call_and_remove=0*)

Invokes Yosys and runs a script file using a subprocess call.

Parameters

- **file** (*str*) – The absolute path to the script file to execute.
- **f_run_quiet** (*int, optional*) – Flag specifying whether the script shall be executed in quiet mode (i.e. with flag -qq) (1) or print all messages (i.e. with flag -QT) (0). Default value is 1.
- **f_call_and_remove** (*int, optional*) – Flag specifying whether the script file shall be removed after execution (1) or not (0). Default value is 0.

Returns The exit code of the subprocess call.

1.5.3 runtime package

runtime.circa module

runtime.contstants module

class runtime.constants.**DIRS**
Bases: object

Constant global directory names.

CIRCA_REPO = ''
Root directory of the CIRCA framework. Is set dynamically on start.

TEMPLATES = ''
Directory containing template circuits for QualityAssurance. Is set dynamically on start.

APPROX_CIRCUITS = '/approx_circuits'
Name of the directory in which the approximated circuit files will be stored.

CANDIDATES = '/candidates'
Name of the parent directory for the candidates' own directories.

OUTPUT = '/output'
Name of the output directory.

class runtime.constants.**LOG**
Bases: object

Constant default log file names.

APPROX_FILE = 'approx_circuits_log.log'
Name of the log file containing only the most important log entries of each loop iteration.

LOG_FILE = 'log.log'
Name of the log file in which all log entries are stored.

class runtime.constants.**QC**
Bases: object

Global constants for quality constraints.

WORST_CASE = 'WC'
Unique ID of the Worst-Case error metric.

BIT_FLIP = 'BF'
Unique ID of the Bit-Flip error metric.

RELATIVE = 'REL'
Unique ID of the Relative error metric.

BIT_MASK = 'BM'
Unique ID of the Bit-Mask error metric.

SUPPORTED_QCS = ['WC', 'BF', 'REL', 'BM']
List of all error metric IDs.

class runtime.constants.**APPROX**
Bases: object

Global constants for approximation methods.

PRECISION_SCALING = 'PS'
Unique ID of the Precision Scaling approximation technique.

AIG_REWRITING = 'AIG'
Unique ID of the AIG-Rewriting approximation technique.

SUPPORTED_AMS = ['PS', 'AIG']
List of all approximation technique IDs.

METHODS_KEYWORD = 'AppMethods'
Keyword for specifying approximation methods in candidate annotations.

OPTIONS_KEYWORD = 'AMOptions'
Keyword for specifying additional approximation options in candidate annotations.

class runtime.constants.**CFG**
Bases: object

Configuration file section names and keywords.

SECTION_GENERAL = 'General'

SECTION_INPUT = 'Input'

SECTION_SEARCH = 'Search'

SECTION_ESTIMATION = 'Estimation'

SECTION_QA = 'QualityAssurance'

SECTION_APPROX = 'Approximation'

SECTION_OUTPUT = 'Output'

GENERAL_TOP_MODULE = 'TopModule'

INPUT_METHOD = 'Method'

INPUT_KEY = 'Key'

SEARCH_METHOD = 'Method'

ESTIMATION_METHOD = 'Method'

QA_METHOD = 'Method'

QA_QCS = 'QualityConstraints'

QA_CIRCUIT_TYPE = 'CircuitType'

QA_OUTPUT_SIGNAL = 'OutputSignal'

QA_OUTPUT_SIGNED = 'OutputIsSigned'

QA_OUTPUT_CONCAT = 'OutputSignalConcat'

QA_VALID_SIGNAL = 'ValidSignal'

QA_RESET_SIGNAL = 'ResetSignal'

QA_RESET_HIGH_ACTIVE = 'ResetSignalIsHighActive'

QA_START_SIGNAL = 'StartSignal'

QA_START_HIGH_ACTIVE = 'StartSignalIsHighActive'

APPROX_METHOD = 'Method'

APPROX_QCS = 'QualityConstraints'

APPROX_QCS_BOUND = 'bound'

APPROX_QCS_STEP = 'step'

OUTPUT_METHOD = 'Method'

class runtime.constants.**CIRCUIT_TYPE**
Bases: object

Possible values of the CircuitType config option.

COMBINATIONAL = 'combinational'

RUN_TO_COMPLETION = 'run_to_completion'

STREAMING = 'streaming'

class runtime.constants.**SQCC**
Bases: object

TOP_MODULE = 'sqcc'

NAME_VALID = 'valid_sqcc'

NAME_Q = 'q_sqcc'

1.5.4 stages package

front_end package

front_end.front_end module

class front_end.front_end.**FrontEnd**(*settings*)
Bases: abc.ABC

Abstract Base Class for FrontEnd stage. The FrontEnd stage parses the input design and configuration file and initializes data structures for the next stages.

Inheriting classes must implement methods: __init__ setup process

Parameters **settings** (*dict*) – A dictionary using options from the config as keys and strings of their values as values.

__init__(*settings*)
Instantiates a subclass instance using the options from the configuration file.

setup(*cfg_file*, *input_file*)
Sets up the *FrontEnd* instance using config and input files. Provides default implementation.

Parameters

- **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.
- **input_file** (*str*) – The absolute path to the input circuit file in Verilog format.

Raises ValueError – If *cfg_file* or *input_file* is invalid, i.e. None or empty.

process()
Returns a *GeneralInformation* instance: Must generate a *CandidateSet*, *VariantSet* and *ApproximatedCircuits*, set paths and create directories. Must also create Verilog and Blif format files of the original circuit.

Parameters to set in GeneralInformation object: top_module pis pos cand_set variant_set approx_circuits base_directory (optional, default is input file path) output_dir (optional, default is input file path)

front_end.front_end_factory module

class front_end.front_end_factory.**FrontEndFactory**
Bases: object

Factory class for creating FrontEnd instances.

Add an entry `"MyFrontEnd": lambda s: MyFrontEnd(s)` in `SUPPORTED_SUBCLASSES` for the factory to recognize your own implementation.

SUPPORTED_SUBCLASSES = {'AnnotatedCandidates': <function FrontEndFactory.<lambda>>}

static factory(*cfg_file*)
Parses the Method parameter from the configuration file and returns the corresponding subclass instance if supported.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.

Returns An instance of the *FrontEnd* subclass specified by the Method parameter in the configuration file's FrontEnd section.

Return type FrontEnd

Raises NotImplementedError – If the subclass ID from the configuration file cannot be recognized.

front_end.front_end_anno_cand module

class front_end.front_end_anno_cand.**AnnotatedCandidates**(*settings*)
Bases: circa.stages.front_end.front_end.FrontEnd

FrontEnd subclass that reads candidate information from annotations in the original design file.

Candidates in the Verilog design are annotated with a key word and optional approximation settings that are placed between the `module` keyword and the module name. The annotation keyword can be set in the configuration file. Default is `ANNOTATION_KEYWORD = r'<<<APPROX_CAND>>>'`.

Raises ValueError – If there is no *Key* keyword in the `settings` dict, i.e. in the configuration file.

class ANNOTATION_REGEX_GROUP
Bases: object

Regex group names used to find and parse annotations.

ENTIRE_MATCH = 'gEntireMatch'

APPROX_KEY = 'gApproxKey'

APPROX_PARAMETERS = 'gApproxParameters'

MODULE_NAME = 'gModuleName'

MODULE_PORTS = 'gModulePorts'

setup(*cfg_file*, *input_file*)
Sets up the *FrontEnd* instance using config and input files. Provides default implementation.

Parameters

- **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.
- **input_file** (*str*) – The absolute path to the input circuit file in Verilog format.

Raises `ValueError` – If *cfg_file* or *input_file* is invalid, i.e. `None` or empty.

process()
Returns a *GeneralInformation* instance: Must generate a *CandidateSet*, *VariantSet* and *ApproximatedCircuits*, set paths and create directories. Must also create Verilog and Blif format files of the original circuit.

Parameters to set in GeneralInformation object: top_module pis pos cand_set variant_set approx_circuits base_directory (optional, default is input file path) output_dir (optional, default is input file path)

_extractAnnotationsAndCleanDesign()
Parses the input file and searches for annotations in the code. Puts every annotation found into comments and stores the cleaned design. A key word is used to recognize annotations. The key is set in the config.

Returns A list containing a dictionary for each annotation found in the input design file. Every dictionary has all keys specified in `ANNOTATION_REGEX_GROUP`.

Return type *list* of *dict*

Raises `RuntimeError` – If no annotations were found in the input circuit file.

_generateCandidates(*cand_set*)
Uses annotations to extract candidate information, instantiates *Candidate* objects and stores them in the *CandidateSet*.

Parameters **cand_set** (`CandidateSet`) – *CandidateSet* instance in which the *Candidates* are stored.

quality_assurance package

quality_assurance.quality_assurance module

class quality_assurance.quality_assurance.**QualityAssurance**(*settings*)
Bases: `abc.ABC`

Abstract Base Class for QualityAssurance stage. The QualityAssurance stage validates approximated circuits to ensure that their quality constraints are not violated.

Inheriting classes must implement methods: __init__ setup validateCircuit

Parameters **settings** (*dict*) – A dictionary using options from the config as keys and strings of their values as values.

setup(*gen_info*)
Sets up the *QualityAssurance* instance using a *GeneralInformation* instance. Provides default implementation

Parameters **gen_info** (`GeneralInformation`) – The framework's *GeneralInformation* instance.

validateCircuit(*node*)
Validates the circuit represented by a *Node* by trying to verify that the circuit meets all its quality constraints.

Parameters **node** (`Node`) – The *Node* to validate.

Returns `True` if the circuit of the given *Node* is valid, otherwise `False`.

Return type bool

quality_assurance.quality_assurance_factory module

quality_assurance.quality_assurance_abc_dprove module

class quality_assurance.quality_assurance_abc_dprove.**ABCDprove**(*settings*)
Bases: circa.stages.quality_assurance.quality_assurance.QualityAssurance

QualityAssurance subclass using the `dprove` command of ABC for circuit validation.

SUPPORTED_CIRCUIT_TYPES = ['combinational', 'run_to_completion', 'streaming']

SUPPORTED_ERROR_METRICS = ['BF', 'WC']

C_SQCC_TOP_MODULE = 'sqcc'

top_module
The name of the input circuit's top level module.

Type str

setup(*gen_info*)
Sets up the *QualityAssurance* instance using a *GeneralInformation* instance. Provides default implementation

Parameters **gen_info** (`GeneralInformation`) – The framework's *GeneralInformation* instance.

validateCircuit(*node*)
Validates the circuit represented by a *Node* by trying to verify that the circuit meets all its quality constraints.

Parameters **node** (`Node`) – The *Node* to validate.

Returns `True` if the circuit of the given *Node* is valid, otherwise `False`.

Return type bool

_addCutToSqcc(*node*, *test_design*)
Adds the circuit to be validated to the SQCC and writes it into a file.

Parameters

- **node** (`Node`) – The *Node* instance representing the circuit to validate.
- **test_design** (*str*) – The absolute path to the circuit file in which the SQCC will be stored in Verilog or Blif format.

_setupSQCC()
Creates and writes the SQCC file implementing a QEC with the global quality constraints specified in the config. The input circuit is flattened and inserted as reference circuit.

Returns The absolute path to the SQCC circuit file in Verilog format.

Return type str

_writeSqcc()
Writes the SQCC file template into which the input circuit and the QEC will be inserted and returns its file name. Modules have to be loaded afterwards.

Returns The absolute path to the SQCC circuit file in Verilog format.

Return type str

_setupQec()
Writes a quality evaluation circuit implementing the global quality constraints specified in the config to a file and returns its file name.

Returns The absolute path to the QEC circuit file in Verilog format.

Return type str

Raises `ValueError` – If relative error is used in the quality constraints (Relative error is not supported yet).

_prepareQecWrapper()
Exchanges regexes with information gathered from the original circuit.

Returns Content of the modified QEC file as a string.

Return type str

_prepareWcQc(*msb_0*, *lsb_1*, *msb_1*, *error_wc*, *is_signed*, *mod_suffix=''*)
Loads template for worst-case QC and parses regexes.

Returns Content of the modified QC file as a string.

Return type str

_prepareRelErrQc(*msb_0*, *lsb_1*, *msb_1*, *lower_bound*, *upper_bound*, *mod_suffix=''*)
Loads template for relative error QC and parses regexes.

Returns Content of the modified QC file as a string.

Return type str

_prepareBitFlipErrQc(*msb_0*, *lsb_1*, *msb_1*, *max_bit_flip*, *mod_suffix=''*)
Loads template for bit flip error QC and parses regexes.

Returns Content of the modified QC file as a string.

Return type str

static _findAndReplace(*file*, *repl_pairs*)
Replaces all occurrences of `key` with `repl` for every `(key, repl)` pair in *repl_pairs* in each line of the given file and returns the lines of the file as a string.

Parameters

- **file** (*File*) – File handle for the file to read from.
- **repl_pairs** (*list* of (*tuple* of str)) – A list containing the `(key, repl)` pairs as tuples of strings.

Returns The content of the input file with all specified pairs replaced as a string.

Return type str

quality_assurance.quality_assurance_abc_pdr module

class quality_assurance.quality_assurance_abc_pdr.**ABCPdr**(*settings*)
Bases: `circa.stages.quality_assurance.quality_assurance_abc_dprove.ABCDprove`

QualityAssurance subclass using the `pdr` command of ABC for circuit validation.

validateCircuit(*node*)
Validates the circuit represented by a *Node* by trying to verify that the circuit meets all its quality constraints.

Parameters **node** (Node) – The *Node* to validate.

Returns True if the circuit of the given *Node* is valid, otherwise False.

Return type bool

quality_assurance.quality_assurance_ic3 module

class quality_assurance.quality_assurance_ic3.**IC3**(*settings*)
Bases: circa.stages.quality_assurance.quality_assurance_abc_dprove.ABCDprove

QualityAssurance subclass using the sat command of Yosys for circuit validation.

validateCircuit(*node*)
Validates the circuit represented by a *Node* by trying to verify that the circuit meets all its quality constraints.

Parameters **node** (Node) – The *Node* to validate.

Returns True if the circuit of the given *Node* is valid, otherwise False.

Return type bool

quality_assurance.quality_assurance_testing module

approximation package

approximation.approximation module

class approximation.approximation.**Approximation**(*cfg_file*)
Bases: object

Singleton class for finishing creation and ensuring validity of the *CandidateSet*. Provides functionality to delegate approximation tasks to specific approximators.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file, ending on *.cfg*.

Raises ValueError – If no default approximation methods or quality constraints are specified in the configuration file.

SUPPORTED_ERROR_METRICS = ['WC', 'BF', 'BM']

__init__(*cfg_file*)
Parses app_methods and quality constraints from config file.

static _parseApproxMethods(*cfg_file*)
Parses default approximation methods from the config file and returns a list containing an *ApproxMethod* instance for each specified approximation method without redundancies.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file, ending on *.cfg*.

Returns A list containing an *ApproxMethod* instance for each approximation method ID found in the *Method* option of the configuration file's *Approximation* section. At most one *ApproxMethod* instance of each type is included.

Return type *list* of ApproxMethod

static _parseQualityConstraints(*cfg_file*)
Parses default quality constraints from the config file and returns a list containing an *ErrorMetric* instance for each specified quality constraint without redundancies.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file, ending on *.cfg*.

Returns A list containing an *ErrorMetric* instance for each error metric ID found in the *QualityConstraints* option of the configuration file's *Approximation* section. At most one *ErrorMetric* instance of each type is included.

Return type *list* of ErrorMetric

setup(*gen_info*)
Sets up Approximation instance using a *GeneralInformation* instance. Sets default *app_methods* and default *quality_constraints* attributes on each *Candidate* in *cand_set* if attribute is empty and returns the root node.

Parameters **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

Returns The root node of the search space tree.

Return type Node

Raises ValueError – If a candidate is left without approximation methods or quality constraitns after incompatibilities were removed.

performApproximations(*nodes*)
Generates approximated versions of all *Variants* in all *Nodes* in a given list and synthesizes the resulting circuits.

Parameters **nodes** (*list* of Node) – A list of *Node* instances whose circuits will be approximated.

_getRootNode()
Creates and returns the root node of the search space tree and adds its *Variants* to the *VariantSet*.

Returns The root node of the search space tree.

Return type Node

static _autoDetectBoundaries(*cand*)
Automatically detects quality boundaries for a *Candidate*. PO information about the *Candidate* is required.

Parameters **cand** (Candidate) – The *Candidate* to check for boundaries.

approximation.approx_method module

class approximation.approx_method.**ApproxMethod**(*settings=None*)
Bases: abc.ABC

Abstract Base Class for Approximation Methods.

Inheriting classes must implement methods: methodId __init__ approximateVariant applyRules

static methodId()
Returns this *ApproxMethod*'s unique ID string.

id

settings

path_template

approximateVariant(*variant*, *gen_info*)
Generates approximated variant and stores it in a file.

Parameters

- **variant** (Variant) – The *Variant* instance to be approximated.
- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

applyRules(*variant*)
Checks if the given *Variant*'s error bounds are consistent with this *ApproxMethod*'s set of rules and adjusts them accordingly if not. Error bounds must not be decreased in order to avoid infinite loops and redundant steps in Variant generation.

Parameters **variant** (Variant) – The *Variant* to apply the rules to.

Returns True if the rules have been applied correctly and the *Variant* has a valid configuration afterwards, False if the rules could not be applied.

Return type bool

approximation.approximator_factory module

class approximation.approximator_factory.**ApproximatorFactory**
Bases: object

Factory class for creating *ApproxMethod* instances.

Provides functionality to create *ApproxMethod* instances based on their unique class IDs and settings dictionaries.

SUPPORTED_APP_METHODS = {'AIG': <function ApproximatorFactory.<lambda>>, 'PS': <functi

static factory(*method_id*, *settings=None*)
Creates an *ApproxMethod* instance of the type specified by *methodId*.

The *settings* attribute of the *ApproxMethod* can be set as well. If no *ApproxMethod* of the given *methodId* is known, a *NotImplementedError* will be raised.

Parameters

- **method_id** (*str*) – The unique ID string of the *ApproxMethod* subclass to instantiate.
- **settings** (Dictionary) – A dictionary containing arbitrary settings for the new *ApproxMethod*.

Returns An instance of the *ApproxMethod* subclass specified by *method_id*.

Return type ApproxMethod

Raises NotImplementedError – If *method_id* matches none of the supported *ApproxMethod* subclasses.

approximation.approx_method_ps module

class approximation.approx_method_ps.**ApproximatorPS**(*settings=None*)
Bases: circa.stages.approximation.approx_method.ApproxMethod

static methodId()
Returns this *ApproxMethod*'s unique ID string.

approximateVariant(*variant*, *gen_info*)
Generates approximated variant and stores it in a file.

Parameters

- **variant** (Variant) – The *Variant* instance to be approximated.
- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

applyRules(*variant*)
Ensures that error bounds of metrics BitFlip and WorstCase will not create bit masks of different sizes by increasing the lower value until it matches the larger. This way, each increment of any of the two error bounds will increase the bit mask by one and therefore no bit mask will be applied twice for approximation. Variants which do not have both of these error metrics will not be changed.

If the options 'PSPowerTwoSteps' and *PSPreciseMask* are set to 'True', only WC error will be adjusted to fill the resulting bit mask with ones. Note that the resulting increase of the WC value may not be a multiple of the *step* attribute of the WC error metric.

_generateMask(*cand*, *variant*)
Generates a bit-mask for precision scaling and returns it as string.

Parameters

- **cand** (Candidate) – The *Candidate* for which to create the bit mask.
- **variant** (Variant) – The *Variant* specifying the quality constraints.

Returns The bit mask as a string (LSB...MSB) containing only 1s and 0s and of length equal to the *Candidate*'s output size.

Return type str

approximation.approx_method_aig module

class approximation.approx_method_aig.**ApproximatorAIG**(*settings=None*)
Bases: circa.stages.approximation.approx_method.ApproxMethod

static methodId()
Returns this *ApproxMethod*'s unique ID string.

approximateVariant(*variant*, *gen_info*)
Generates approximated variant and stores it in a file.

Parameters

- **variant** (Variant) – The *Variant* instance to be approximated.
- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

applyRules(*variant*)
Checks if the given *Variant*'s error bounds are consistent with this *ApproxMethod*'s set of rules and adjusts them accordingly if not. Error bounds must not be decreased in order to avoid infinite loops and redundant steps in Variant generation.

Parameters **variant** (Variant) – The *Variant* to apply the rules to.

Returns True if the rules have been applied correctly and the *Variant* has a valid configuration afterwards, False if the rules could not be applied.

Return type bool

_prepareQcWrapper(*candidate*)
Exchanges regexes by information gathered from candidate.

_prepareWcQc(*msb_0*, *lsb_1*, *msb_1*, *error_wc*, *is_signed*, *mod_suffix=''*)
Loads template for worst-case QC and parses regexes.

_prepareRelErrQc(*msb_0*, *lsb_1*, *msb_1*, *lower_bound*, *upper_bound*, *mod_suffix=''*)
Loads template for relative error QC and parses regexes.

_prepareBitFlipErrQc(*msb_0*, *lsb_1*, *msb_1*, *max_bit_flip*, *mod_suffix=''*)
Loads template for bit flip error QC and parses regexes.

_findAndReplace(*file*, *repl_pairs*)
Replaces all occurrences of key with repl for every (key, repl) pair in repl_pairs in each line of the given file and returns lines as string.

Parameters

- **file** (*file handle*) – The file in which to find and replace the
- **pairs of strings.** (*given*) –
- **repl_pairs** (*list* of *tuple* (str, str)) – A list of string tuples
- **replace) specifying which strings** (*(find,)* –
- **which other strings** (*with*) –

Returns A string containing the content of the given file after all replacements.

Return type str

estimation package

estimation.estimation module

class estimation.estimation.**Estimation**(*gen_info*)
Bases: object

This class provides functions for generating and parsing estimated statistics of approximated circuit configurations.

estimateCircuitStats(*nodes*)
Adds a dictionary {"area", "delay", "power", "error"} created using external tools and parsing stat files to each node.

Parameters

- **nodes** (*list* of Node) – A list of *Node* instances the stats
- **which will be estimated.** (*of*) –

Raises RuntimeError – If this *Estimation* instance's *method* is unknown.

_abc_if(*file_in*, *file_out*, *keys=None*)
Generates a stat file for *file_in* using external tools and parses the keys from the file. Returns a *Dictionary* with *keys* as keys and values converted to floats.

Parameters

- **file_in** (*str*) – The absolute path to the input circuit file in Verilog or Blif format.
- **file_out** (*str*) – The absolute path to the output file in any writable format (use *.stat by convention).
- **keys** (*list* of str) – List of keywords to parse. Default value is ["area", "delay", "power", "error"].

search package

search.search module

class search.search.**Search**(*settings*)
Bases: abc.ABC

Abstract Base Class for Search stage. The Search stage defines which method will be used to traverse the search space tree and when the search has finished.

Inheriting classes must implement methods: __init__ setup hasTerminated popNextNode expandSearchSpace evaluateNodes

Parameters **settings** (*dict*) – A dictionary using options from the config as keys and strings of their values as values.

setup(*gen_info*, *root_node*)
Sets up the *Search* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters

- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.
- **root_node** (Node) – A *Node* instance which is the search space tree's root.

hasTerminated()
Returns whether the search is finished or not.

popNextNode()
Returns the *Node* that is to be validated next. Can return None if no more nodes exist.

Returns

The next *Node* to be validated or None if no such *Node* exists.

Return type Node

expandSearchSpace()
Expands the search space tree and returns a list of all new *Nodes* to be approximated.

Returns

A list containing all *Nodes* that need to be approximated.

Return type *list* of Node

evaluateNodes()
Uses search-specific heuristics and circuit stats to evaluate nodes for *popNextNode*.

search.search_factory module

search.search_gradient_descent module

class search.search_gradient_descent.**GradientDescent**(*settings*)
Bases: circa.stages.search.search.Search

Search subclass implementing a gradient descent search where the gradients are computed as the area change of a *Node* with respect to its parent *Node*.

class SEL_STRATEGY
Bases: object

Identifiers for selection strategies.

BEST = 'best'

RND = 'random'

DEFAULT = 'best'

SUPPORTED = ['best', 'random']

class CFG
Bases: object

Parser settings for the configuration file options.

EFFORT = {'mandatory': True, 'parameter': 'Effort'}

SEL_STRATEGY = {'mandatory': False, 'parameter': 'SelectStrategy'}

__init__(*settings*)
Extends __init__ of *Search*.

Raises RuntimeError – If a configuration option is missing or its value cannot be parsed or is not supported.

setup(*gen_info*, *root_node*)
Sets up the *Search* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters

- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.
- **root_node** (Node) – A *Node* instance which is the search space tree's root.

hasTerminated()
Returns whether the search is finished or not.

popNextNode()
Returns the *Node* that is to be validated next. Can return None if no more nodes exist.

Returns

The next *Node* to be validated or None if no such *Node* exists.

Return type Node

expandSearchSpace()
Expands the search space tree and returns a list of all new *Nodes* to be approximated.

Returns

A list containing all *Nodes* that need to be approximated.

Return type *list* of Node

evaluateNodes()
Overrides *evaluateNodes* of abstract super class *Search*.

Raises RuntimeError – If an invalid effort level is set.

search.search_simulated_annealing module

class search.search_simulated_annealing.**SimulatedAnnealing**(*settings*)
Bases: circa.stages.search.search.Search

class HEURISTIC
Bases: object

Heuristic IDs.

AREA = 'area'

DEFAULT = 'area'

SUPPORTED = ['area']

class CFG
Bases: object

Parser settings for the configuration file options.

T_MIN = {'mandatory': True, 'parameter': 'TMin'}

ALPHA = {'mandatory': True, 'parameter': 'Alpha'}

EQUILIBRIUM = {'mandatory': True, 'parameter': 'Equilibrium'}

HEURISTIC = {'mandatory': False, 'parameter': 'Heuristic'}

__init__(*settings*)
Extends __init__ of *Search*.

Raises RuntimeError – If a configuration option is missing or its value cannot be parsed or is not supported.

setup(*gen_info*, *root_node*)
Sets up the *Search* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters

- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.
- **root_node** (Node) – A *Node* instance which is the search space tree's root.

hasTerminated()
Returns whether the search is finished or not.

acceptanceProbability(*node_new*, *node_old*)

popNextNode()
Returns the *Node* that is to be validated next. Can return None if no more nodes exist.

Returns

The next *Node* to be validated or None if no such *Node* exists.

Return type Node

expandSearchSpace()
Expands the search space tree and returns a list of all new *Nodes* to be approximated.

Returns

A list containing all *Nodes* that need to be approximated.

Return type *list* of Node

evaluateNodes()
Uses search-specific heuristics and circuit stats to evaluate nodes for *popNextNode*.

search.search_mcts module

class search.search_mcts.**MCTS**(*settings*)
Bases: circa.stages.search.search.Search

class CFG
Bases: object

SCALER = {'mandatory': True, 'parameter': 'Scaler'}

BUDGET = {'mandatory': True, 'parameter': 'Budget'}

__init__(*settings*)
Extends __init__ of *Search*.

Raises RuntimeError – If a configuration option is missing or its value cannot be parsed or is not supported.

setup(*gen_info*, *root_node*)
Sets up the *Search* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters

- **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.
- **root_node** (Node) – A *Node* instance which is the search space tree's root.

hasTerminated()
Returns whether the search is finished or not.

utcSelect(*list_of_nodes*)

hasOpenList(*node*)

hasCloseList(*node*)

hasActiveChildren(*node*)

getMaxRewardChildren(*node*)

popNextNode()
Returns the *Node* that is to be validated next. Can return None if no more nodes exist.

Returns

The next *Node* to be validated or None if no such *Node* exists.

Return type Node

expandSearchSpace()
Expands the search space tree and returns a list of all new *Nodes* to be approximated.

Returns

A list containing all *Nodes* that need to be approximated.

Return type *list* of Node

backPropagate(*node*, *value*)

evaluateNodes()
Uses search-specific heuristics and circuit stats to evaluate nodes for *popNextNode*.

search.search_jump_search module

search.search_initial_var_dfs module

class search.search_initial_var_dfs.**InitialVariantDFS**(*settings*)
Bases: circa.stages.search.search.Search

Search implementation that accepts parameters to specify the starting node of a depth-first search. An optional budget can also be specified to limit the number of visited nodes.

class CFG
Bases: object

Configuration options.

INITIAL_VAR_PARAM = 'InitialVariants'

BUDGET_PARAM = 'Budget'

setup(*gen_info*, *root_node*)
Creates DFS starting node by applying initial variant parameters.

hasTerminated()
Returns whether the search is finished or not.

popNextNode()
Returns the *Node* that is to be validated next. Can return None if no more nodes exist.

Returns

The next *Node* to be validated or None if no such *Node* exists.

Return type Node

expandSearchSpace()
Expands the search space tree and returns a list of all new *Nodes* to be approximated.

Returns

A list containing all *Nodes* that need to be approximated.

Return type *list* of Node

evaluateNodes()
Uses search-specific heuristics and circuit stats to evaluate nodes for *popNextNode*.

output package

output.output module

class output.output.**Output**(*settings*)
Bases: abc.ABC

Abstract Base Class for Output stage. The Output stage returns the approximation results and finishes the program flow.

Inheriting classes must implement methods: __init__ setup process

Parameters **settings** (*dict*) – A dictionary using options from the config as keys and strings of their values as values.

setup(*gen_info*)
Sets up the *Output* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

process()
Outputs formatted results using information stored in *gen_info*.

output.output_factory module

class output.output_factory.**OutputFactory**
Bases: object

Factory class for creating Output instances.

Add an entry `"MyOutput": lambda s: MyOutput(s)` in SUPPORTED_SUBCLASSES for the factory to recognize your own implementation.

SUPPORTED_SUBCLASSES = {'DummyOutput': <function OutputFactory.<lambda>>, 'output_bes

static factory(*cfg_file*)
Parses the Method parameter from the configuration file and returns the corresponding subclass instance if supported.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.

Returns An instance of the *Output* subclass specified by the Method parameter in the configuration file's *Output* section.

Return type Output

Raises NotImplementedError – If the subclass ID from the configuration file cannot be recognized.

output.output_best_area module

class output.output_best_area.**OutputBestArea**(*settings*)
Bases: circa.stages.output.output.Output

Output subclass that prints a table containing all valid approximated circuits sorted by area.

setup(*gen_info*)
Sets up the *Output* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

process()
Outputs formatted results using information stored in *gen_info*.

output.output_dummy module

class output.output_dummy.**DummyOutput**(*settings*)
Bases: circa.stages.output.output.Output

Basic Output subclass demonstrating how the Output stage can be used.

setup(*gen_info*)
Sets up the *Output* instance using a *GeneralInformation* instance. Provides a default implementation.

Parameters **gen_info** (GeneralInformation) – The framework's *GeneralInformation* instance.

process()
Outputs formatted results using information stored in *gen_info*.

stages.quaes module

1.5.5 utils package

utils.cfg_util module

class utils.cfg_util.**CfgParser**
Bases: object

parsers = {}

static getConfigParser(*cfg_file*)
Tries to open the configuration file and return a *RawConfigParser*. Prints error message and raises exception if config file is not found. A *RawConfigParser* is created once for each config file for which any *CfgParser* function is called.

Parameters **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.

Returns A static *RawConfigParser* instance that is assigned to the specified configuration file.

Return type RawConfigParser

Raises ValueError – If no configuration file for *cfg_file* is found.

static parseOption(*cfg_file*, *section*, *option*, *is_optional=False*)
Tries to open the config file and parse for the given section and option. Prints error message and raises exception if config file, section or option is not found.

Parameters

- **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.
- **section** (*str*) – The name of the section to parse from the config file.
- **option** (*str*) – The name of the option to parse and return from the section.
- **is_optional** (*bool, optional*) – Is the option to parse optional?

Returns The value of the configuration's option with the specified name in the specified section.

Return type str

Raises ValueError – If the specified section or option is not found.

static getOptionsDict(*cfg_file*, *section*)
Tries to open the config file, parse the given section and return its options as a dictionary. Prints error message and raises exception if config file or section is not found.

Parameters

- **cfg_file** (*str*) – The absolute path to the configuration file in **.cfg* format.
- **section** (*str*) – The name of the section to parse from the config file.

Returns A dictionary containing all option-value pairs from the specified configuration section.

Return type dict

Raises `ValueError` – If the specified section is not found.

`utils.cfg_util.`**parseQualityConstraintSpecifier**(*spec*)

Reads a quality constraint specifier expression and returns a dictionary containing error metric names or the approx. methods keyword as keys and dictionaries with *bound* and *step* values or a list of approx. method IDs as values. The dictionary may also contain an *AMOptions* key with a dictionary of arbitrary content as value.

Error messages will be logged if there are parsing errors. Each error metric or approximation method will only appear once in the result.

Example specifier expression: `BF:bound=4;WC:step=5;AppMethods:PS`

Parameters **spec** (*str*) – The quality constraint specifier expression to parse.

Returns A dictionary containing all information from the specifier string in a format that is easy to access.

Return type dict

utils.logutils module

class `utils.logutils.`**IndentationAdapter**(*logger*, *extra=None*, *padding='t'*, *offset=6*)

Bases: `logging.LoggerAdapter`

Custom LoggerAdapter adding indentation to log messages depending on the nesting depth of the calling function.

Parameters

- **logger** (`Logger`) – The *Logger* instance to use as a base for the adapter.
- **extra** (*dict, optional*) – Special argument for *LoggerAdapter*, is not used here. Default value is `{}`.
- **padding** (*str, optional*) – The string to be used as padding. Default value is `" "`.
- **offset** (*int, optional*) – Indentation is reduced by *offset* levels. Default value is `6`.

_indent()

Returns the indentation level using the current size of the stack and the specified offset. Minimum indentation is 0.

process(*msg*, *kwargs*)

Inserts indentation into message string.

class `utils.logutils.`**ColoredFormatter**(*fmt*, *datefmt=None*, *style='%'*, *use_color=True*)

Bases: `logging.Formatter`

Custom Formatter adding color to log messages depending on the level of the message. DEBUG – dark white INFO – white WARNING – yellow ERROR – red CRITICAL – red

format(*record*)

Adds color escapes before passing the record to the default format method. Messages with level higher than INFO are printed bold.

utils.shutil2 module

utils.shutil2.**environ**(*var*)

utils.shutil2.**chdir**(*path*)

utils.shutil2.**getcwd**()

utils.shutil2.**rename**(*src*, *dst*)

utils.shutil2.**join**(*path*, **paths*)

utils.shutil2.**isdir**(*path*)

utils.shutil2.**isfile**(*path*)

utils.shutil2.**islink**(*path*)

utils.shutil2.**exists**(*path*)

utils.shutil2.**basename**(*path*)

utils.shutil2.**dirname**(*path*)

utils.shutil2.**trimext**(*path*)

utils.shutil2.**getext**(*path*)

utils.shutil2.**mkdir**(*path*)

utils.shutil2.**listdirs**(*path*, *rec=False*)

utils.shutil2.**listfiles**(*path*, *rec=False*, *ext=None*, *rel=True*)

utils.shutil2.**abspath**(*path*)

utils.shutil2.**relpath**(*path*, *rel*)

utils.shutil2.**symlink**(*src*, *dst*, *rel=True*)

utils.shutil2.**walk**(*path*, *ffunc*, *dfunc=None*, *ext=None*, *followlinks=False*)

utils.shutil2.**remove**(*path*)

utils.shutil2.**rmtree**(*path*)

utils.shutil2.**copytree**(*src*, *dst*, *followlinks=False*)

utils.shutil2.**linktree**(*src*, *dst*)

utils.timing_util module

class utils.timing_util.**TimestampManager**

Bases: object

Service class that collects and manages timestamps and time measurements for several clients.

Clients are identified by IDs. A client can set timestamps and start or stop time measurements. Each timestamp or measurement can store a message string containing information about its purpose.

The class also provides functionality for formatted output of the collected data or computing basic statistics.

timestamp(*client_id*, *msg=''*, *set_duration=False*)

Creates and stores a new timestamp for a client with a given ID.

If no client with the provided ID exists, a new collector is created for the ID. All timestamps and measurements with this ID will thereafter be collected by that collector.

The timestamp can optionally contain a duration value, which will be computed as the time difference to the most previous timestamp. This will not work if there is no previous timestamp.

Parameters

- **client_id** (*str*) – The ID of the client.
- **msg** (*str, optional*) – The message to be stored in the timestamp. Default value is "".
- **set_duration** (*bool, optional*) – Set duration of timestamp to the time difference to the previous timestamp. Default is *False*.

tic(*client_id*, *msg=None*)

Creates a new single-use time measurement starting point for a client with the given ID.

Single-use measurement points will be stacked to allow for nested time measurements. The first point ever created for a client will persist however and behave much like a permanent measurement point without an ID.

Example

The indentation levels in this scheme mark the measurement point associations:

```
tic()
    tic()
    ...
    toc()
toc()
toc()
```

If no collector for the given *client_id* exists, a new collector will be assigned to the ID. A message string can be specified to simultaneously create a new timestamp for the client.

Parameters

- **client_id** (*str*) – The ID of the client.
- **msg** (*str, optional*) – The message text of the optional timestamp. If no message is specified, no timestamp will be created. Default value is *None*.

toc(*client_id*, *msg=''*)

Creates a new timestamp representing a measurement corresponding to the last single-use measurement point.

If this measurement point is not the very first of the client's collector, it is removed from the stack. The duration of the timestamp will be set to *None* if there is no single-use measurement point.

Parameters

- **client_id** (*str*) – The ID of the client.
- **msg** (*str, optional*) – The message text of the timestamp. Default value is "".

Raises KeyError – If no collector for the specified *client_id* exists.

clearTics(*client_id*)

Removes all single-use time measurement points from the collector of the client with the given ID.

Parameters **client_id** (*str*) – The ID of the client.

Raises KeyError – If no collector for the specified *client_id* exists

setTimerStart(*client_id*, *measurement_id=None*, *msg=None*)
Creates a permanent time measurement point for the client with the given ID.

Each time measurement point has a unique ID. If the specified *measurement_id* is already present in the client's collector, it will be replaced by a new one.

If the optional *msg* parameter is set, a timestamp will be created aswell. If no collector for the given *client_id* exists, a new collector will be assigned to the ID.

Parameters

- **client_id** (*str*) – The ID of the client.
- **measurement_id** (*str, optional*) – Unique identifier for the measurement point. Default value is *None*.
- **msg** (*str, optional*) – The message text of the optional timestamp. If no message is specified, no timestamp will be created. Default value is *None*.

measureTime(*client_id*, *measurement_id=None*, *msg=''*)
Creates a new timestamp for the client with the given *client_id* that represents a time measurement.

Parameters

- **client_id** (*str*) – The ID of the client.
- **measurement_id** (*str, optional*) – The ID of the measurement point. Default value is *None*.
- **msg** (*str, optional*) – The message text of the timestamp. Default value is *""*.

Raises KeyError – If there is no collector for the client with the given *client_id* or the collector has no permanent measurement point with the given *measurement_id*.

getTimestamps(*client_id*, *pattern=''*)
Returns a list of all timestamps belonging to the given client with messages that match the given pattern. The pattern can be left empty to include all timestamps.

Parameters

- **client_id** (*str*) – The ID of the client.
- **pattern** (*str, optional*) – Pattern to match in timestamp messages. Is not used if empty. Default value is **""**.

Returns A list of *Timestamps* with messages matching *pattern*.

Return type *list* of Timestamp

Raises

- KeyError – If ther is no collector for the client with the given
- *client_id*.

class utils.timing_util.**TimestampCollector**(*client_id*)
Bases: object

Container class for timestamps of a single client.

Parameters **client_id** (*str*) – ID of the client associated to this *TimestampCollector*.

timestamp(*t*, *msg*, *set_duration*)
Creates and collects a new timestamp.

Parameters

- **t** (*float*) – The time for which to create the timestamp.
- **msg** (*str*) – The message to store in the timestamp.
- **set_duration** (*bool*) – Try to set duration to time difference to previous timestamp if *True*, otherwise set duration to *None*.

tic(*t*)
Pushes a new time measurement point on the tic stack.

Parameters **t** (*float*) – The time of the measurement point.

toc(*t*, *msg*)
Creates a new time measurement timestamp using the tic stack as measurement point provider.

The top measurement point of the stack will be used as reference point and deleted if it is not the only element on the stack.

Parameters

- **t** (*float*) – The second time of the time measurement.
- **msg** (*str*) – The message to store in the timestamp.

clearTics()
Resets the tic stack.

setTimerStart(*t*, *client_id*)
Sets the permanent time measurement point with the given *client_id* to the given time *t*.

measureTime(*t*, *client_id*, *msg*)
Creates a new timestamp representing a time measurement starting at a permanent measurement point.

The duration attribute of the timestamp is computed as the difference between the current time *t* and the time of the measurement point with the specified *client_id*.

Parameters

- **t** (*float*) – The current time.
- **client_id** (*str*) – The ID of the time measurement point.
- **msg** (*str*) – The message to store in the timestamp.

Raises KeyError – If there is no permanent time measurement point with the specified *client_id*.

timestamps

utils.verilog_utils module

utils.verilog_utils.**separateModules**(*input_design*, *output_dir*, *f_files_in_dir=False*, *modules=None*)
Extracts modules from a Verilog file and writes each in its own file.

The design file may contain annotations as specified by the *AnnotatedCandidates* FrontEnd implementation.

Parameters

- **input_design** (*str*) – The absolute path to the input circuit file in Verilog format.
- **output_dir** (*str*) – The absolute path to the directory in which to store the module circuit files.

- **f_files_in_dir** (*bool, optional*) – Specifies whether each output file is placed in a folder with the module's name (`True`) or directly in the output directory (`False`). Default value is `False`.
- **modules** (*list* of str, optional) – A list specifying the names of the modules to extract. If the list is empty or `None`, all modules are extracted. Otherwise, only the modules on the list are extracted. Default value is `None`.

utils.verilog_utils.**addModules**(*input_design*, *output_design*, *modules*)

Adds modules to a Verilog file. The file may contain annotations as specified by the *AnnotatedCandidates* FrontEnd implementation.

Parameters

- **input_design** (*str*) – The absolute path to the input circuit file in Verilog format.
- **output_design** (*str*) – The absolute path to the output circuit file.
- **modules** (*list* of str) – A list specifying the modules to add by specifying the absolute path to the circuit file which should be appended to the input circuit as a new module.

utils.verilog_utils.**removeModules**(*input_design*, *output_design*, *modules*)

Removes modules from a Verilog file. The file may contain annotations as specified by the *AnnotatedCandidates* FrontEnd implementation.

Parameters

- **input_design** (*str*) – The absolute path to the input circuit file in Verilog format.
- **output_design** (*str*) – The absolute path to the output circuit file.
- **modules** (*list* of str) – A list specifying the names of the modules to be removed.

utils.verilog_utils.**extractPisPos**(*input_design*, *module*, *cand=None*)

Extracts information about primary inputs and primary outputs from a *Candidate*'s Verilog file.

Parameters

- **input_design** (*str*) – The absolute path to the input circuit file in Verilog format.
- **module** (*str*) – The name of the module from which to extract the information.
- **cand** (`Candidate`, optional) – A *Candidate* instance can be specified to write the PI/PO information directly into the *Candidate*'s *pis* and *pos* attributes.

Returns A list containing two lists of *Signals*, one for PIs and one for POs, if *cand* is `None`. Otherwise `None`.

Return type (*list* of (*list* of `Signal`))

Raises `ValueError` – If the specified module was not found.

utils.verilog_utils.**undoBitBlasting**(*file_out*, *module*, *wrapper*, *pis*, *pos*)

Wraps a module in a Verilog circuit file in a new module. The new module has restored busses. This might come in handy since bit-blasting might have lead to split up busses/vectors, e.g., when file has been converted to BLIF and back to Verilog.

Parameters

- **file_out** (*str*) – The absolute path to the output circuit file in Verilog format.
- **module** (*str*) – The name of the module to wrap.
- **wrapper** (*str*) – The name of the new wrapper module.
- **pis** (*list* of `Signal`) – A list containing the primary input *Signals* of the wrapped module.

- **pos** (*list* of `Signal`) – A list containing the primary output *Signals* of the wrapped module.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Contact

Please, find the contact information on our website or open an issue on GitLab.

Python Module Index

Index

Symbols

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W