Edalize Documentation

Release 0.1.3

Olof Kindgren

Jul 29, 2022

REFERENCE

Edalize is a Python Library for interacting with EDA tools. It can create project files for supported tools and run them in batch or GUI mode (where supported).

CHAPTER

ONE

EDA METADATA

The EDAM (EDA Metadata) API is a data structure with the intention to describe all input parameters that an EDA tool will need to run synthesis or build a simulation model from a set of HDL files. The data described in EDAM is tool-agnostic, with the option to supply tool-specific parameters when required. The data structure itself is a tree structure with lists and dictionaries using simple data types as strings and integers for the actual values, and is suitable for (de)serialization with YAML or JSON.

Most keys are optional. The ones which are required are marked accordingly

Field Name	Type	Description
files	List of *File*	Contains all the HDL source files, constraint files, vendor IP description files, memory initialization files etc. for the project.
hooks	*Hook*	A dictionary of extra commands to execute at various stages of the project build/run.
name	String	**Required** Name of the project
parameters	Dict of *Parameter*	Specifies build- and run-time parameters, such as plusargs, VHDL generics, Verilog defines etc.
tool_options	*Tool Options*	A dictionary of tool-specific options.
toplevel	List of String	Toplevel module(s) for the project.
vpi	List of *VPI*	VPI modules to build for the project.

1.1 File

A file has a name, which is the absolute path or the relative path to the working directory. It also has a type, which describes the intended usage of the file. Different EDA tools handle different subsets of files and are expected to ignore files that are not applicable to them, but might issue a warning. By specifying *user* as the file type, the backends will explicitly ignore the file. The valid file types are based on the IP-XACT 2014 standard, with some additional file types added. The file types not covered by IP-XACT are listed below

- QIP : Intel Quartus IP file
- UCF : Xilinx ISE constraint file
- verilogSource-2005 : Verilog 200 source
- vhdlSource-2008 : VHDL 2008 source
- xci : Xilinx Vivado IP file
- xdc : Xilinx Vivado constraint file

Field Name	Type	Description
name	String	**Required** File name with (absolute or relative) path
file_type	String	**Required** File type
is_include_file	Bool	Indicates if this file should be treated as an include file (default false)
include_path	String	When is_include_file is true, the directory containing the file will be added to the include path. include_path allows setting an explicit directory to use instead
logical_name	String	Logical name (e.g. VHDL/SystemVerilog library) of the file

1.2 Hook

Hooks are scripts that can be registered to execute during various phases of Edalize. The Hook structure contains a key for each of the four defined stages, and the value of each key is a list of *Script* to be executed.

The four defined stages are

Key	Description
pre_build	Executed before calling *build*
post_build	Executed after calling *build*
pre_run	Executed before calling *run*
post_run	Executed after calling *run*

1.2.1 Script

Field Name	Type	Description
cmd	List of String	Command to execute
env	Dict of String	Additional environment variables to set before launching script
name	String	User-friendly name of the script

1.3 Parameter

A parameter is used for build- and run-time parameters, such as Verilog plusargs, VHDL generics, Verilog defines, Verilog parameters or any extra command-line options that should be sent to the simulation model. Different tools support different subsets of parameters. The list below describes valid parameter types

- cmdlinearg : Command-line argument to be sent to a running simulation model
- generic : VHDL generic to be set at elaboration-time
- plusarg : Verilog plusarg to be set at run-time
- vlogdefine : Verilog define to be set at compile-time
- vlogparam : Verilog toplevel parameter to be set at compile-time

Field Name	Type	Description
datatype	String	**Required** Data type of the parameter. Valid values are *bool*, *file*, *int*, *str*. *file* is similar to *str*, but the value is treated as a path and converted to an absolute path
default	Specified by datatype	Default value to use if user does not provide a value during the configure or run stages
description	String	User-friendly description of the parameter
paramtype	String	**Required** Type of parameter. Valid values are *cmdlinearg*, *generic*, *plusarg*, *vlogparam*, *vlogdefine*

1.4 Tool options

Tool options are used to set tool-specific options. Each key corresponds to a specific EDA tool.

Field Name	Type	Description
ghdl	String	Options for *GHDL*
icarus	String	Options for *Icarus* Verilog
icestorm	String	Options for Project *IceStorm*
ise	String	Options for Xilinx *ISE*
isim	String	Options for Xilinx *iSim*
modelsim	String	Options for Mentor *ModelSim*
openfpga	String	Options for OpenFPGA *OpenFPGA*
quartus	String	Options for Intel *Quartus*
rivierapro	String	Options for Aldec *RivieraPro*
spyglass	String	Options for Synposys *SpyGlass*
trellis	String	Options for Project *Trellis*
vcs	String	Options for Synopsys *VCS*
verilator	String	Options for *Verilator*
vivado	String	Options for Xilinx *Vivado*
vunit	String	Options for *VUnit*
xcelium	String	Options for Cadence *Xcelium*
xsim	String	Options for Xilinx *XSim*

1.4.1 ghdl

Field Name	Type	Description
analyze_options	List of String	Extra options used for the GHDL analyze stage (*ghdl -a*)
run_options	List of String	Extra options used when running GHDL simulations (*ghdl -r*)

1.4.2 icarus

Field Name	Type	Description
iverilog_options	List of String	Extra options for compilation with *iverilog*
timescale	String	Default (Verilog) timescale to use before user sets one explicitly

1.4.3 icestorm

Field Name	Type	Description
arachne_pnr_options	List of String	Options for ArachnePNR Place & Route
nextpnr_options	List of String	Options for NextPNR Place & Route
pnr	String	Select P&R tool. Valid values are *arachne* and *next*. Default is *arachne*
yosys_synth_options	List of String	Options for Yosys Synthesis

1.4.4 ise

Field Name	Type	Description
family	String	FPGA family e.g. *spartan6*, *virtex5*
device	String	Device identifier e.g. *xc6slx45*
package	String	Device package e.g. *csg324*
speed	String	Device speed grade e.g. *-2*
board_device_index	String	Specifies the FPGA's device number in the JTAG chain, starting at 1.

1.4.5 isim

Field Name	Type	Description
fuse_options	List of String	Extra options for compilation with *fuse*
isim_options	List of String	Extra options for running compiled simulation model

1.4.6 modelsim

Field Name	Type	Description
vlog_options	List of String	Extra options for each Verilog file compiled with *vlog*
vsim_options	List of String	Extra options for running the simulation with *vsim*

1.4.7 openfpga

The following environment variables need to be sourced before running any simulation on SOFA (**S**kywater **O**pen-source **F**PG**A**) IPs:

- `OPENFPGA_PATH`: directory of the OpenFPGA framework Github repo (documentation)
- `SOFA_PATH`: directory of the SOFA eFPGA IPs Github repo

Field Name	Type	Description
arch	String	FPGA architecture e.g. *sofa-hd*, *sofa-chd*, *sofa-qlhd* and *sofa-plus-hd*
task_options	List of String	Extra options for running the task simulation with OpenFPGA framework (see the OpenFPGA documentation)

1.4.8 quartus

Field Name	Type	Description
board_device_index	List of String	Specifies the FPGA's device number in the JTAG chain. The device index specifies the device where the flash programmer looks for the Nios® II JTAG debug module. JTAG devices are numbered relative to the JTAG chain, starting at 1. Use the tool *jtagconfig* to determine the index.
family	String	FPGA family e.g. *Cyclone IV E*
device	String	Device identifier. e.g. *EP4CE55F23C8* or *5CSXFC6D6F31C8ES*
quartus_options	List of String	Extra command-line options for Quartus
dse_options	List of String	Command-line options for Design Space Explorer

1.4.9 rivierapro

Field Name	Type	Description
vlog_options	List of String	Extra options for each Verilog file compiled with *vlog*
vsim_options	List of String	Extra options for running the simulation with *vsim*

1.4.10 spyglass

Field Name	Type	Default	Description
methodology	String	`GuideWare/latest/block/rtl_handoff`	Selected methodology
goals	List of String	`[ 'lint/lint_rtl' ]`	Selected goals
rule_parameters	List of String	`[]`	Options passed with `set_option` to Spyglass, e.g. "handlememory yes" to prevent error SYNTH_5273 on generic RAM descriptions
spyglass_parameters	List of String	`[]`	Rule parameters passed with `set_parameter` to Spyglass, e.g. `handle_static_caselabels yes` to allow localparam to be used in case labels (e.g. in state machines)

1.4.11 trellis

Field Name	Type	Description
nextpnr_options	List of String	Options for NextPNR Place & Route
yosys_synth_options	List of String	Options for Yosys Synthesis

1.4.12 vcs

Field Name	Type	Description
vcs_options	List of String	Compile time options passed to `vcs`
run_options	List of String	Runtime options passed to the simulation

1.4.13 verilator

Field Name	Type	Description
cli_parser	String	If *cli_parser* is set to managed, Edalize will parse all command-line options. Otherwise, they are sent directly to the compiled simulation model.
libs	List of String	Extra options to be passed as -LDFLAGS when linking the C++ testbench
mode	String	*cc* runs Verilator in regular C++ mode. *sc* runs in SystemC mode. *lint-only* only performs linting on the Verilog code
verilator_options	List of String	Extra options to be passed when verilating model

1.4.14 vivado

Field Name	Type	Description
part	String	Device identifier. e.g. *xc7a35tcsg324-1*
jobs	Integer	Number of jobs. Useful for parallelizing OOC (Out Of Context) syntheses.

1.4.15 vunit

Field Name	Type	Description
vunit_options	List of String	Extra options for the VUnit test runner
add_libraries	List of String	A list of framework libraries to add. Allowed values include "array_util", "com", "json4hdl", "osvvm", "random", "verification_components"
vunit_runner	String	Name of the Python file exporting a `VUnitRunner` class (must derive from `edalize.vunit_hooks.VUnitHooks`) that is used to configure and execute test. This allows very customized test control via VUnit's Python-interfaces.

In case a more advanced VUnit configuration or execution of the testbench is necessary, the option `vunit_runner` can be used to specify the filename of a Python script which can hook into the construction, parametrization, and execution of the test runner. For this to work, the Python script must export a `class VUnitRunner(vunit_hooks.VUnitHooks)` which derives from (and optionally overrides) the behavior of `vunit_hooks.VUnitHooks`.

```
from edalize.vunit_hooks import VUnitHooks
from vunit import VUnit
from vunit.ui import Library, Results
from typing import Mapping, Collection


class VUnitRunner(VUnitHooks):
    """Example of custom VUnit instrumentation."""

    def create(self) -> VUnit:
        """Customized creation of the test runner"""
        vu = VUnit.from_argv()
        vu.enable_check_preprocessing()
        return vu

    def handle_library(self, logical_name: str, vu_lib: Library):
        """Override this to customize each library, e.g. with additional simulator␣
↪options.
        This hook will be invoked for each library, after all source files have been␣
↪added.
        :param logical_name: The logical name of the library
        :param vu_lib: The vunit.ui.Library instance, configured with all sources of␣
↪this `logical_name`
        """
        # e.g. you can access and customize test-bench entities of this library:
```

(continues on next page)

(continued from previous page)

```
        if logical_name == "my_tb_library_name":
            entity = vu_lib.entity("my_toplevel_tb")
            entity.set_generic("message", "Test message")
            entity.add_config(name="TestConfig1",
                              generics=dict(CLK_FREQ=10000000))
            entity.add_config(name="TestConfig2",
                              generics=dict(CLK_FREQ=54687500))

    def main(self, vu: VUnit):
        """Override this for final parametrization of the :class:`VUnit` instance (after
↪all libraries have been added),
        or for custom invocation of VUnit
        """
        def post_run_handler(results: Results):
            results.merge_coverage(file_name="coverage_data")

        vu.main(post_run=post_run_handler)
```

1.4.16 xcelium

Field Name	Type	Description
xmvlog_options	List of String	Extra options for compilation with *xmvlog*
xmvhdl_options	List of String	Extra options for compilation with *xmvhdl*
xmsim_options	List of String	Extra options for running simulation with with *xsim*
xrun_options	List of String	Extra options for invocation with with *xrun*

1.4.17 xsim

Field Name	Type	Description
xelab_options	List of String	Extra options for compilation with *xelab*
xsim_options	List of String	Extra options for running simulation with with *xsim*

1.4.18 toplevel

Name of the top level module/entity

1.5 VPI

Each *Vpi* object contains information on how to build the corresponding VPI library

Field Name	Type	Description
include_dirs	List of String	Extra include directories
libs	List of String	Extra libraries
name	String	Name of VPI library
src_files	List of String	Source files for VPI library

CHAPTER

TWO

EDALIZE PACKAGE

2.1 Submodules

2.2 edalize.edatool module

class edalize.edatool.**Edatool**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: object

build()

build_main(*target=None*)

build_post()

build_pre()

configure(*args=[]*)

configure_main()

configure_post()

configure_pre()

classmethod get_doc(*api_ver*)

parse_args(*args*, *paramtypes*)

render_template(*template_file*, *target_file*, *template_vars={}*)

Render a Jinja2 template for the backend.

The template file is expected in the directory templates/BACKEND_NAME.

run(*args={}*)

run_main()

run_post()

run_pre(*args=None*)

set_default_target(*target*)

class edalize.edatool.**FileAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: Action

edalize.edatool.**get_edatool**(*name*)

edalize.edatool.**get_edatools**()

edalize.edatool.**jinja_filter_param_value_str**(*value*, *str_quote_style=''*, *bool_is_str=False*)

Convert a parameter value to string suitable to be passed to an EDA tool.

Rules:

- Booleans are represented as 0/1 or "true"/"false" depending on the bool_is_str argument
- Strings are either passed through or enclosed in the characters specified in str_quote_style (e.g. "" or '')
- Everything else (including int, float, etc.) are converted using the str() function.

edalize.edatool.**subprocess_run_3_9**(**popenargs*, *input=None*, *capture_output=False*, *timeout=None*, *check=False*, ***kwargs*)

edalize.edatool.**walk_tool_packages**()

2.3 edalize.ghdl module

class edalize.ghdl.**Ghdl**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['vlogparam', 'generic']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.4 edalize.icarus module

class edalize.icarus.**Icarus**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['plusarg', 'vlogdefine', 'vlogparam']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.5 edalize.icestorm module

class edalize.icestorm.**Icestorm**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['vlogdefine', 'vlogparam']

build_post()

build_pre()

configure_main()

classmethod get_doc(*api_ver*)

2.6 edalize.ise module

class edalize.ise.**Ise**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

MAKEFILE_TEMPLATE = '#Auto generated by Edalize\ninclude config.mk\n\nall: $(TOPLEVEL).bit\n\n$(TOPLEVEL).bit: $(NAME)_run.tcl $(NAME).xise\n\txtclsh $^\n\n$(NAME).xise: $(NAME).tcl\n\txtclsh $<\n'

PGM_FILE_TEMPLATE = '\n# Batch script for programming the device using a JTAG interface.\n# Used with:\n# $ impact -batch {pgm_file}\n\nsetMode -bscan\nsetCable -port auto\nidentify\nassignFile -p {board_device_index} -file {bit_file}\nprogram -p {board_device_index}\nsaveCDF -file {cdf_file}\nquit\n'

TCL_FILE_TEMPLATE = '#Auto generated by Edalize\nproc project_new_exist_ok name {{\n if {{ [catch {{ project new $name }}] }} {{\n project open $name\n }}\n}}\n\nproc xfile_add_exist_ok name {{\n if {{ [catch {{ xfile get [file tail $name] name }}] }} {{\n xfile add $name\n }}\n}}\n\nproject_new_exist_ok {design}\nproject set family {family}\nproject set device {device}\nproject set package {package}\nproject set speed {speed}\nproject set "Generate Detailed MAP Report" true\n'

TCL_RUN_FILE_TEMPLATE = '#Auto generated by Edalize\nproject open $::argv\nprocess run "Generate Programming File"\n'

argtypes = ['vlogdefine', 'vlogparam', 'generic']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.7 edalize.isim module

class edalize.isim.**Isim**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

CONFIG_MK_TEMPLATE = '#Auto generated by Edalize\nTARGET = {target}\nTOPLEVEL = {toplevel}\n\nVLOG_DEFINES = {vlog_defines}\nVLOG_INCLUDES = {vlog_includes}\nVLOG_PARAMS = {vlog_params}\n\nFUSE_OPTIONS =\t{fuse_options}\nISIM_OPTIONS =\t{isim_options}\n\nEXTRA_OPTIONS ?= {extra_options}\n'

MAKEFILE_TEMPLATE = '#Auto generated by Edalize\ninclude config.mk\n\nall: $(TARGET)\n\n$(TARGET):\n\tfuse $(TOPLEVEL) -prj $(TARGET).prj -o $(TARGET) $(VLOG_DEFINES) $(VLOG_INCLUDES) $(VLOG_PARAMS) $(FUSE_OPTIONS)\n\nrun: $(TARGET)\n\t./$(TARGET) -tclbatch run_$(TARGET).tcl $(ISIM_OPTIONS) $(EXTRA_OPTIONS)\n\nrun-gui: $(TARGET)\n\t./$(TARGET) -gui $(ISIM_OPTIONS) $(EXTRA_OPTIONS)\n'

RUN_TCL_TEMPLATE = '#Auto generated by Edalize\nwave log -r /\nrun all\nquit\n'

argtypes = ['plusarg', 'vlogdefine', 'vlogparam']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.8 edalize.modelsim module

class edalize.modelsim.**Modelsim**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['plusarg', 'vlogdefine', 'vlogparam', 'generic']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.9 edalize.openfpga module

class edalize.openfpga.**Openfpga**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=False*)

Bases: *Edatool*

This calls the parent constructor, but also identifies whether the current system has correctly set the following environment variables:

- `OPENFPGA_PATH`: directory of the OpenFPGA framework, available here: https://github.com/lnis-uofu/OpenFPGA
- `SOFA_PATH`: directory of the SOFA eFPGA IPs, available here: https://github.com/lnis-uofu/SOFA

argtypes = ['plusarg', 'vlogdefine', 'vlogparam']

build_main()

configure_main()

Configuration is the first phase of the build.

This writes an OpenFPGA task file for SOFA/SOFA+ architectures, which will generate the according OpenFPGA flow. It first collects all verilog sources, top_module and then writes them into the task file.

Note: OpenFPGA flow may uses Yosys/VPR backend for Synthesis and P&R, respectively.

classmethod get_doc(*api_ver*)

run_main()

Run the FPGA simulation.

edalize.openfpga.**logger = <Logger edalize.openfpga (WARNING)>**

OpenFPGA Flow Backend.

A core (usually the system core) can add the following files:

- Benchmark RTL sources (Yosys supports only Verilog file type) and module name
- The target FPGA architecture name, made with OpenFPGA fabric flow (SOFA,...)
- Source the required environment variables: OPENFPGA_PATH, SOFA_PATH
- Optional parameters: task options ('–debug', ...)

2.10 edalize.quartus module

class edalize.quartus.**Quartus**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=False*)

Bases: *Edatool*

Initial setup of the class.

This calls the parent constructor, but also identifies whether the current system is using a Standard or Pro edition of Quartus.

argtypes = ['vlogdefine', 'vlogparam', 'generic']

build_main()

configure_main()

Configuration is the first phase of the build.

This writes the project TCL files and Makefile. It first collects all sources, IPs and constraints and then writes them to the TCL file along with the build steps.

file_type(*f*)

classmethod get_doc(*api_ver*)

isPro = False

makefile_template = {False: 'quartus-std-makefile.j2', True: 'quartus-pro-makefile.j2'}

qsys_file_filter(*f*)

run_main()

Program the FPGA.

src_file_filter(*f*)

2.11 edalize.rivierapro module

class edalize.rivierapro.**Rivierapro**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['plusarg', 'vlogdefine', 'vlogparam']

build_main()

build_pre()

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.12 edalize.symbiyosys module

class edalize.symbiyosys.**Symbiyosys**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['vlogdefine', 'vlogparam']

build_main()

configure_main()

static get_doc(*api_ver*)

run_main()

tool_options = {'lists': {'tasknames': 'String'}}

2.13 edalize.spyglass module

class edalize.spyglass.**Spyglass**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['vlogdefine', 'vlogparam']

configure_main()

Configuration is the first phase of the build.

This writes the project TCL files and Makefile. It first collects all sources, IPs and constraints and then writes them to the TCL file along with the build steps.

src_file_filter(*f*)

tool_options = {'lists': {'goals': 'String', 'rule_parameters': 'String', 'spyglass_options': 'String'}, 'members': {'methodology': 'String'}}

tool_options_defaults = {'goals': ['lint/lint_rtl'], 'methodology': 'GuideWare/latest/block/rtl_handoff', 'rule_parameters': [], 'spyglass_options': []}

2.14 edalize.trellis module

class edalize.trellis.**Trellis**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['vlogdefine', 'vlogparam']

configure_main()

classmethod get_doc(*api_ver*)

2.15 edalize.vcs module

class edalize.vcs.**Vcs**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['plusarg', 'vlogdefine', 'vlogparam']

configure_main()

run_main()

tool_options = {'lists': {'run_options': 'String', 'vcs_options': 'String'}}

2.16 edalize.verilator module

class edalize.verilator.**Verilator**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['cmdlinearg', 'plusarg', 'vlogdefine', 'vlogparam']

build_main()

check_managed_parser()

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.17 edalize.vivado module

class edalize.vivado.**Vivado**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

Vivado Backend.

A core (usually the system core) can add the following files:

- Standard design sources
- Constraints: Supply xdc files with file_type=xdc or unmanaged constraints with file_type SDC
- IP: Supply the IP core xci file with file_type=xci and other files (like .prj) as file_type=user

argtypes = ['vlogdefine', 'vlogparam', 'generic']

build_post()

build_pre()

configure_main()

classmethod get_doc(*api_ver*)

run_main()

Program the FPGA.

For programming the FPGA a vivado tcl script is written that searches for the correct FPGA board and then downloads the bitstream. The tcl script is then executed in Vivado's batch mode.

2.18 edalize.vunit module

class edalize.vunit.**Vunit**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['cmdlinearg']

build_main()

configure_main()

classmethod get_doc(*api_ver*)

get_vunit_runner_path(*src_files*)

run_main()

src_file_filter(*f*)

src_file_vhdl_standard_filter(*f*)

testrunner = 'run.py'

testrunner_template = 'run.py.j2'

2.19 edalize.vunit_hooks module

This module exports *VUnitHooks* which can be used to implement advanced VUnit test cases.

class edalize.vunit_hooks.**VUnitHooks**

Bases: object

Derive the *VUnitRunner* instance from this class and override its member functions if necessary.

create()

Override this function to specify custom instantiation of VUnit.

Return type
VUnit

handle_library(*logical_name*, *vu_lib*)

Override this to customize each library, e.g. with additional simulator options.

main(*vu*)

Override this for final parametrization of the VUnit instance, or for custom invocation of VUnit.

class edalize.vunit_hooks.**VUnitRunner**

Bases: *VUnitHooks*

The default runner which will be used if no `vunit_runner.py` is specified.

2.20 edalize.xcelium module

class edalize.xcelium.**Xcelium**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

argtypes = ['plusarg', 'vlogdefine', 'vlogparam', 'generic']

configure_main()

classmethod get_doc(*api_ver*)

run_main()

2.21 edalize.xsim module

class edalize.xsim.**Xsim**(*edam=None*, *work_root=None*, *eda_api=None*, *verbose=True*)

Bases: *Edatool*

CONFIG_MK_TEMPLATE = '#Auto generated by Edalize\nTARGET = {target}\nTOPLEVEL = {toplevel}\n\nVLOG_DEFINES = {vlog_defines}\nVLOG_INCLUDES = {vlog_includes}\nGEN_PARAMS = {gen_params}\n\nXELAB_OPTIONS =\t{xelab_options}\nXSIM_OPTIONS = {xsim_options}\n'

```
MAKEFILE_TEMPLATE = '#Auto generated by Edalize\ninclude config.mk\n\nall:
xsim.dir/$(TARGET)/xsimk\n\nxsim.dir/$(TARGET)/xsimk:\n\txelab $(TOPLEVEL) -prj
$(TARGET).prj -snapshot $(TARGET) $(VLOG_DEFINES) $(VLOG_INCLUDES) $(GEN_PARAMS)
$(XELAB_OPTIONS)\n\nrun:  xsim.dir/$(TARGET)/xsimk\n\txsim -R $(XSIM_OPTIONS)
$(TARGET) $(EXTRA_OPTIONS)\n\nrun-gui:  xsim.dir/$(TARGET)/xsimk\n\txsim --gui
$(XSIM_OPTIONS) $(TARGET) $(EXTRA_OPTIONS)\n'

argtypes = ['plusarg', 'vlogdefine', 'vlogparam', 'generic']

configure_main()

classmethod get_doc(api_ver)

run_main()
```

2.22 edalize.reporting module

2.23 edalize.vivado_reporting module

2.24 edalize.quartus_reporting module

2.25 edalize.ise_reporting module

2.26 Module contents

edalize.**get_edatool**(*name*)

edalize.**get_edatools**()

edalize.**get_flow**(*name*)

edalize.**walk_tool_packages**()

CHAPTER

THREE

DEVELOPMENT SETUP

3.1 Setup development environment

Note: If you have already installed Edalize, remove it first using `pip3 uninstall edalize`.

To develop Edalize and test the changes, the edalize package needs to be installed in editable or development mode. In this mode, the `edalize` command is linked to the source directory, and changes made to the source code are immediately visible when calling `edalize`.

```
# Install all Python packages required to develop edalize
pip3 install --user -r dev-requirements.txt

# Install Git pre-commit hooks, e.g. for the code formatter and lint tools
pre-commit install

# Install the edalize package in editable mode
pip3 install --user -e .
```

Note: All commands above use Python 3 and install software only for the current user. If, after this installation, the `edalize` command cannot be found adjust your `PATH` environment variable to include `~/.local/bin`.

After this installation is completed, you can

- edit files in the source directory and re-run `edalize` to immediately see the changes,
- run the unit tests as outlined in the section below, and
- use linter and automated code formatters.

3.2 Formatting and linting code

The Edalize code comes with tooling to automatically format code to conform to our expectations. These tools are installed and called through a tool called pre-commit. No setup is required: whenever you do a `git commit`, the necessary tools are called and your code is automatically formatted and checked for common mistakes.

To check the whole source code `pre-commit` can be run directly:

```
# check and fix all files
pre-commit run -a
```

CHAPTER

FOUR

TESTING EDALIZE

To run the tests, call **`pytest`**.

4.1 Mocks for commands

We provide mocks (stand-ins) for all tools that we want to exercise in tests (located in `tests/mock_commands/`). These mocks are very simplified "models" of the actual tool, and are called instead of the actual tool. They are prepended to `PATH` by the setup code in the `edalize_common.make_edalize_test()` pytest fixture.

In the easiest case, these mocks just write out the commandline they were called with (into a file `<tool>.cmd`).

In a more complex test setup (e.g. for `vcs`),

- if a tool is creating files, we create it too (the file to create is taken from a tool option given)
- we make the file executable
- we set the access and modified times of generated files to the current time

4.2 Testcases

To define a testcase, use the `edalize_common.make_edalize_test()` pytest factory fixture. This defines a factory that you can call to set up a mocked-up backend appropriately. See the documentation for `edalize_common.TestFixture` for details of the supported keywords.

The `backend` attribute of the returned fixture has `configure()`, `build()` and `run()` methods. The testcase should call these in order, setting up files as necessary between calls and checking whether the results match by calling the fixture's `compare_files()` method.

If the environment variable `GOLDEN_RUN` is set, the `compare_files()` method copies the generated files are copied to become the new reference files, rather than checking their contents.

4.3 Helper Module

4.3.1 edalize_common

CHAPTER

FIVE

INDEX

CHAPTER

SIX

SEARCH PAGE

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

J

L

M

O

P

Q

R

S

T

V

W

X