Netlist Paths

James Hanlon

Jun 13, 2022

CONTENTS:

Netlist Paths is a library and command-line tool for querying a Verilog netlist. It reads an XML representation of a design's netlist, produced by Verilator, and provides facilities for inspecting variables and their data type and dependency information. The library is written in C++ and has a Python interface allowing it to be integrated easily into scripts.

Compared with standard front-end EDA tools such as Synopsys Verdi and Spyglass, Netlist Paths is oriented towards command-line use for exploration of a design (rather than with a GUI), and for integration with Python infrastructure (rather than TCL) to build tools for analysing or debugging a design. By focusing on source-level connectivity it is lightweight and will run faster than standard tools to perform a comparable task, whilst also being open source and unrestricted by licensing issues. Applications include critical timing path investigation, creation of unit tests for design structure and connectivity, and development of patterns for quality-of-result reporting.

CHAPTER

ONE

INSTALLATION

The following dependencies must be installed:

- C++ compiler supporting C++17
- CMake (minimum 3.12.0)
- Boost (minimum 1.65.0)
- Python (minimum 3.8)
- Make
- Autoconf
- Flex
- Bison
- Doxygen (only required if building the documentation)

In an Ubuntu 20.04 environment for instance, these can be satisfied with:

```
apt-get update && apt-get install -yq build-essential git flex bison libboost-all-dev␣
↪libfl-dev cmake
```

To build and install netlist paths, configure the build system with CMake, then run Make. Note that Verilator is included as a submodule and as part of the build.

```
git clone https://github.com/jameshanlon/netlist-paths.git
cd netlist-paths
git submodule update --init --recursive
...
mkdir Release
cd Release
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=./install
...
make -j8 install
...
```

Optionally, run the unit tests:

```
ctest --verbose
...
```

To build the documentation add `-DNETLIST_PATHS_BUILD_DOCS=1` to the `cmake` command, and before running the `cmake` step, install the dependencies in a virtualenv:

```
 cd Release
 virtualenv -p python3 env
...
 source env/bin/activate
 pip install -r ../docs/requirements.txt
```

Once the build and install steps have completed, set `PATH` and `PYTHONPATH` appropriately to the `bin` and `lib` directories of the installation to make the command-line tools and Python modules accessible.

CHAPTER

TWO

USER GUIDE

2.1 Command-line tool

The easiest way to use Netlist Paths is with the `netlist-paths` command line tool, which provides ways to access the library's functionalities. For example (assuming `PATH` includes your installation directory), to compile and view a list of named objects in the FSM example module:

```
 netlist-paths --compile examples/fsm.sv --dump-names
Name           Type DType        Width Direction Location
-------------- ---- ------------ ----- --------- ------------------
i_clk          PORT logic        1     INPUT     examples/fsm.sv:3
i_finish       PORT logic        1     INPUT     examples/fsm.sv:6
i_rst          PORT logic        1     INPUT     examples/fsm.sv:4
i_start        PORT logic        1     INPUT     examples/fsm.sv:5
i_wait         PORT logic        1     INPUT     examples/fsm.sv:7
o_state        PORT [2:0] logic  3     OUTPUT    examples/fsm.sv:8
fsm.i_clk      VAR  logic        1     INPUT     examples/fsm.sv:3
fsm.i_finish   VAR  logic        1     INPUT     examples/fsm.sv:6
fsm.i_rst      VAR  logic        1     INPUT     examples/fsm.sv:4
fsm.i_start    VAR  logic        1     INPUT     examples/fsm.sv:5
fsm.i_wait     VAR  logic        1     INPUT     examples/fsm.sv:7
fsm.next_state VAR  packed union 3     NONE      examples/fsm.sv:31
fsm.o_state    VAR  [2:0] logic  3     OUTPUT    examples/fsm.sv:8
fsm.state_q    REG  packed union 3     NONE      examples/fsm.sv:28
```

This output lists each of the variables in the design, their type (variable or register), the Verilog data type, with data type width, the direction of the variable (ports only) and the source location. This list can be filtered by also supplying a pattern (with the `--regex` flag to enable regular expression matching):

```
 netlist-paths --compile examples/fsm.sv --dump-names state --regex
Name           Type DType        Width Direction Location
-------------- ---- ------------ ----- --------- ------------------
o_state        PORT [2:0] logic  3     OUTPUT    examples/fsm.sv:8
fsm.next_state VAR  packed union 3     NONE      examples/fsm.sv:31
fsm.o_state    VAR  [2:0] logic  3     OUTPUT    examples/fsm.sv:8
fsm.state_q    REG  packed union 3     NONE      examples/fsm.sv:28
```

There is similar behaviour with `--dump-nets`, `--dump-ports`, `--dump-regs` to select only net, port or register variable types respectively. The argument `--dump-dtypes` reports the named data types used in the design:

```
 netlist-paths --compile examples/fsm.sv --dump-dtypes fsm --regex
Name             Width Description
---------------- ----- ------------
fsm.state_enum_t 3     enum
fsm.state_t      3     packed union
```

Note that the `--compile` argument causes Verilator to be run to create the XML netlist, and is useful for compiling simple examples. Execution of Verilator can be seen with verbose output:

```
 netlist-paths --compile examples/fsm.sv --verbose
info: Running ".../bin/np-verilator_bin" +1800-2012ext+.sv --bbox-sys --bbox-unsup
  --xml-only --flatten --error-limit 10000 --xml-output l9pvjzq6 examples/fsm.sv
...
```

The `-I` and `-D` arguments can be used with `--compile` to add include directories and macro definitions respectively for Verilator, but for more complex invocations, Verilator can just be run separately and the path to the XML output provided to `netlist-paths` as an argument.

2.2 Command-line tool reference

```
usage: netlist-paths [-h] [-c] [-I include_path] [-D definition] [-o file] [--dump-names␣
↪[pattern]]
                     [--dump-nets [pattern]] [--dump-ports [pattern]] [--dump-regs␣
↪[pattern]]
                     [--dump-dtypes [pattern]] [--dump-dot] [--from point] [--to point]␣
↪[--through point]
                     [--avoid point] [--traverse-registers] [--start-anywhere] [--end-
↪anywhere] [--all-paths]
                     [--regex] [--wildcard] [--ignore-hierarchy-markers] [-v] [-d]
                     files [files ...]

Query a Verilog netlist

positional arguments:
  files                       Input files

options:
  -h, --help                  Show this help message and exit
  --version                   Display the version number and exit
  -c, --compile               Run Verilator to compile a netlist
  -I include_path             Add an source include path (only with --compile)
  -D definition               Define a preprocessor macro (only with --compile)
  -o file,--output file       Specify an output file
  --dump-names [pattern]      Dump all named entities, filter by regex
  --dump-nets [pattern]       Dump all nets, filter by regex
  --dump-ports [pattern]      Dump all ports, filter by regex
  --dump-regs [pattern]       Dump all registers, filter by regex
  --dump-dtypes [pattern]     Dump all data types, filter by regex
  --dump-dot                  Dump a dotfile of the netlist's graph
  --from point                Specify a path start point
  --to point                  Specify a path finish point
```

(continues on next page)

(continued from previous page)

```
  --through point             Specify a path though point
  --avoid point               Specify a point for a path to avoid
  --traverse-registers        Allow paths to traverse registers
  --start-anywhere            Allow paths to start on any variable
  --end-anywhere              Allow paths to end on any variable
  --all-paths                 Find all paths between two points (exponential time)
  --regex                     Enable regular expression matching of names
  --wildcard                  Enable wildcard matching of names
  --ignore-hierarchy-markers  Ignore hierarchy markers: _ . /
  -v, --verbose               Print execution information
  -d, --debug                 Print debugging information
```

2.3 Python module

The functionality of Netlist Paths can be accessed in Python, using the `py_netlist_paths` module, which is is a wrapper around the C++ library. Full details of the API can be found in *Python API reference*, and for the C++ library in *C++ API reference*. As a simple example, the Python Netlist Paths library can be used interactively, first by creating an XML netlist file (and making sure the library install directory is a search path for Python modules:

```
netlist-paths --compile examples/fsm.sv --output fsm.xml
export PYTHONPATH=/path/to/netlist-paths/install/lib:$PYTHONPATH
```

A new `Netlist` object can then be created, specifying the XML file as an argument, and the list of named objects can be retrieved as above using the `get_named_vertices()` method:

```
 python3
Python 3.9.0 (default, Dec  6 2020, 18:02:34)
[Clang 12.0.0 (clang-1200.0.32.27)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from py_netlist_paths import Netlist
>>> netlist = Netlist('fsm.xml')
>>> for v in netlist.get_reg_vertices():
...     print(v.get_name())
...
fsm.state_q
```

This can then easily be turned into a script to create a tool that reports registers in a design:

```
import argparse
import sys
from py_netlist_paths import Netlist

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('netlist_file')
    args = parser.parse_args()
    netlist = np.Netlist(args.netlist_file)
    for register in netlist.get_reg_vertices():
        print('{} {}'.format(register.get_name(), register.get_dtype_str()))
```

This example tool can be run from the examples directory:

```
 netlist-paths --compile examples/fsm.sv --output fsm.xml
 python3 -m examples.list_registers -h
usage: list_registers.py [-h] netlist_file

positional arguments:
  netlist_file

optional arguments:
  -h, --help    show this help message and exit
 python3 -m examples.list_registers fsm.xml
fsm.state_q packed union
```

2.4 Running Netlist Paths on a design

If it is possible to elaborate a design with Verilator, then Netlist Paths can be used to analyse the XML netlist of that design. Verilator produces the netlist with the arguments `--xml-only` (causing it to only create XML output) and `--flatten` (causing it to force inlining of all modules, tasks and functions, fully elaborating the netlist). Examples of complex, real-world designs can be found in the netlist-paths-tests repository.

2.5 Limitations

Netlist Paths has some limitations that should be considered when using it:

- It does not perform elaboration of the design at the bit level, meaning that dependencies are between named variables, rather than components of variables, such as subscripts, slices, or fields. As such, the dependencies it does infer are at a coarse level of granularity. Full bit-level elaboration (or bit blasting) will substantially impact the size of the netlist graph, and the corresponding runtimes, but it is something that may be supported in the future.
- During elaboration of the design, Verilator introduces new entities that do not appear in the original design source, with names beginning or containing `__V`. This is a artefact of Verilator preparing the syntax tree for emission as C++ code for simulation, rather than as a true source-level netlist. Future releases of Verilator may improve this mapping.

CHAPTER

THREE

CONTRIBUTING

Contributions are welcome, check the GitHub issues page for work to do, and please follow the LLVM coding standards.

3.1 Developer notes

The command-line flags `--verbose` and `--debug` provide logging information that can aid debugging.

To produce XML from a test case (noting the underlying call to Verilator), using an adder as an example:

```
 cat tests/verilog/adder.sv
module adder
  #(parameter p_width = 32)(
    input  logic [p_width-1:0] i_a,
    input  logic [p_width-1:0] i_b,
    output logic [p_width-1:0] o_sum,
    output logic               o_co
  );
  assign {o_co, o_sum} = i_a + i_b;
endmodule
 netlist-paths --compile tests/verilog/adder.sv -o adder.xml --verbose
info: Running ".../netlist-paths/Debug/install/bin/np-verilator_bin" +1800-2012ext+.sv
   --bbox-sys --bbox-unsup --xml-only --flatten --error-limit 10000 --xml-output adder.
↪xml tests/verilog/adder.sv
info: Parsing input XML file
info: 1 modules in netlist
info: 0 interfaces in netlist
info: 0 packages in netlist
info: 4 entries in type table
info: Netlist contains 15 vertices and 22 edges
```

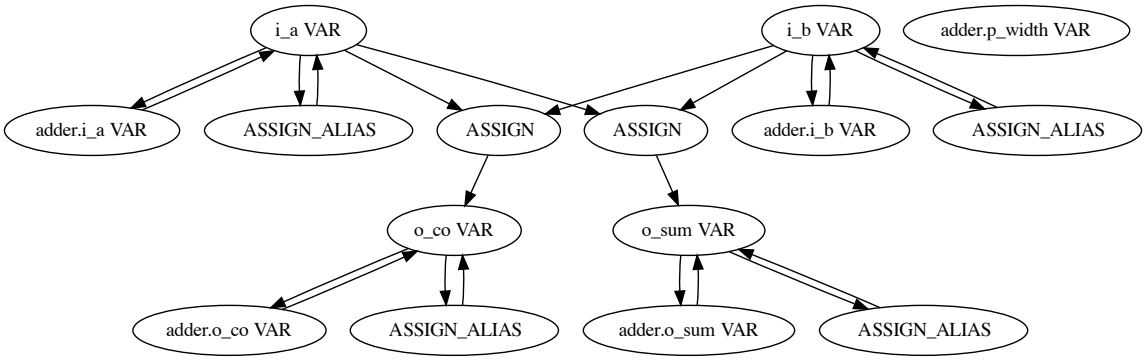
To produce a visualisation of the netlist graph, a dot file can be produced and rendered into an image. This can be useful to understand the structure of the graph, although it is only practical to use with small designs.

```
 netlist-paths adder.xml --dump-dot -o adder.dot
 dot -Tpdf adder.dot -o adder.pdf
```

The tests are split into three categories: C++ unit tests for the library components, Python unit tests for the Python API and Python integration tests for the command-line tools. These can all be run automatically using *ctest*, or individually.

```
 ctest --verbose # Run all tests from the build directory.
```

For the C++ unit tests:




```
 ./tests/unit/UnitTests --help
 ./tests/unit/UnitTests --list_content # Report a list of all the tests.
 ./tests/unit/UnitTests # Run all the unit tests
...
```

For the Python API unit tests:

```
 cd Debug/tests/integration
 python3 -m unittest py_wrapper_tests.py
...
```

For the Python integration tests:

```
 cd Debug/tests/integration
 python3 -m unittest tool_tests.py
...
```

To run the extended test set, the netlist-paths-tests repository contains tests based on external System Verilog designs.

3.2 Internals

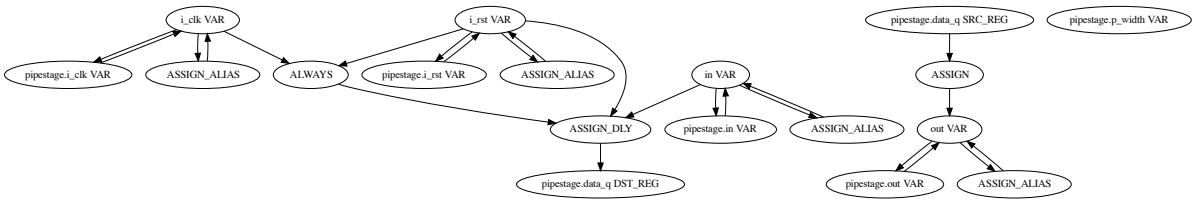
Netlist Paths works by reading an XML representation of a Verilog design produced by Verilator, which is a single flattened module with all tasks, functions, and sub-module instances inlined (Verilator is run using the `--xml-only` and `--flatten` options). The XML is traversed to build up a directed graph data structure representing the design netlist, where nodes of the graph are logic statements or variables, and edges are data dependencies between them. The data-type information of the design is also read from the XML, and a corresponding data-type table is constructed, that each variable references.

To support querying of paths that start and/or finish on registers (as is the case with physical timing reports), registers are identified during parsing of the XML so they can be split into two components: a source and a destination, with out-edges and in-edges of the original node respectively. Register variables are identified when they appear on the left-hand side of a non-blocking assignment[1]. As an example, consider the following module that provides a bank of asynchronous-reset flip flops:

[1] Note that this condition is simplistic and will be extended to check the always block is sensitive to a rising or falling edge of a clock. It is however sufficient to identify registers in most RTL code, which generally does not mix blocking and non-blocking assignments. See Issue 5.

```
module pipestage
  #(parameter p_width=32) (
    input  logic               i_clk,
    input  logic               i_rst,
    input  logic [p_width-1:0] in,
    output logic [p_width-1:0] out
  );
  logic [p_width-1:0] data_q;
  always_ff @(posedge i_clk or posedge i_rst)
    if (i_rst) begin
      data_q <= '0;
    end else begin
      data_q <= in;
    end
  assign out = data_q;
endmodule
```

The netlist graph of the `pipestage` has two distinct components: the fan in-cone to `data_q` (appearing as a `DST_REG`) including the sensitivity list of the always block and the input port, and `data_q` driving the output port (appearing as a `SRC_REG`).



Edges between register endpoints and subsequent uses of the register are retained in the graph to support queries 'through' registers, which can be useful for establishing connectivity in a pipelined design.

CHAPTER

FOUR

C++ API REFERENCE

4.1 DType

class netlist_paths::**DType**
Base class for data types.

Subclassed by netlist_paths::ArrayDType, netlist_paths::BasicDType, netlist_paths::ClassRefDType, netlist_paths::EnumDType, netlist_paths::MemberDType, netlist_paths::RefDType, netlist_paths::StructDType, netlist_paths::UnionDType, netlist_paths::VoidDType

Public Functions

inline virtual const std::string **toString**(const std::string suffix = "") const
Return the string representation of the data type.

Parameters **suffix** – A suffix to allow unpacked array range specifications to be appended with the inner-most dimension on the LHS and outermost on the RHS.

Returns A string representation of the data type.

4.2 Exception

class **Exception** : public exception
Base class for netlist_paths exceptions.

Subclassed by *netlist_paths::XMLException*

class **XMLException** : public netlist_paths::*Exception*
An exception from XML elaboration.

4.3 Netlist

class `netlist_paths::Netlist`
Wrapper for Python to manage the netlist object.

Public Functions

`Netlist`(const std::string &filename)
Construct a new netlist from an XML file.

Parameters `filename` – A path to the XML netlist file.

const std::string `getVertexDTypeStr`(const std::string &name, *VertexNetlistType* vertexType = *VertexNetlistType*::*ANY*) const
Lookup the data type string of a single vertex.

Parameters

- `name` – A pattern specifying a name to match.
- `vertexType` – The type of the vertex to lookup.

Returns A string representing the data type.

size_t `getVertexDTypeWidth`(const std::string &name, *VertexNetlistType* vertexType = *VertexNetlistType*::*ANY*) const
Lookup the data type width of a single vertex.

Parameters

- `name` – A pattern specifying a name to match.
- `vertexType` – The type of the vertex to lookup.

Returns The width of the data type.

size_t `getDTypeWidth`(const std::string &name) const
Lookup the width of a data type by name.

Parameters `name` – A pattern specifying a name to match.

Returns The width of the data type.

const std::vector<*DType**> `getNamedDTypes`(const std::string pattern = std::string()) const
Return a vector of data types in the design. Convert to raw pointers for Python.

bool `startpointExists`(const std::string &name) const
Return true if a single startpoint matching a pattern exists.

Parameters `name` – A pattern specifying a name to match.

Returns True if the startpoint exists.

bool `endpointExists`(const std::string &name) const
Return true if a single endpoint matching a pattern 'name' exists.

Parameters `name` – A pattern specifying a name to match.

Returns True if the endpoint exists.

bool `anyStartpointExists`(const std::string &name) const
Return true if any startpoint matching a pattern 'name' exists.

Parameters **name** – A pattern specifying a name to match.

Returns True if any startpoint exists.

bool **anyEndpointExists**(const std::string &name) const
Return true if any endpoint matching a pattern 'name' exists.

Parameters **name** – A pattern specifying a name to match.

Returns True if any endpoint exists.

bool **regExists**(const std::string &name) const
Return true if a single register matching a pattern 'name' exists.

Parameters **name** – A pattern specifying a name to match.

Returns True if the register exists.

bool **anyRegExists**(const std::string &name) const
Return true if any register matching a pattern 'name' exists.

Parameters **name** – A pattern specifying a name to match.

Returns True if any register exists.

bool **pathExists**(*Waypoints* waypoints) const
Return a Boolean to indicate whether any path exists between two points.

Parameters **waypoints** – A waypoints object constraining the path.

Returns True if a path exists.

Path **getAnyPath**(*Waypoints* waypoints) const
Return any path between two points.

Parameters **waypoints** – A waypoints object constraining the path.

Returns A path if one exists, otherwise an empty vector.

std::vector<*Path*> **getAllPaths**(*Waypoints* waypoints) const
Return all paths between two points, useful for testing.

Parameters **waypoints** – A waypoints object constraining the path.

Returns All paths matching the waypoints, otherwise an empty vector.

std::vector<*Path*> **getAllFanOut**(const std::string startName) const
Return a vector of paths fanning out from a particular start point.

Parameters **startName** – A pattern matching a start point.

Returns All paths fanning out from the matching startpoint, otherwise an empty vector.

std::vector<*Path*> **getAllFanIn**(const std::string endName) const
Return a vector of paths fanning out from a particular start point.

Parameters **endName** – A pattern matching an end point.

Returns All paths fanning in to the matching endpoint, otherwise an empty vector.

std::vector<std::reference_wrapper<const *Vertex*>> **getNamedVertices**(const std::string pattern = std::string()) const
Return a sorted list of unique named vertices in the netlist for searching.

Parameters **pattern** – A pattern to match vertices against.

Returns A vector of *Vertex* references.

inline std::vector<*Vertex**> **getNamedVerticesPtr**(const std::string pattern = std::string()) const
Return a vector of pointers to vertices that have names.

Parameters **pattern** – A pattern to match vertices against.

Returns A vector of pointers to *Vertex* objects.

inline std::vector<*Vertex**> **getNetVerticesPtr**(const std::string pattern = std::string()) const
Return a vector of pointers to net vertices.

Parameters **pattern** – A pattern to match vertices against.

Returns A vector of pointers to *Vertex* objects.

inline std::vector<*Vertex**> **getPortVerticesPtr**(const std::string pattern = std::string()) const
Return a vector of pointers to port vertices.

Parameters **pattern** – A pattern to match vertices against.

Returns A vector of pointers to *Vertex* objects.

inline std::vector<*Vertex**> **getRegVerticesPtr**(const std::string pattern = std::string()) const
Return a vector of pointers to register vertices.

Parameters **pattern** – A pattern to match vertices against.

Returns A vector of pointers to *Vertex* objects.

inline void **dumpDotFile**(const std::string &outputFilename) const
Write a dot-file represenation of the netlist graph to a file.

Parameters **outputFilename** – The file to write the dot output to.

inline bool **isEmpty**() const
Return true if the netlist is empty.

4.4 Options

class netlist_paths::**Options**
A class encapsulating options.

Public Functions

inline void **setMatchWildcard**()
Set matching to use wildcards.

inline void **setMatchRegex**()
Set matching to use regular expressions.

inline void **setMatchExact**()
Set matching to be exact.

inline void **setIgnoreHierarchyMarkers**(bool value)
Set matching to ignore (true) or respect (false) hierarchy markers (only with wildcard or regular expression matching modes). Note that hierarchy markers are '.', '/' and '_'.

inline void **setMatchOneVertex**()
Set matching to identify one vertex, and for it to be an error if more than one vertex is matched.

inline void **setMatchAnyVertex**()
Set matching to identify multiple vertices, and for just one to be chosen arbitrarily.

inline void **setTraverseRegisters**(bool value)
Enable or disable path traversal of registers.

inline void **setRestrictStartPoints**(bool value)
Set path start point restriction. When set to true, paths must start on top-level ports or registers. When set to false, paths can start on any variable.

inline void **setRestrictEndPoints**(bool value)
Set path end point restriction. When set to true, paths must end on top-level ports or registers. When set to false, paths can end on any variable.

inline void **setErrorOnUnmatchedNode**(bool value)
Setup the XML parser to raise an error when an unmatched node is encountered. For testing purposes only.

inline void **setLoggingVerbose**()
Enable verbose output.

inline void **setLoggingDebug**()
Enable debug output (including verbose messages).

inline void **setLoggingQuiet**()
Supress verbose and debug messages.

inline void **setLoggingErrorOnly**()
Supress warning, verbose and debug messages.

Public Static Functions

static inline *Options* &**getInstance**()
Return a reference to the *Options* object.

static inline *Options* ***getInstancePtr**()
Return a pointer to the *Options* object.

4.5 Path

class `netlist_paths::`**Path**
A class to represent a path through the netlist graph.

Public Functions

inline **Path**()
Construct an empty path.

inline **Path**(const std::vector<const *Vertex**> sourceVertices)
Constrct a *Path* from a list of Vertices.

Parameters **sourceVertices** – A vector of vertices to copy.

inline **Path**(const *Path* &path)
Copy constructor.

Parameters **path** – The *Path* to copy.

inline bool **operator==**(const *Path* &other) const
Test equality with another path.

Parameters **other** – The *Path* to test against.

Returns A Boolean indicating equal or not.

inline bool **contains**(const *Vertex* *vertex) const
Return true if the vertex is contained in this path.

Parameters **vertex** – The *Vertex* to test.

Returns A Boolean indicating whether vertex is contained in this path.

inline void **reverse**()
Reverse the order of vertices in this path.

inline void **appendVertex**(const *Vertex* *vertex)
Append a vertex to the end of this path.

Parameters **vertex** – The *Vertex* to append to this path.

inline void **appendPath**(const *Path* &path)
Append a path to the end of this path.

Parameters **path** – The *Path* to append to this path.

4.6 RunVerilator

class netlist_paths::**RunVerilator**
A class that provides facilities to run Verilator and produce XML netlists.

Public Functions

RunVerilator()
Default constructor. Locate the *Netlist* Paths Verilator executable relative to the location of the *Netlist* Paths library.

RunVerilator(const std::string &verilatorLocation)
Constructor, providing a path to a Verilator executable.

Parameters **verilatorLocation** – The location of the Netlists Paths Verilator executable (np-verilator_bin).

int **run**(const std::vector<std::string> &includes, const std::vector<std::string> &defines, const std::vector<std::string> &inputFiles, const std::string &topModule, const std::string &outputFile) const
Run Verilator.

Parameters

- **includes** – A vector of search paths for include files.
- **defines** – A vector of macro definitions.
- **inputFiles** – A vector of source file paths.
- **outputFile** – A path specifying an output file.

int **run**(const std::string &inputFile, const std::string &outputFile) const
Run Verilator with a single source file and no other options.

Parameters

- **inputFile** – A source file path.
- **outputFile** – A path specifying an output file.

4.7 Vertex

class netlist_paths::**Vertex**
A class representing a vertex in the netlist graph.

Public Functions

inline **Vertex**(*VertexAstType* type, Location location)
Construct a logic vertex.

Parameters

- **type** – The AST type of the logic statement.
- **location** – The source location of the logic statement.

inline **Vertex**(*VertexAstType* type, *VertexDirection* direction, Location location, std::shared_ptr<*DType*> dtype, const std::string &name, bool isParam, const std::string &paramValue, bool publicVisibility)
Construct a variable vertex.

Parameters

- **type** – The AST type of the variable.
- **direction** – The direction of the variable type.
- **location** – The source location of the variable declaration.
- **dtype** – The data type of the variable.
- **name** – The name of the variable.
- **isParam** – A flag indicating the variable is a parameter.
- **paramValue** – The value of the parameter variable.
- **publicVisibility** – A flag indicating the variable has public visibility.

inline **Vertex**(const *Vertex* &v)
Copy constructor.

inline std::string **getBasename**() const
Given a hierarchical variable name, eg a.b.c, return the last component c.

Returns The last heirarchical component of a variable name.

inline bool **isGraphType**(*VertexNetlistType* type) const
Match this vertex against different graph types.

Parameters **type** – A categorisation of a vertex.

Returns Whether the vertex matches the specified type.

inline bool **compareLessThan**(const *Vertex* &b) const
Less than comparison between two *Vertex* objects.

inline bool **compareEqual**(const *Vertex* &b) const
Equality comparison between two vertex objects.

inline bool **isTop**() const
Return true if the vertex is in the top scope.

inline bool **isPublic**() const
Return true if the vertex has public visibility.

inline bool **isSrcReg**() const
Return true if the vertex is a source register.

inline bool **isDstReg**() const
Return true if the vertex is a destination register.

inline bool **isLogic**() const
Return true if the vertex is a logic statement.

inline bool **isParameter**() const
Return true if the vertex is a parameter variable.

inline bool **isNet**() const
Return true if the vertex is a net variable.

inline bool **isReg**() const
Return true if the vertex is a register variable.

inline bool **isSrcRegAlias**() const
Return true if the vertex is an alias of a source register variable.

inline bool **isDstRegAlias**() const
Return true if the vertex is an alias of a destination register variable.

inline bool **isPort**() const
Return true if the vertex is a port variable. Handle a special case where port registers have REG VertexAstType.

inline bool **isCombStartPoint**() const
Return true if the vertex is a valid start point for a combinatorial path within the netlist.

- Source register
- Alias of a source register
- A top input or inout port

inline bool **isCombEndPoint**() const
Return true if the vertex is a valid end point for a combinatorial path within the netlist.

- Destination register
- Alias of a destination register
- A top output or inout port

inline bool **isStartPoint**() const
Return true if the vertex is a valid start point for a path.

inline bool **isEndPoint**() const
Return true if the vertex is a valid end point for a path.

inline bool **isMidPoint**() const
Return true if the vertex is a valid mid point for a path.

inline bool **canIgnore**() const
Return true if the vertex has been introduced by Verilator.

inline bool **isNamed**() const
Return true if the vertex has a name, ie is a variable of some description.

inline const char ***getSimpleAstTypeStr**() const
Map *Vertex* AST types to useful names that can be included in reports.

inline std::string **toString**() const
Return a string description of this vertex.

Public Static Functions

static inline bool **determineIsTop**(const std::string &name)
Return whether a variable name is a top signal.

A variable is 'top' when it is not prefixed with any hierarchy path. This applies to top-level ports and parameters only, since all other variables exist within a module scope.

Parameters **name** – The name of a variable.

Returns Whether the variable is in the top scope.

enum **netlist_paths::VertexAstType**
Vertex types corresponding to the Verilator XML AST format.

Values:

enumerator **ALWAYS**

enumerator **ASSIGN**

enumerator **ASSIGN_ALIAS**

enumerator **ASSIGN_DLY**

enumerator **ASSIGN_W**

enumerator **CASE**

enumerator **C_CALL**

enumerator **C_FUNC**

enumerator **C_METHOD_CALL**

enumerator **C_STMT**

enumerator **DISPLAY**

enumerator **DST_REG**

enumerator **DST_REG_ALIAS**

enumerator **FINISH**

enumerator **IF**

enumerator **INITIAL**

enumerator **INSTANCE**

enumerator **JUMP_BLOCK**

enumerator **LOGIC**

enumerator **PORT**

enumerator **READ_MEM**

enumerator **SEN_GATE**

enumerator **SFORMATF**

enumerator **SRC_REG**

enumerator **SRC_REG_ALIAS**

enumerator **VAR**

enumerator **WHILE**

enumerator **WIRE**

enumerator **INVALID**

enum netlist_paths::**VertexNetlistType**

Vertex categorisation within the netlist graph, used for selecting collections of vertices with particular properties.

Values:

enumerator **DST_REG**

enumerator **DST_REG_ALIAS**

enumerator **END_POINT**

enumerator **IS_NAMED**

enumerator **LOGIC**

enumerator **MID_POINT**

enumerator **NET**

enumerator **PORT**

enumerator **REG**

enumerator **SRC_REG**

enumerator **SRC_REG_ALIAS**

enumerator **START_POINT**

enumerator **ANY**

enum netlist_paths::**VertexDirection**
The direction of a variable, applying only to ports.

Values:

enumerator **NONE**

enumerator **INPUT**

enumerator **OUTPUT**

enumerator **INOUT**

4.8 Waypoints

class netlist_paths::**Waypoints**
A class representing a set of waypoints to constrain a search for a path.

Public Functions

inline **Waypoints**()
Construct an empty *Waypoints* object.

inline **Waypoints**(const std::string start, const std::string end)
Construct a *Waypoints* object with patterns matching start and finish points.

Parameters

- **start** – A pattern specifying a start point.
- **end** – A pattern specifying an end point.

inline void **addStartPoint**(const std::string name)
Set a named start point.

Parameters **name** – A pattern specifying a start point.

inline void **addEndPoint**(const std::string name)
Set a named end point.

Parameters **name** – A pattern specifying an end point.

inline void **addThroughPoint**(const std::string name)
Add a through point.

Parameters **name** – A pattern specifying a mid point.

inline void **addAvoidPoint**(const std::string name)
Add a point to avoid.

Parameters **name** – A pattern specifying a mid point to avoid.

inline const std::vector<std::string> &**getWaypoints**() const
Access the waypoints.

Returns An ordered vector of patterns for each waypoint.

inline const std::vector<std::string> &**getAvoidPoints**() const
Access the avoid points.

Returns An ordered vector of patterns for each avoid point.

CHAPTER

FIVE

PYTHON API REFERENCE

INDEX

N