POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

# CAD Tools for Emerging Nanotechnologies

Relatori:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Prof. Fabrizio Riente

Candidato:
Francesco Russo

Aprile 2019

# Summary

With the scaling of CMOS technology slowly coming to a halt due to quantum mechanics effects progressively eroding its performance as transistor dimensions shrink, new and experimental technologies for digital circuits are being explored and prototyped with increased effort. Design tools for such technologies have been created for these purposes and their development and refinement is now more critical than ever.

ToPoliNano, developed by the VLSI group of the Polytechnic University of Turin, is one such tool, dedicated to the design and simulation of digital circuits built with field-coupled Magnetic QCA[1] technologies. A robust EDA tool by itself with an easy-to-use GUI, it only missed, with regards to the more established professional software available to circuit designers that work with CMOS technology, the flexibility of a dual GUI/command-line offering.

The aim of this Master's thesis consisted therefore of expanding the capabilities of ToPoliNano by equipping it with a text-based workflow, allowing users to access the program's powerful features by means of text commands and automation scripts. In order to bring the aforementioned text-based workflow to life, three software modules have been developed in succession as part of the thesis, each complementing the others:

- a command-line interface (CLI) able to tokenize, parse and interpret a variety of text commands mapped to the functionalities of ToPoliNano (for example: layout generation, layout import/export, circuit simulation, etc.);

- a new general-purpose preprocessor capable of processing variables, loops and conditional statements, which unlocks the possibility of writing complex scripts that go beyond a simple list of commands;

- a console mode (loaded into terminal), referred to as no-GUI mode, which directly interfaces the user with the CLI (and, by extension, to ToPoliNano) without the need to load the GUI (which also allows the user to interact with the CLI but via widgets included in the graphical interface).

[1]Quantum-dot Cellular Automata: https://www.ece.uic.edu/~vmetlush/2005-6.pdf

As the final part of the thesis, a comprehensive user manual was written - illustrating the purpose of each supported text command, the structure of the command-line work-flow and the advanced scripting features available to the user, including a collection of example scripts - to help the user gain a complete, in-depth understanding of the new instruments at their disposal for designing digital circuits.

# Table of contents

# Chapter 1

# Introduction

ToPoliNano is a powerful CAD tool developed for designing and simulating digital circuits built with experimental nanomagnetic technologies based on magnetic QCA.

In field-coupled Magnetic QCA (also referred to as "MQCA") technologies "*the base cell is a single domain nanomagnet, with only two possible magnetizations which represent the two logic values 0 and 1*"[1], replacing the CMOS representation of logic values with voltage levels. Magnetic QCA is of great interest thanks to the substantial advantages it carries over CMOS: "*extreme low power consumption and intrinsic memory ability, i.e. the possibility to implement circuits with mixed computational and memory abilities*"[2],; moreover, MQCA-based circuits "*can be realized with current technology, with high-end electron beam lithography*"[3], which facilitates their manufacturing.

The design workflow for MQCA circuits is made up by a sequence of steps not unlike those of traditional CMOS design procedures:

- circuit compilation, which consists of parsing the VHDL file and creating an abstract, hierarchical tree of nodes from it, representing the HDL ciruit structure itself;
- layout generation, which takes the aforementioned graph as input and produces a working, physical layout of the circuit made of nanomagnets;
- circuit simulation, which - by means of a testbench provided by the user - stimulates the circuit with test inputs, reads its outputs and displays them to the user

[1]M. Graziano, M. Vacca and M. Zamboni, "Magnetic QCA Design: Modeling, Simulation and Circuits". *Cellular Automata - Innovative Modelling for Science and Engineering* (2011): p. 37. Available from: https://www.intechopen.com/books/cellular-automata-innovative-modelling-for-science-and-engineering/magnetic-qca-design-modeling-simulation-and-circuits

[2]M. Graziano, M. Vacca and M. Zamboni, "Magnetic QCA Design: Modeling, Simulation and Circuits" (2011): p. 37

[3]M. Graziano, M. Vacca and M. Zamboni, "Magnetic QCA Design: Modeling, Simulation and Circuits" (2011): p. 42

for verification purposes.

Said workflow is illustrated in Figure 1.1[4].

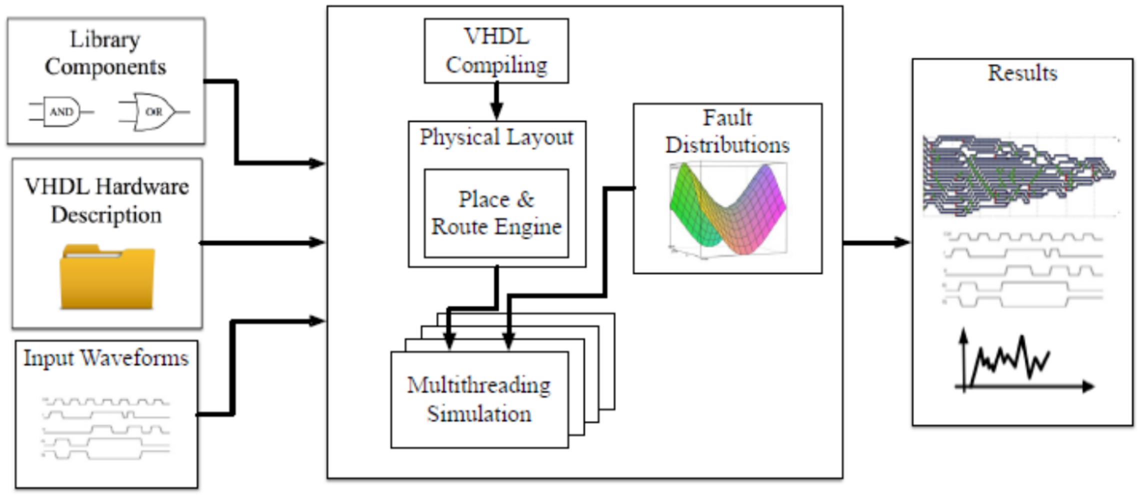


**Figure 1.1**: Design workflow of ToPoliNano

ToPoliNano possesses additional, secondary functionalities that integrate the core workflow: a successfully generated layout may be saved to file for later use, and it is possible to import a pre-existing layout into the program.

The objective of this thesis has been to make ToPoliNano a more versatile EDA tool by developing a suite of software modules that enable the user to harness the features of ToPoliNano by means of text-based commands and, by extension, automation scripts, thereby offering a powerful alternative to the pre-existing GUI workflow:

- a command-line interface or "CLI" (described in Chapter 2), which is tasked with translating text commands into calls to the relevant functionalities of ToPoliNano and with autonomously executing scripts;

- a general-purpose preprocessor (described in Chapter 3), whose role is to "pre-digest" (i.e. pre-process) scripts by simplifying complex control flow statements such as loops and if-constructs, with the end product being a streamlined script with no such structures which can then be passed to the CLI for execution;

[4]S. Pennavaria, "Design and implementation of a graphic editor for nanotechnologies" (2015): p. 15

- a console mode or "no-GUI mode" (described in Chapter 4), which allows the user to load ToPoliNano in a terminal window and to interact with it by means of text commands and scripts without also loading its GUI.

In order to ensure that the CLI and preprocessor features are appropriately documented, well-understood by the user and correctly used, a user manual was also written (it can be found in the Appendix): its contents include descriptions of all command keywords and their options, preprocessor syntax for loops, variables and if-constructs and example scripts to get the user started writing their own.

# Chapter 2

# Command-line interface

Most (if not all) present-day programs sport a graphical user interface (GUI), which helps the user navigate software functionalities by means of mouse interaction. While this approach is the easiest (especially to first-time users of a program) and the least error-prone, it is also the slowest since today's powerful hardware, capable of extremely fast processing, has to wait for manual input by a human being every time a task has been completed.

A much older means of machine-human communication is the command-line interface (CLI). At first glance there is little difference between a GUI and a CLI in terms of efficiency: a GUI lets the user control a program via mouse clicks, while a CLI lets the user enter text commands. Where the CLI shines, though, is in the power of automation it grants to the user. This power, harnessed via script files, allows a human being to package a series of newline-separated commands into a text file and feed it to the program, which then proceeds to unpack it and execute its contents in order, relieving the user from the need of constant presence at the screen, a benefit that becomes apparent as the automated tasks hit (and exceed) the double-digits.

For this reason, a CLI was developed for ToPoliNano to let the user take control of it via text commands and automate tasks with ease by means of script files.

## 2.1 Functional overview

The CLI supports execution of all core functionalities of ToPoliNano via a number of keywords, each with its own set of arguments (some of which introduced by a -[letter]/--[word] expression):

- compilation of a VHDL file is performed via the "compile" keyword, followed by the file path (including the .vhd/.vhdl file extension);

- the layout of a compiled circuit is produced by entering the "layout" keyword, with several arguments available that specify the fine details of the process (like the sequence of optimization algorithms to apply, the maximum accepted fanout, etc.);

- the simulation of a circuit layout is started via the "simulate" keyword, followed by the path of a testbench file, and many arguments are offered to fine-tune the time parameters (resolution, rise and fall times, clock cycle, etc.) and other characteristics;

- a layout may be saved in .qll format via the "save" keyword, followed by the desired file path;

- a screenshot of the layout can be exported by entering the "export" keyword, and its format specified through a dedicated argument;

- a layout in .qll format is loadable into ToPoliNano by means of the "import" keyword, followed by the path of the .qll file;

- a ToPoliNano script file (which is a simple text file ending in .do format) may be fed to the program via the "do" keyword, followed by the file path;

- program closure is triggered via the "quit" keyword.

Additionally, if the user wants to clear the CLI of all printed log messages they may enter a special keyword: "clear".

By convention, a "command" is formed by a keyword followed by any number of its arguments (some of which may be mandatory, like for example the testbench argument that follows the "simulate" keyword). A comprehensive representation of the naming conventions adopted in this text in relation to the elements of a command can be found in Figure 2.1.

The complete list of keywords with their respective arguments is made available below:

- compile <top-level-filename.vhd>;

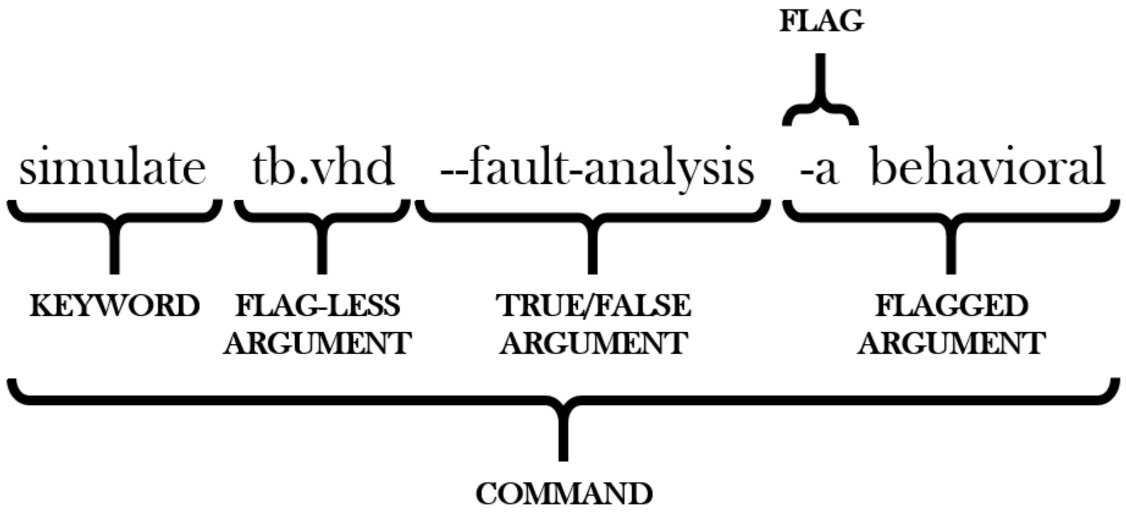


**Figure 2.1**: Naming conventions used in this text for the elements of a CLI command

- layout

  [options]:

  i. -a, --algorithm (barycenter, kl, sa_exponential, sa_timberwolf);
  ii. -d, --design-approach (flat, p_hiearchical, f_hierarchical);
  iii. -l, --load;
  iv. -g, --geometry (width=50, height=100,
  horizontal_space=20, vertical_space=20);
  v. -f, --fanout (integer=2);
  vi. -v, --vertical-wire (integer=4);
  vii. -m, --magnet-clockzone (integer=4);
  viii. -s, --sa-embedded;
  ix. -w, --domain-wall;
  x. -t, --satimberwolf-parameters (low_temp=0.1,
  high_temp=40000, first_alpha=0.8, second_alpha=0.95,
  third_alpha=0.8, first_temp=1000, second_temp=10000,
  n_iterations=20);
  xi. -e, --saexponential-parameters (low_temp=0.1,
  high_temp=10000, alpha=0.95, n_iterations=20);

xii. -o, --output <path/to/destination/filename.qll>;

- save <path/to/destination/filename.qll>;

- export
  [options]:
  i. -o, --output <path/to/destination=workspace/Results>;
  ii. -n, --filename <filename=top-level-name>;
  iii. -f, --format <svg, png, bmp, jpeg, pdf, ps>;
  iv. -a, --area <scene, selected, view>;

- import <qll-full-path.qll>;

- quit;

- simulate <testbenchfile.vhd>
  [options]:
  i. -t, --time (int=1[unit=us]), unit can be ps, ns, $\mu$s, ms;
  ii. -r, --resolution <1ps, 10ps, 100ps, 1ns, 10ns,
  100ns, 1us, 10us, 100us, 1ms>;
  iii. -c, --clock (int=3[unit=ns]), unit can be ps, ns, $\mu$s, ms;
  iv. -u, --risetime (int=0[unit=ns]), unit can be ps, ns, $\mu$s, ms;
  v. -d, --falltime (int=0[unit=ns]), unit can be ps, ns, $\mu$s, ms;
  vi. -a, --algorithm (behavioral, single_domain);
  vii. -f, --fault-analysis;
  viii. -v, --max-variations (int x=3, int y=2);
  ix. -i, --iterations (int=1000);
  x. -o, --output (path/to/destination=workspace/Results).

The CLI accepts the following path types:

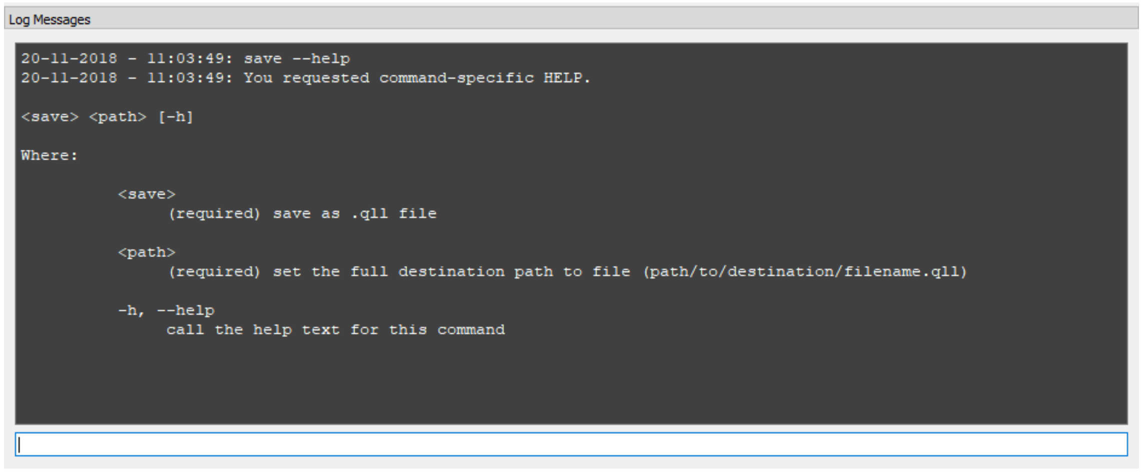
- just the filename (e.g. "circuit.vhd"), in which case ToPoliNano will look for the file starting from the relevant workspace directory ("Input_Files" for VHDL entities, "Testbenches" for VHDL testbenches, etc.);

- relative path (e.g. "project_3/circuit.vhd"), which prompts ToPoliNano to look for the file in the same manner it does for the previous path type;

- absolute path (e.g. "C:/Users/Alex/Project_Files/circuit.vhd" for Windows users, "/home/user/Documents/Project_Files/circuit.vhd" for UNIX users).

Detailed help text describing usage of all keywords and arguments can be called via the standalone "-h/--help" argument; alternatively, "[keyword] -h" or "[keyword] --help" may be entered to only print help text for the chosen keyword and its related arguments, as shown in Figure 2.2.

Lastly, like many others, the CLI of ToPoliNano supports command history of the current session: pressing the "up" (↑) key will retrieve older commands while pressing the "down" (↓) key will retrieve more recent ones, as illustrated by Figure 2.3. Command history is wiped on program exit.



```
20-11-2018 - 11:03:49: save --help
20-11-2018 - 11:03:49: You requested command-specific HELP.

<save> <path> [-h]

Where:

          <save>
               (required) save as .qll file

          <path>
               (required) set the full destination path to file (path/to/destination/filename.qll)

          -h, --help
               call the help text for this command
```

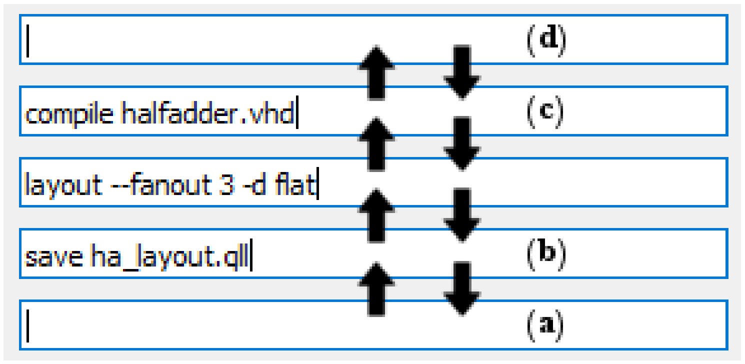
**Figure 2.2**: Command-specific help for "save"

## 2.2 Designing the CLI

In this section, the inner workings of the CLI are illustrated and some key design choices are discussed.

The CLI was developed on top of the powerful Qt library, which already powers ToPoliNano itself, and designed following the Model-View-Controller (MVC) architectural pattern:

- the View consists of a single-line text editor where the user enters commands, plus a pre-existing log box (already tasked with giving updates on the results of



```
compile halfadder.vhd
layout --fanout 3 -d flat
save ha_layout.qll
```

**Figure 2.3**: Command history: a) initially blank editor; b) pressing "up" produces the most recently entered command; c) the oldest entered command is reached; d) after reaching the oldest command, the editor is blank again; pressing "down" will reverse the process.

operations) which warns the user about any errors caused by improper formatting of the commands themselves;

- the Controller is a class that forwards user commands to the CLI back-end, receives the information extracted by the latter and sends it to the "main" module of ToPoliNano, which will start the appropriate operation;

- the Model, also referred to as "CLI back-end", is a class tasked with tokenizing, parsing and interpreting a user command, i.e. extracting and classifying the information it holds, which is then sent back to the Controller.

These three components are completely independent from one another, as the MVC pattern requires; additionally, they are *unaware* of each other, thanks to their communications occurring over Qt's signal/slot system (briefly explained in Sub-section 2.2.1). These two characteristics ensure that the CLI components are as loosely coupled as possible, for easier maintenance and upgrade of the source code.

The full capabilities of the CLI components extend beyond this brief introduction and are illustrated in Sub-sections 2.2.2, 2.2.3 and 2.2.4.

### 2.2.1 Qt's signal/slot system

The main reason for the existence of Qt's signal/slot system is to enable communications between GUI widgets without resorting to callbacks; however, any class derived from `QObject` (Qt's base class) may possess signals and slots of its own without having to be a widget; this makes the signal/slot system a versatile tool that lends itself to purposes other than what Qt originally planned for it, including achieving detailed inter-class communications and very loose coupling at the same time.

Signals have the appearance of function declarations and are declared in a class definition under the "signals" keyword, but do not have an implementation; a signal can have any number of parameters (no parameters is also a possibility), ranging from native types such as `int` and `bool`, to standard-library classes like `string` and `vector` , to user-defined classes. Since a signal is not a function, it cannot simply be called; instead, it can be "launched" (or "emitted", in Qt terms) by the class it belongs to by putting the `emit` keyword before the call (made with the arguments that its function signature requires). A more in-depth explanation of signal emission is given further down this section.

Slots are member functions that are declared in a class definition under the "slots" keyword; if called explicitly, they act like regular functions but, if a signal and a slot have been previously "linked" via the `QObject::connect()` function, when the signal is emitted the "connected" slot will be called as a consequence.

In order to be connected, a signal and a slot must have the same function signature, i.e. the same number of parameters, with the same types (native or otherwise), in the same order. In fact, when a signal is "emitted", it carries data within itself (made up by the arguments it was emitted with), and the slot is called with that same data as arguments, thereby achieving a data transmission between the class that owns the signal and the class that owns the slot. An example of signal and slot declaration, connection and interaction is provided in Listing 2.1.

```
class Sender : public QObject
{
  Q_OBJECT
  //...
signals:
  exampleSignal(int value, bool flag);

public:
  void sendSignal()
  {
      emit exampleSignal(3, true);
      return;
    }
    //...
}

class Receiver : public QObject
{
  Q_OBJECT
  //...
slots:
  void exampleSlot(int value, bool flag)
  {
    // Process data...
    return;
  }
  //...
}

int main()
{
  Sender s;
  Receiver r;

  QObject::connect(&s, SIGNAL( exampleSignal(int,bool) ),
                   &r, SLOT( exampleSlot(int,bool) );
  s.sendSignal(); // exampleSignal is emitted
  // exampleSlot is called immediately after

  return 0;
}
```

**Listing 2.1:** Example code written to illustrate signal and slot declaration, connection and interaction.

If a signal and a slot forming a pair belong to two classes that live in the same thread, their connection is called "direct", which means that when the signal is emitted, execution immediately jumps to the slot as if it was explicitly called in place of the emission, with no other instructions in-between. If, instead, the two classes live in two different threads A and B (managed via instances of the Qt class `QThread`), the connection is called "queued"; in this case, when the signal is emitted from `QThread` A it is wrapped by `QThread` B into an "event" (a Qt object used to represent a variety of occurrences, including user interactions with the GUI) and put into the so-called "event queue" of `QThread` B (each `QThread` has its own independent event queue); once older events have been processed by `QThread` B, the event containing the emitted signal is processed, triggering the execution of the corresponding slot. This entails that if signals 1, 2 and 3 are emitted one after the other from `QThread` A they will be stacked onto the event queue of `QThread` B in that same order and will be processed by `QThread` B (triggering, one after the other, the slots they are linked to) in said order, preserving the intended sequence. A visual representation of the event queue is provided in Figure 2.4.

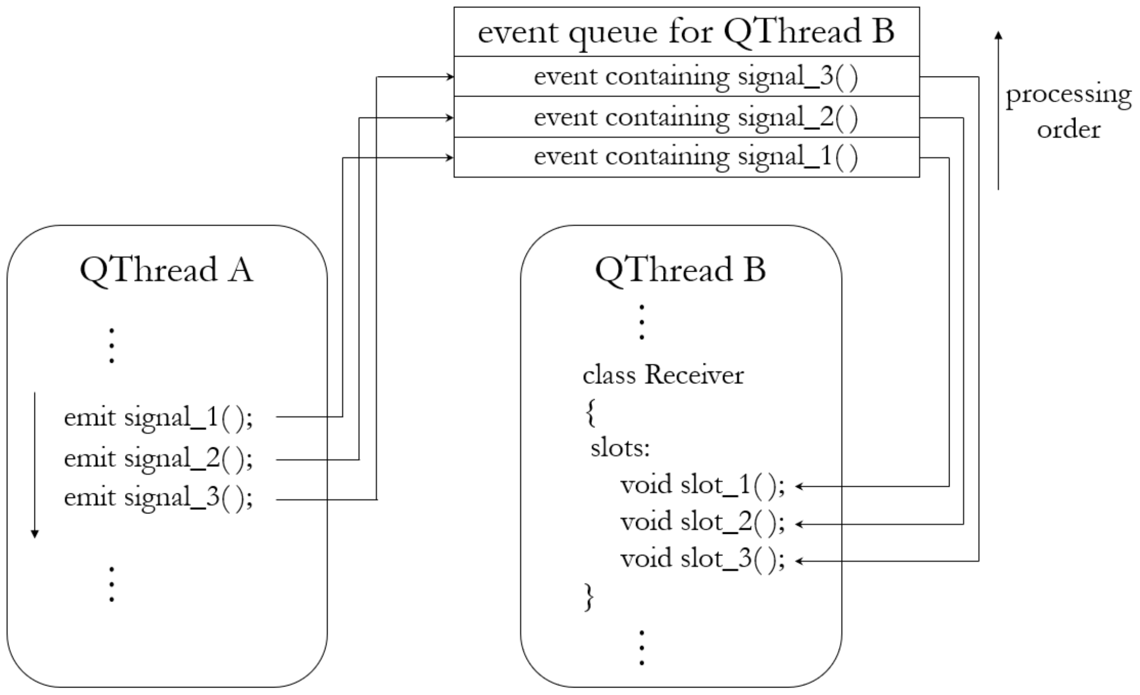
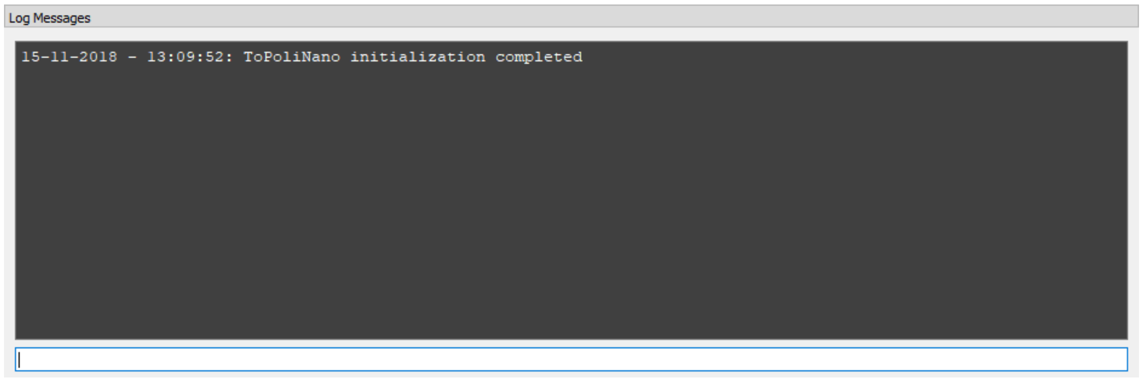


**Figure 2.4**: Three signals (1, 2, 3) are emitted from QThread A; they are wrapped into events by thread B and put into its own event queue; they are then processed by QThread B in the same order they were emitted, triggering the corresponding slots one after the other.

Qt's built-in per-thread event queue is employed by the CLI, together with a simple semaphore, to put commands in a queue when ToPoliNano is busy performing a computationally-intensive operation (like, for example, building the layout of a circuit or performing a simulation) so that they can be executed correctly and in order by ToPoliNano once the program is done with the current task, instead of being lost.

### 2.2.2 GUI elements

The text editor and the log box that allow interaction with the CLI (shown in Figure 2.5) are similar to those commonly found in programs like Synopsys® Design Compiler® and Mentor Graphics® Modelsim®.



```
15-11-2018 - 13:09:52: ToPoliNano initialization completed
```

**Figure 2.5**: Text editor and log box.

The user writes a command into the editor; then, pressing the Enter key clears the editor itself and sends the command (via a signal) to the CLI controller for processing. The entered command is also printed to the log box (a `QTextEdit` object) as a timestamped message.

For the purpose of enabling the user to browse command history via the "up" and "down" keys from within the editor, a `LogLineEdit` class (which constitutes the single-line editor) was derived from Qt's `QLineEdit` class in order to reimplement the `keyPressEvent()` event-catcher function of the latter; the reimplemented version - on either key press - sends a signal to the CLI controller (which keeps a record of all commands entered by the user in the active session) detailing whether the requested command is older ("up") or newer ("down") than the one currently in the editor (see Figure 2.3 for a visual representation of the history mechanism).

Log messages are also printed to the log box to update the user as ToPoliNano executes the action related to an entered command.

Due to the editor and the log box being attributes of the same class (called `LogPannel` ), the special "clear" keyword is directly handled by said class (instead of by the CLI back-end), which clears the log box.

### 2.2.3 Controller

The CLI controller acts as the mediator between the CLI back-end on one side and the main functionalities of ToPoliNano plus the GUI elements of the CLI on the other. To do so, it is equipped with a number of signals and slots that send and receive different kinds of data to and from each side:

- a slot receives command-carrying signals coming from the GUI and forwards them to the back-end;
- for each keyword except "do" (which does not involve any functionality of ToPoliNano), a keyword-specific slot picks up the relevant data-carrying signal coming from the back-end and forwards the data to ToPoliNano via a keyword-specific signal;
- a slot receives log-message-carrying signals coming from the back-end and transmits the messages to the GUI;
- a slot receives history-query signals coming from the GUI (the ones triggered by "up" and "down" key presses) and transmits back to the GUI a command from history via a signal.

Whenever the user enters a command, it is appended by the CLI controller to the command history (a `QVector<QString>` called `reqHistory`). In order to make said history browsable by the user, an iterator moves through the vector: every time the user goes backward through history, the iterator is decreased by 1 and then the `reqHistory` element at index value equal to the iterator is sent to the GUI, and viceversa it is increased by 1 when the user goes forward through history.

A visual representation of the communications occurring within the CLI via signals and slots is shown in Figure 2.6.

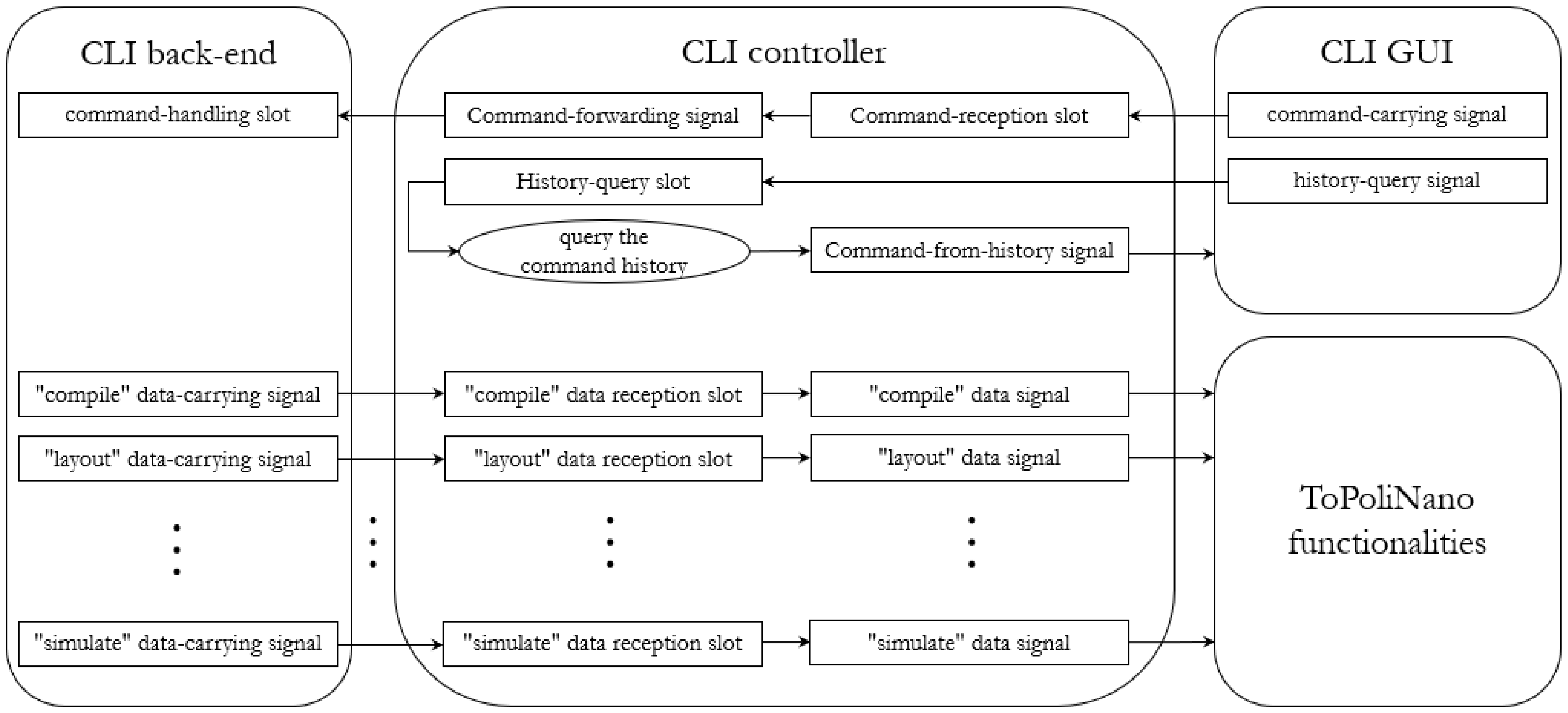

CLI back-end
command-handling slot
"compile" data-carrying signal
"layout" data-carrying signal
"simulate" data-carrying signal
CLI controller
Command-forwarding signal
Command-reception slot
History-query slot
query the command history
Command-from-history signal
"compile" data reception slot
"compile" data signal
"layout" data reception slot
"layout" data signal
"simulate" data reception slot
"simulate" data signal
CLI GUI
command-carrying signal
history-query signal
ToPoliNano functionalities

**Figure 2.6**: Internal CLI communications via signals and slots.

### 2.2.4 Back-end

The CLI back-end is the data-crunching part of the CLI, which performs the whole sequence of operations needed to properly extract information from a text command:

- tokenization of the command (i.e. its splitting into sub-strings and reorganization into an appropriate data structure);
- parsing of the tokens in order to recognize their roles (e.g. keyword, argument, etc.);
- interpretation of the parsed tokens to check if the information they contain is valid and to package it for transmission.

The back-end functionalities are implemented in a single class named `CLIbackend`.

Since the received command is formatted as a `QString` (Qt's own string class), its tokenization is trivial as the class features a handy .split() function which returns its contents as a `QStringList` (i.e. a list of `QString`). Its elements are then moved into a `std::vector<std::string>` for reasons that will be explained in the following paragraph. Before moving on to the parsing stage, the back-end takes note of the first element - the keyword ("compile", "save", etc.) - of the vector.

The parsing stage is performed by leveraging an open-source library called TCLAP[1], chosen for its ease of use and detailed documentation over Qt's, `QCommandLineParser` class, which is clunkier and more convoluted. The TCLAP library offers various "argument" classes, each representing a different kind of argument, for example flag-less argument (`UnlabeledValueArg`), flagged argument (`ValueArg`) and true/false argument (`SwitchArg`) - collectively referred to as `Arg` classes in this text for the sake of brevity - and a `CmdLine` class which performs the actual token parsing. TCLAP works with the `std::string` class instead of the `QString` class, hence the aforementioned conversion from `QStringList` to `std::vector<std::string>`.

At the beginning of the parsing stage a `CmdLine` object is armed with a specific set of `Arg` objects, each representing one of the arguments that can follow the keyword (which the back-end had taken note of for this purpose). After that, the token vector (i.e. the `std::vector<std::string>`) created in the tokenization stage is fed to the

[1]http://tclap.sourceforge.net/

CmdLine object for parsing: each token is consequently identified and assigned to the corresponding Arg object.

The third and last stage the back-end goes through is the interpretation stage, the longest and most crucial. In the interpretation stage, the Arg objects the CmdLine object was armed with are queried via their getValue() function; each Arg object returns the value of its corresponding argument or a default value if that argument was not explicitly mentioned by the user in the command.

Each retrieved value is then run through a series of checks to make sure it is valid: for example the testbench argument for the "simulate" keyword must be a string ending with ".vhd" or ".vhdl", the --design-approach argument for the "layout" keyword can only have one of three values ("flat", "p_hiearchical", "f_hiearchical"), and so on. If all values are confirmed to be valid, they are then transmitted to the CLI controller via signal emission. An example of checks and signal emission performed by the back-end is provided in Listing 2.2.

```
if (commandVal.compare("export") == 0)
{
    // OUTPUT DIRECTORY
    std::string outputExportString = outputExportArg->getValue();
    QString outputStr;
    if (outputExportString.compare("default") == 0) // If the
       user did not employ the "--output" option...
        outputStr = workPath + "/" + resultsPath + "/"; // ...Set
           the default results directory as the output directory
    else
    {
        outputStr = QString::fromStdString(outputExportString);
        validatePath(outputStr, false, workPath + "/" +
           resultsPath + "/");
    }

    // FILENAME
    std::string exportFilenameString = exportFilenameArg->
       getValue();
    if (exportFilenameString.compare("") == 0) // If it is an
       empty string...
    {
        usageAndErrorsGen->interpreterError("filename cannot be
            an empty string", commandArg->shortID(),
            exportFilenameArg->shortID() );
        return false; // Abort interpretation without executing
            the request
    }

    // FORMAT
    std::string formatString = formatArg->getValue();
    if ( (formatString.compare("svg") != 0) &&
         (formatString.compare("png") != 0) &&
         (formatString.compare("bmp") != 0) &&
         (formatString.compare("jpeg") != 0) &&
         (formatString.compare("pdf") != 0) &&
         (formatString.compare("ps") != 0) )
    {
        usageAndErrorsGen->interpreterError("invalid format",
            commandArg->shortID(), formatArg->shortID());
        return false; // Abort interpretation without executing
            the request
    }

    // AREA
    std::string area = areaArg->getValue();
    if ( (area.compare("scene") != 0) && (area.compare("selected"
        ) != 0) && (area.compare("view") != 0) )
    {
        usageAndErrorsGen->interpreterError("invalid area",
            commandArg->shortID(), areaArg->shortID() );
        return false; // Abort interpretation without executing
            the request
    }

    emit SignalBackendExport( outputStr,
      QString::fromStdString( exportFilenameString ),
      QString::fromStdString( formatString ),
      QString::fromStdString( area ) ); // Send the data to the
          CLI controller
```

```
}
```

**Listing 2.2**: Correctness checks and signal emission for "export"-keyword commands

**Error handling**

The aforementioned sequence of operations is successfully completed only if the command contains no mistakes; conversely, if the command is in some way faulty, the CLI will not transmit it on grounds of either a parsing error or an interpretation error.

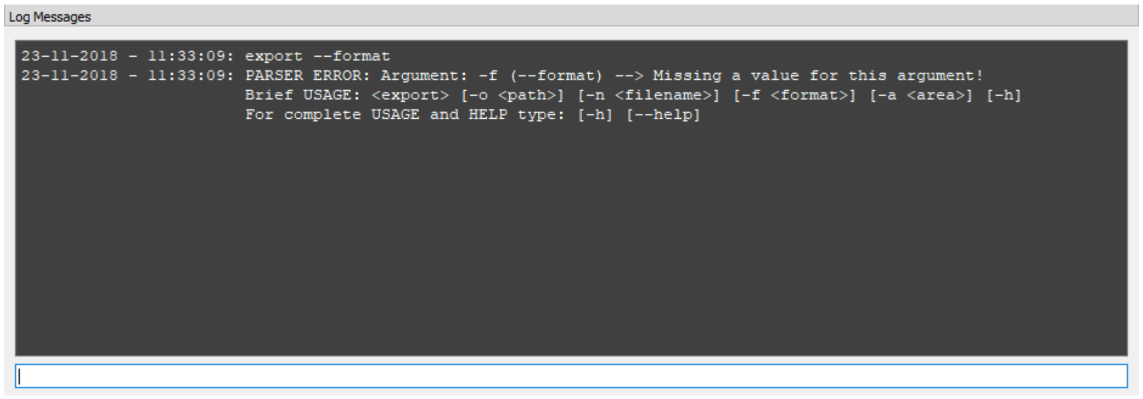
A parsing error, like the name suggests, is triggered during the parsing stage if the parser (the `CmdLine` object) is not able to correctly parse the command; it is caused by:

- an invalid keyword, for example "comppile" (which contains a typo) or "start" (which is not among the accepted keywords);
- an invalid argument flag for the employed keyword, like in "export -x svg" ("export" does not contemplate a flagged argument with "-x" as flag);
- a flagged argument that is missing a value on its right, for example "layout --fanout" (an integer must be specified as value after the flag), or that contains a value of invalid type, like "layout --fanout three" (the argument value is a string, while it should be an integer);
- an invalid true/false argument for the employed keyword, like in "layout --quick" (no "--quick" true/false argument is available for "layout");
- the command being an empty string;
- a mandatory argument missing, like the VHDL circuit argument that follows the "compile" keyword.

Each of these mistakes is described by an appropriate error message, like the one in Figure 2.7, followed by a brief help text for the employed keyword.

An interpretation error, instead, is triggered during the interpretation stage. The difference with regards to parser errors lies in the fact that the parser does not actually know what the argument values should look like, apart from their type, therefore it will not object to incorrect values that still fit the data type. Thus, an interpretation error is caused by:

- a file path not ending with the appropriate file extension (for example, the test-bench argument for "simulate" must end either in .vhdl or in .vhd) or that is too short (for example ".vhdl" by itself is not accepted because a filename must contain at least one character apart from the file extension);
- an incomplete set of comma-separated integer values, like in "layout --geometry 60,120" which only contains 2 out of 4 required integer values;
- a string not coinciding with one of the accepted literals, for example in "layout --algorithm kj" where "kj" is not one of the four accepted literals ("barycenter", "kl", "sa_exponential", "sa_timberwolf").



```
23-11-2018 - 11:33:09: export --format
23-11-2018 - 11:33:09: PARSER ERROR: Argument: -f (--format) --> Missing a value for this argument!
                       Brief USAGE: <export> [-o <path>] [-n <filename>] [-f <format>] [-a <area>] [-h]
                       For complete USAGE and HELP type: [-h] [--help]
```

**Figure 2.7**: Parser error caused by "--format" having no value to its right

When a parsing or interpreter error occurs, the command data is not transmitted to the controller: instead, an error message is transmitted by the back-end to the controller and by the controller to the log box, where it is printed, and ToPoliNano remains in idle state. In case the faulty command was part of a script, the execution of the script will be aborted, instead of skipping to the following one in the file; this is done to prevent ToPoliNano from behaving unexpectedly.

### Extending the functionalities of TCLAP

As mentioned before, TCLAP was chosen for its ease of use and detailed documentation, but having been originally conceived as a basis for terminal applications it came with a few limitations that had to be overcome in order to produce the desired functionalities.

Firstly, when a parsing error occurs the `CmdLine` class relies, for error output, on another TCLAP class called `StdOutput`. Such class is designed to print parsing errors on `std::cerr` (the standard output stream for errors) and to print help text on `std::cout` (the standard output stream). While this implementation is perfectly fine for a terminal application, it is not if the CLI is being developed for use in a GUI-based program, which does not rely on standard output streams for communication with the user. Moreover, the output string is split into 75-character lines made of whole words (i.e. not hyphened or broken) and is then printed one such line at a time; again, this solution provides terminal-friendly output which however proves not only unnecessary, but also inconvenient on GUI, especially considering that the program window is resizable (thus the line length is variable) and that the `QTextEdit` class (the log box) already wraps all text without breaking words - and does so in real-time when the window is resized.

Secondly, since a typical terminal application has different functionalities and options assigned to distinct flags, the `CmdLine` class does not support "conditional" assignment of a functionality to a flag depending on whether a certain condition is met or not; this entails that a flag could not lead to a different functionality depending on the keyword that preceded it. Additionally, the class does not provide a way to "reset" its argument list, making the workaround of simply "resetting and rearming" the parser at every cycle unfeasible.

A few other limitations were also present but will not be discussed as they are trivial compared to the ones mentioned above.

Fortunately, as the documentation of TCLAP reports[2], if the developer wants different output beahviour than the default one, the library supports subclassing `StdOutput` and using the derived class as a replacement to it. This prompted the creation of a new class, `customOutput`, derived from `StdOutput`, to reimplement much of the original code devoted to producing output; additionally, as TCLAP is strictly a parsing library it is not designed for interpretation and thus does not produce interpretation errors, therefore `CLIbackend`, which performs interpretation, is also tasked with generating interpretation errors.

[2]http://tclap.sourceforge.net/manual.html#CHANGE_OUTPUT

Remembering that the back-end constitutes the "model" part of the MVC trifecta (which entails that it must not update the GUI directly), in `customOutput` any parsing errors or help text are not printed to standard output anymore; instead, they are saved to a `QString` object called `cliOutput` that it shares with `CLIbackend`, as `CLIbackend` itself saves any interpretation errors in it too. Since, in case a command is such that it will trigger multiple parser and/or interpretation errors, the first one that the back-end encounters will cause it to halt its operations, transmit an error message to the controller, clear `cliOutput` and then wait for the next command to be received, `cliOtuput` always contains at most one error message, guaranteeing that it will not mistakenly retain older messages and consequently confuse the user with unrelated warnings when printed to the log box.

To circumvent the lack of flexibility of `CmdLine` with regards to flags, `CLIbackend` employs dynamic memory management (i.e. runtime creation and destruction of objects): at the beginning of its operations when a command has just been received, `CLIbackend` creates at run-time an empty `CmdLine` object, then it takes note of the keyword (the first word in the command) and proceeds to arm the `CmdLine` object with the appropriate set of `Arg` objects for that keyword, thus creating a tailor-made parser. Once the interpretation stage is over, `CLIbackend` deletes at run-time the `CmdLine` object.

Thus, at the end of the whole sequence, no trace is left of said object, and the cycle can start over fresh: a new empty `CmdLine` object is created and then fitted to the keyword of the current command, it parses the command itself and is then deleted once interpretation is over. While this solution introduces a slight overhead with dynamic memory allocation, it makes the CLI a weightless module in memory so long as it is not working on a command (i.e. while it is idle).

### The script-execution algorithm

Interpreting a command with a keyword linked to any functionality of ToPoliNano is a relatively straightforward operation: the contents of the command are extracted, classified and packaged for transmission back to ToPoliNano through the controller. However, interpretation of a command requesting the execution of a script (i.e. "do scriptname.do") is a different task, a "meta"-task of sorts, since a script file contains a list of commands in a specific order, and can even include more script-execution commands too (a situation which may be called "nested scripting"); this calls for a dedicated algorithm capable not only of handling scripts, but also of supporting any nesting depth. Such algorithm consists of two parts: depth-independent interpretation of a script-execution command and a command-selection loop.

The interpretation stage of a script-execution command (apart from the correctness checks) consists of a few key passages:

- the script file is opened and each line of text (which is a separate command) from first to last is appended to a local-scoped command vector called `scriptReqs` (where "reqs" stands for "requests");
- the order of `scriptReqs` is reversed (for reasons that will be explained later in this section);
- the vector elements from first to last are appended to a global-scoped command vector called `scriptReqsMaster`;
- a boolean variable called `isScript` is set to true to inform that a script-execution command has been successfully interpreted.

If the script-execution command that was successfully interpreted is at "depth 0" (i.e. it is manually entered by the user instead of being read from a script file), `isScript` being set to true triggers entry into the command-selection loop. Such loop writes into a local variable the last element from `scriptReqsMaster` (which is the first command in the script file and thus the first intended to be handled), pops the element from the vector (effectively deleting it from memory and decreasing the vector size by 1) and then performs the regular tokenize-parse-interpret sequence to extract information from the command and transmit it to the controller; the loop keeps iterating as long as the size of `scriptReqsMaster` is bigger than zero, i.e. as long as there are commands - read from script files - that haven't been handled yet.

Independence of the interpretation stage from the depth at which a script-execution command is located is given by the fact that, once the depth-0 script file has been read, if another such command is encountered and consequently a script file is read, its contents will be seamlessly appended to the global-scoped command vector, and thus handled immediately after within the loop.

The reversal of the order of `scriptReqs` is required to streamline updates to the contents of `scriptReqsMaster`, both when adding and when removing elements:

- it makes the first command by order of appearance in the script the last element by vector index, which makes its deletion (after being saved into a local variable, as mentioned before) a straightforward operation via `scriptReqsMaster.pop_back()`;

- if a nested script file is read, the commands it contains are easily added to `scriptReqsMaster` via appending.

An example of nested scripting reaching depth 1 is provided in Figure 2.8 for a more immediate understanding of the algorithm.

## 2.3 Testing the CLI

### 2.3.1 Meaning of software testing

Integrated circuit design places great importance on testing a circuit by means of instantiating it within a test environment called "testbench", then feeding it stimuli (i.e. binary test signals) and recording the signals it outputs.

Testing is as important in software design as it is in hardware design, its purpose being to ensure that software modules work as intended; software testing is performed in a similar fashion to digital circuit testing: a software program (or a single class of said program in the case of "unit testing", a specific type of software testing) is instantiated within a test environment, which may or may not be a class of its own; the test environment then proceeds to stimulate the software under test by calling its functions with test arguments and checking their return values against the expected ones. Each function-call/check pair constitutes a test: it is executed, the tester (often the same developer that wrote the software under test) is notified of its success ("pass") or failure ("fail") and then the test environment moves to the next test with no manual input required of the tester.

### 2.3.2 Performed tests

As the CLI is an elaborate tool appointed with the role of controlling ToPoliNano, testing it constitutes a necessary step to guarantee it correctly enforces the user's commands: to this aim the testing library provided by Qt, `QtTest`, was chosen to build test environments and test cases, in virtue of its integration with Qt's signal/slot system and with Qt's own types like `QString`.

Two different test environments in the form of classes were prepared to respectively test the CLI as a whole - minus the GUI, since GUI testing requires simulating user input like key presses and mouse clicks, which is tricky and produces brittle tests due to minor GUI changes causing test failure even though the underlying functionality hasn't changed - and its `CLIbackend` class.

Both test environments belong to the so-called "white-box testing" category; "white-box testing" is concerned with verifying the internal structure of a program rather than simply examining its functionalities with no knowledge of the source code (a testing category called "black-box testing"), and it does so by feeding to the software test inputs engineered to exercise most (or all) of the different code execution paths available, in order to thoroughly verify its behaviour.

Additionally, a data-driven testing approach was employed, which consists of having the testing class read the test inputs and the expected outputs from file instead of hard-coding them in the source code of the class. This approach has multiple benefits:

- it keeps the testing class source code flexible by moving test inputs and expected outputs (the "variable" components of a test environment) away from the code itself and into a text file, meaning that if the software under test is updated and causes some tests to fail as a consequence it is very easy to fix obsolete inputs or outputs, since it only requires a simple text editor;

- adding new test cases goes from altering the source code and re-compiling to entering inputs and outputs in plain text into the text file.

#### Testing the CLI as a whole

Testing the CLI aimed at verifying that each keyword and its related arguments were correctly identified, interpreted and packaged into a signal.

For each keyword a test data file exists, whose contents are lines of text; each line is a test case and comprises of test input (a command containing the keyword and some of its arguments), expected output (the data extracted from the relevant signal emitted by `CLIcontroller`) and a short description of the test case, isolated from each other by a special character sequence (" ***** ") acting as separator. Figure 2.9 shows the contents of a test data file.

The testing class, `CLItest`, separately tests the behaviour of the CLI in response to commands based on each keyword (except for the "do" keyword, for reasons discussed in Sub-section 2.3.3); in other words, testing is grouped by keyword.

For each keyword, the testing procedure is carried out by two functions called one after the other. Function `CLItest::test[keyword]_data()` is called first, and it performs the following operations:

- it opens the relevant test data file and reads the data line by line;
- for each line, it saves the test input as a `QString` object, while it splits the expected output into the values it is made of (for example, an "export" signal contains 4 values: directory, filename, format, area); if the test input is purposefully wrong, then the expected output is an error message rather than the contents of a `CLIcontroller` signal and it is saved as a single `QString` object; the description of the test case is also saved as a `QString` object;
- it builds a table by creating a column for test inputs and as many columns as the values that make up the expected output, plus a column for expected error messages, then filling rows with the data extracted by each line and labeling every row with the description coming from the same line; each row therefore constitutes a test case and has a number of fields consisting of test input, output values and error message; if the input produces an error message, the output value fields are filled with dummies, and viceversa if the input produces regular output values then the error message field is left blank.

One of such functions is presented in Listing 2.3.

```
void CLItest::testExport_data()
{
    // "export" REQUESTS
    QVector<QString> exportReqs; // This vector will contain all "export" requests extracted from a dedicated file
    // "export" REPLIES
    QVector<QString> exportReplies; // This vector will contain all  "export" replies extracted from a dedicated file
    // "export" LABELS
    QVector<QString> labelsVector; // This vector will contain all  "export" labels extracted from a dedicated file

    QVector< QVector<QString>* > dataVector; // Vector containing pointers to all vectors that will contain test data (test stimuli, expected return values, etc.)
    dataVector.append(&exportReqs);
    dataVector.append(&exportReplies);
    dataVector.append(&labelsVector);
    retrieveTestData(dataVector, "unit-test-files/CLIcontroller/testExport.txt"); // Fill the vectors with test data from the file

    // "export" DATAPIECES
    QVector<QString> exportDirs; // This vector will contain all "export" dirs extracted from "exportReplies" strings
    QVector<QString> exportFilenames; // This vector will contain all "export" filenames extracted from "exportReplies" strings
    QVector<QString> exportFormats; // This vector will contain all "export" formats extracted from "exportReplies" strings
    QVector<QString> exportAreas; // This vector will contain all "export" areas extracted from "exportReplies" strings
    QVector<QString> expErrorMsg; // This vector will contain all "export" expected error messages extracted from          "exportReplies" strings

    // EXTRACT DATAPIECES FROM EACH "exportReplies" STRING
    for(int c = 0; c < exportReplies.size(); c++)
    {
        QString valueSet = exportReplies.at(c); // Read a line from the file - each line is an expected set of values (e.g. "workspace3/Results topLevel png view")
        if(valueSet.contains("error", Qt::CaseInsensitive)) // If no signal will be called because a faulty request was made...
        {
            // ...Save the error message...
            expErrorMsg.append(valueSet);

            // ...Then append dummy values for compliance with the testing system
            exportDirs.append( "dummyDirectory" );
            exportFilenames.append( "dummyFilename" );
            exportFormats.append( "dummyFormat" );
            exportAreas.append( "dummyArea" );
        }
        else
        {
            QStringList valueList = valueSet.split(" ", QString::SkipEmptyParts);
            exportDirs.append( valueList.at(0) );
            exportFilenames.append( valueList.at(1) );
            exportFormats.append( valueList.at(2) );
            exportAreas.append( valueList.at(3) );
            expErrorMsg.append(""); // No error is produced if the request is not faulty
        }
    }

    // CREATION OF TEST COLUMNS AND ROWS
    if( (exportReqs.size() != exportDirs.size() )       ||
        (exportReqs.size() != exportFilenames.size() ) ||
        (exportReqs.size() != exportFormats.size() )   ||
        (exportReqs.size() != exportAreas.size() )   ) // If the two files did not contain the expected amount of replies...
    {
        QString errorQString = "Error: request file and replies
            file for 'export' do not contain the same number of
            lines";
        std::string errorStdString = errorQString.toStdString();
        const char *errorCString = errorStdString.c_str();
        QSKIP( errorCString ); // Print the error and skip
            testing of the "export" functionality
    }
    else
    {
        QTest::addColumn<QString>("export_req");
        QTest::addColumn<QString>("expected_directory");
        QTest::addColumn<QString>("expected_filename");
        QTest::addColumn<QString>("expected_format");
        QTest::addColumn<QString>("expected_area");
        QTest::addColumn<QString>("expected_errormsg");
        for(int i = 0; i < exportReqs.size(); i++)
        {
            std::string stdLabel = labelsVector.at(i).toStdString
                ();
            const char* cLabel = stdLabel.c_str();
            QTest::newRow(cLabel) << exportReqs.at(i)
                << exportDirs.at(i)
                << exportFilenames.at(i)
                << exportFormats.at(i)
                << exportAreas.at(i)
                << expErrorMsg.at(i); // Create a row in the
                    testing table
        }
    }

    return;
}
```

**Listing 2.3**: Function `CLItest::testExport_data()`.

Function `CLItest::test[keyword]()` is called after its `_data()` counterpart, once for every row in the table, and it performs the following operations:

- it instantiates two `QSignalSpy` objects; one will listen for the signal emitted by

CLIcontroller that carries the actual otuput values (to be compared with the expected output values), while the other one will listen for another signal emitted by CLIcontroller that carries the error message; for any given test case, only one of the two signals is emitted;

- it fetches the elements of a row via QFETCH macros;
- it stimulates the CLI by sending it a command (the test input);
- it listens for signals emitted by CLIcontroller; if the received signal is the one carrying the actual output values (the data extracted from the test input by the back-end), each of these values is compared with the corresponding expected output value by means of a QCOMPARE macro, else if the received signal is the one carrying an actual error message, said message is compared with the expected one via the aforementioned QCOMPARE macro.

All comparisons performed with QCOMPARE macros in a call to CLItest::test[keyword]() must be successful; in other words, the actual output values (or the actual error message, if the test case is designed to produce one) must follow the correct (the "expected") behaviour of the program. If they are, then the function will print to standard output a "PASS" mark, followed by its own signature and by the label of the executed test case (i.e. of the row); conversely, if at least one comparison is unsuccessful the function will print to standard output a "FAIL" mark, then signature and label and finally a failure message notifying the tester of the unsuccessful comparison and showing the actual and expected values which triggered the failure for not coinciding. One of the CLItest::test[keyword]() functions is presented in Listing 2.4.

```
void CLItest::testExport()
{
    QSignalSpy spyExport( controllerUnderTest, SIGNAL(
       SignalFrontendExport(const QString, const QString, const
       QString, const QString) ) ); // Create a QSignalSpy that
       follows "SignalFrontendExport(const QString)"
    QSignalSpy spySignalToLog( controllerUnderTest, SIGNAL(
       SignalSendBackendOutput(const QString, Logger::LogLevel)))
       ; // Create a QSignalSpy that intercepts backend messages
       directed to LogPannel

    // FETCH THE TEST REPLIES
    QFETCH(QString, export_req);
    QFETCH(QString, expected_directory);
    QFETCH(QString, expected_filename);
    QFETCH(QString, expected_format);
    QFETCH(QString, expected_area);
    QFETCH(QString, expected_errormsg);

    // STIMULATE THE UNIT UNDER TEST
    controllerUnderTest->SlotHandleRequest(export_req); // Handle an "export" request extracted from a dedicated file

    spyExport.wait(1000);
    if(spyExport.count() == 1) // If the signal was called, analyze it
    {
        QList<QVariant> arguments = spyExport.takeFirst(); // Take the arguments of the first emitted signal
        QString dirArg = arguments.at(0).toString(); // Convert the "directory" argument of "SignalFrontendExport()" back to QString
        QString filenameArg = arguments.at(1).toString(); // Convert the "filenames" argument of "SignalFrontendExport()" back to QString
        QString formatArg = arguments.at(2).toString(); // Convert the "format" argument of "SignalFrontendExport()" back to QString
        QString areaArg = arguments.at(3).toString(); // Convert the "area" argument of "SignalFrontendExport()" back to QString

        QCOMPARE(dirArg,      expected_directory);
        QCOMPARE(filenameArg, expected_filename);
        QCOMPARE(formatArg,   expected_format);
        QCOMPARE(areaArg,     expected_area);
    }
    else checkErrorMsg(spySignalToLog, expected_errormsg); // 
       Perform a QCOMPARE between the error message carried by 
       the signal and the expected error message

    return;
}
```

**Listing 2.4**: Function `CLItest::testExport()`.

### Testing CLIbackend

While testing the CLI as a whole ensures that the program correctly responds to human interaction, testing individual classes (i.e. performing unit testing) provides more in-depth insight on software behaviour and helps spot elusive bugs. In the case of `CLIbackend` however, since the `TCLAP::CmdLine` class relies by design on the `TCLAP::StdOutput` class to output parsing errors, it's worth noting that the testing code for `CLIbackend` does not strictly belong to the unit testing category.

The focus of the testing was the function `CLIbackend::tokenizeAndParseCmd()` which, as the name implies, is responsible for the first two stages of extraction: tokenization and parsing. This function returns a boolean value which is true if tokenization and parsing were successful, while it is false if either stage encountered an issue. If the value is true, it triggers the last stage consisting of interpretation. As mentioned in Subsection 2.2.4, the parser does not know what the argument values should look like apart from their type, thus if some arguments have incorrect values that still fit the data type the function `CLIbackend::tokenizeAndParseCmd()` returns true. This behaviour is expected, as another function down the pipeline, responsible for interpretation, raises an error later on.

The testing procedure is the same previously described in Subsection 2.3.2: carried out by two functions called one after the other, `CLIbackendTest::testTokenizeAndParseCmd_data()` and `CLIbackendTest::testTokenizeAndParseCmd()`, with `CLIbackendTest` being the testing class. A few test cases are shown in Figure 2.10.

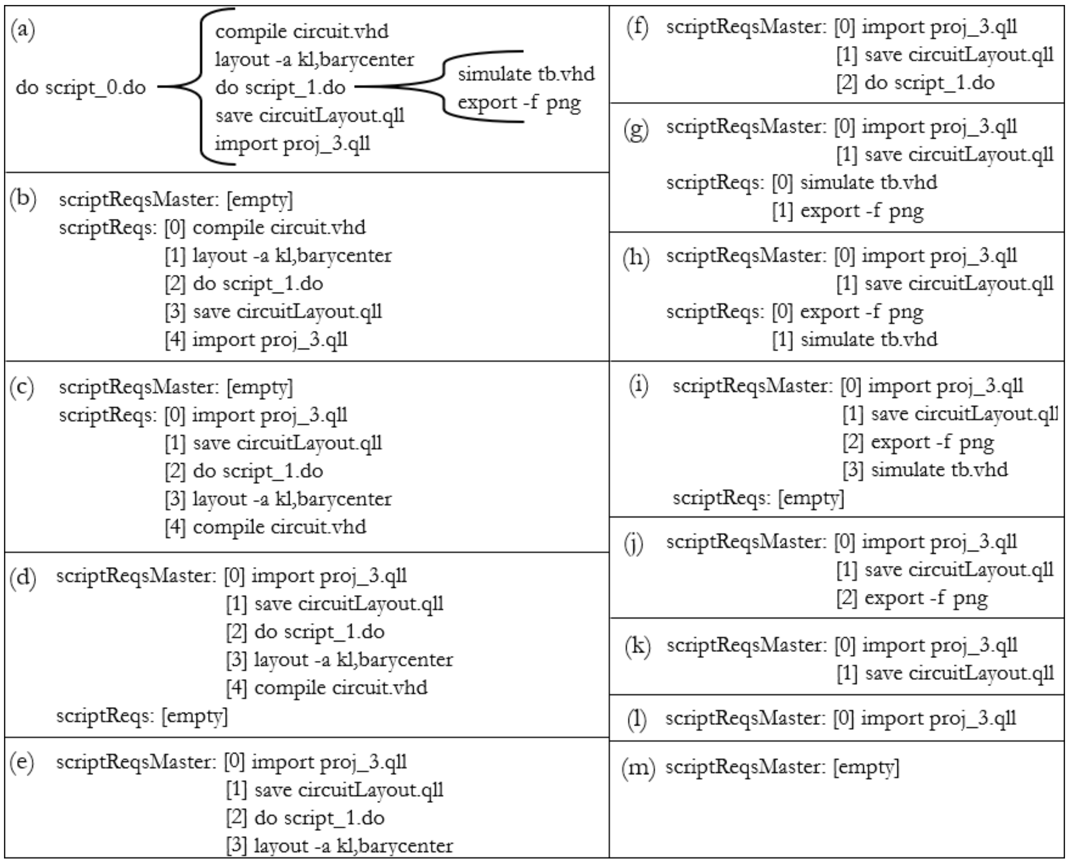
### 2.3.3 Final words on testing

The testing technique adopted to test the CLI is called "white-box testing": such technique provides an excellent way of easily exercising most execution paths in the program code; however, an even more efficient approach would have been TDD, which consists of writing enough testing code to make one test, then writing enough program code to make the test pass, then starting a new cycle by expanding the testing code and then the program code, and so on; for this reason, TDD was selected as the approach of choice for developing the script preprocessor described in Chapter 3.

Additionally, this being a thesis work there were time constraints to observe that limited the extent to which the CLI could be tested:

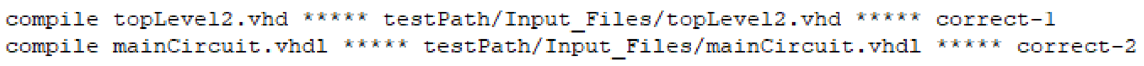
- as anticipated in Sub-section 2.3.2, commands based on the "do" keyword were not tested; the reason for this lies in "do" being a "meta"-command of sorts, which lets the user execute a sequence of commands saved on file simply by calling "do file.do"; therefore, testing "do"-keyword commands means verifying that the CLI handles the commands stored in a script file in the correct order; verifying this requires that the testing class listens for and extracts data from several different transmitted signals forwarded by the controller, a much more time-consuming procedure to code when compared to the others;

- testing of `CLIbackend` was limited to two-thirds of its main functionalities, embodied by its `tokenizeAndParseCmd()` function, rather than expanded to also cover the interpretation function (`interpretCmd()`) and utility functions responsible for "behind-the-scenes" work like instantiating the `Arg` objects.



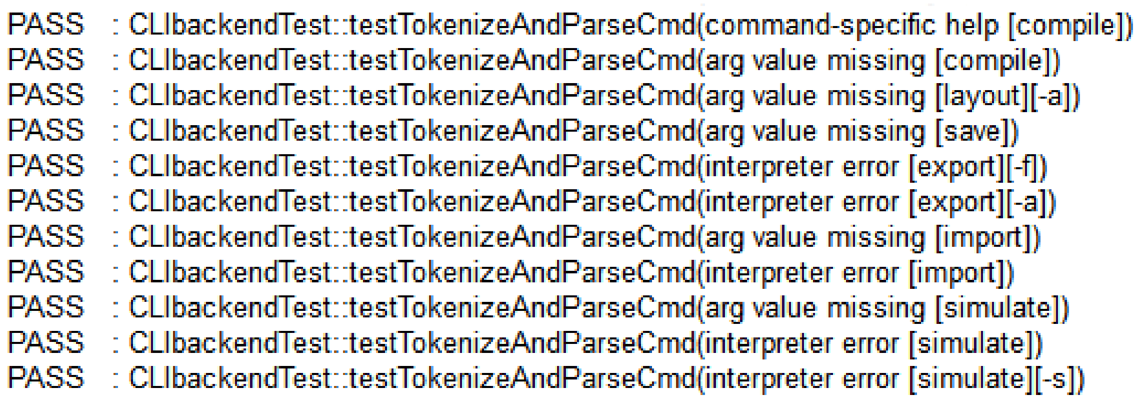
```
(a)                   compile circuit.vhd
                      layout -a kl,barycenter
    do script_0.do    do script_1.do            simulate tb.vhd
                      save circuitLayout.qll    export -f png
                      import proj_3.qll

(b) scriptReqsMaster: [empty]
    scriptReqs: [0] compile circuit.vhd
                [1] layout -a kl,barycenter
                [2] do script_1.do
                [3] save circuitLayout.qll
                [4] import proj_3.qll

(c) scriptReqsMaster: [empty]
    scriptReqs: [0] import proj_3.qll
                [1] save circuitLayout.qll
                [2] do script_1.do
                [3] layout -a kl,barycenter
                [4] compile circuit.vhd

(d) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
                      [2] do script_1.do
                      [3] layout -a kl,barycenter
                      [4] compile circuit.vhd
    scriptReqs: [empty]

(e) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
                      [2] do script_1.do
                      [3] layout -a kl,barycenter

(f) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
                      [2] do script_1.do

(g) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
    scriptReqs: [0] simulate tb.vhd
                [1] export -f png

(h) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
    scriptReqs: [0] export -f png
                [1] simulate tb.vhd

(i) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
                      [2] export -f png
                      [3] simulate tb.vhd
    scriptReqs: [empty]

(j) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll
                      [2] export -f png

(k) scriptReqsMaster: [0] import proj_3.qll
                      [1] save circuitLayout.qll

(l) scriptReqsMaster: [0] import proj_3.qll

(m) scriptReqsMaster: [empty]
```

**Figure 2.8**: a) two script files are involved in this example: "script_0.do" and "script_1.do";
b) interpretation of "do script_0.do" yields 5 commands, saved into `scriptReqs`;
c) the order of `scriptReqs` is reversed;
d) the contents of `scriptReqs` are appended to `scriptReqsMaster`;
e) the loop is entered; the last element of `scriptReqsMaster`, "compile ...", is interpreted and popped from `scriptReqsMaster`;
f) the last element of `scriptReqsMaster`, "layout ...", is interpreted and popped from `scriptReqsMaster`;
g) the last element of `scriptReqsMaster`, "do script_1.do", is interpreted and popped from `scriptReqsMaster`; it yields 2 commands, saved into `scriptReqs`;
h) the order of `scriptReqs` is reversed;
i) the contents of `scriptReqs` are appended to `scriptReqsMaster`;
j) the last element of `scriptReqsMaster`, "simulate ...", is interpreted and popped from `scriptReqsMaster`;
k) the last element of `scriptReqsMaster`, "export ...", is interpreted and popped from `scriptReqsMaster`;
l) the last element of `scriptReqsMaster`, "save ...", is interpreted and popped from `scriptReqsMaster`;
m) the last element of `scriptReqsMaster`, "import ...", is interpreted and popped from `scriptReqsMaster`, which is now an empty vector; the loop is exited.



```
compile topLevel2.vhd ***** testPath/Input_Files/topLevel2.vhd ***** correct-1
compile mainCircuit.vhdl ***** testPath/Input_Files/mainCircuit.vhdl ***** correct-2
```

**Figure 2.9**: Test data for the "compile" keyword; each line contains in order the test input, the expected output and the test case description



```
PASS   : CLIbackendTest::testTokenizeAndParseCmd(command-specific help [compile])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [compile])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [layout][-a])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [save])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [export][-f])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [export][-a])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [import])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [import])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [simulate])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [simulate])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [simulate][-s])
```

**Figure 2.10**: Results of test cases from the CLIbackend testing procedure

# Chapter 3

# Script preprocessor

By itself, the CLI is able to understand a script file containing a plain list of commands. While such a capability is enough to automate tasks, it does not lend itself to more advanced scripting that may include conditional statements or token replacements.

In order to enable the user to write complex scripts that give more granular control over the flow of execution, a new general-purpose preprocessor[1] was developed for use by the CLI.

## 3.1 Functional overview

The preprocessor makes available to the user some key features of scripting languages (like TCL) with a simple syntax for easier writing:

- integer variables used in boolean conditions (>, >=, <, <=, ==) and embeddable in filenames for later replacement with their value;
- conditional constructs: if, if/else, if/elif/else, if/elif/elif/.../else;
- loops: for, while.

Conditional constructs and loops are freely nestable to achieve the desired complexity.

When the CLI opens a script file, it hands the contents to the preprocessor as a `QStringList` (i.e. a vector of `QString`); the preprocessor performs a variety of operations on it (which are described in detail in Section 3.3) based on the constructs employed within the script, while leaving the CLI commands untouched save for any token replacements (keep in mind that a preprocessor often has no knowledge or interest in the underlying language; the C preprocessor is an example of such behaviour). The end result, returned as a `QStringList`, is a plain list of commands mirroring what

[1]https://en.wikipedia.org/wiki/General-purpose_macro_processor

a live interpretation of the script - which evaluates conditions and executes branches and loops - would look like; in other words, the preprocessor "pre-digests" the script, producing in output a simple list compatible with the capabilities of the CLI, which therefore needs no modifications to its own logic. Figure 3.1 shows the overall procedure.

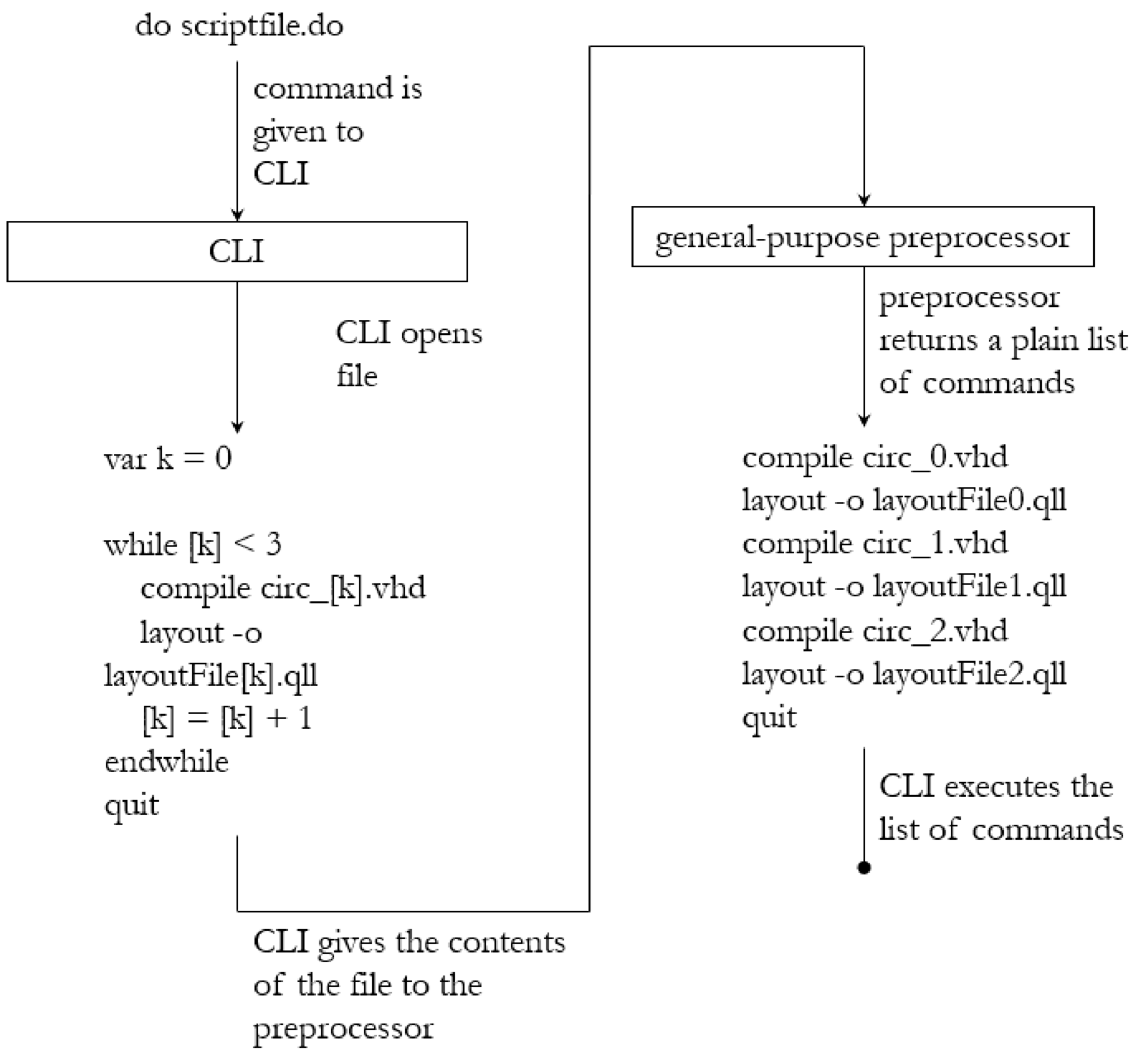

do scriptfile.do
command is given to CLI
CLI
CLI opens file
var k = 0
while [k] < 3
compile circ_[k].vhd
layout -o layoutFile[k].qll
[k] = [k] + 1
endwhile
quit
CLI gives the contents of the file to the preprocessor
general-purpose preprocessor
preprocessor returns a plain list of commands
compile circ_0.vhd
layout -o layoutFile0.qll
compile circ_1.vhd
layout -o layoutFile1.qll
compile circ_2.vhd
layout -o layoutFile2.qll
quit
CLI executes the list of commands

**Figure 3.1**: The CLI opens a script file and gives the contents to the preprocessor; the latter returns the preprocessed list of commands; the CLI executes said list.

## 3.2 Technical overview

Under the hood, the preprocessor performs three main operations:

- detection: it detects integer variable definitions (e.g. "var num = 0") and creates symbols (i.e. tokens) from them, each with the name and value written in the definition;
- tree creation: it scans the script contents (received from the CLI) and builds a tree of nodes from them;
- tree traversal: it performs a depth-first, pre-order traversal[2] of the tree, which produces the simplified script file.

Any node in the tree represents one of the following:

- a list of related if/elif/else branches;
- a while loop;
- a for loop;
- an assignment to an integer variable (except for variable definitions);
- a CLI command

with the sole exception of the special root node, called "Head node", which the tree is grown from.

As the contents are parsed and turned into nodes, if a newly created node A represents a list of conditional branches or a loop, the block of statements in its scope is scanned first before moving on, and the resulting nodes are appended to the aforementioned node A as "child nodes"; in other words, scoped blocks of statements are recursively parsed, generating a chain of nodes with parent/child relationships.

The preprocessor is built as a hierarchy of classes, with the simpler ones employed by the more complex ones; all of them are declared within a common namespace, called `ScriptExt` (which stands for "Script Extender").

[2]https://en.wikipedia.org/wiki/Tree_traversal#Depth-first_search

## 3.3 Designing the preprocessor

### 3.3.1 namespace ScriptExt

The namespace serves to gather all the classes that make up the preprocessor in order to prevent name conflicts or ambiguities with other, unrelated classes. This is a necessary precaution due to the complexity of the source code of ToPoliNano, and also provides a clearer understanding of the composition of the preprocessor code itself. The namespace also hosts a `QStringList` object called "collectiveLog" which, as the name implies, collects all log messages generated by the various parts of the preprocessor as it processes a script. Such log is not printed on-screen nor on console, acting more as a diagnostic tool that collects information on the activities performed by the preprocessor; the user is still able to freely consult the collectiveLog if they wish to do so. The namespace is shown in Listing 3.1.

```
#ifndef SCRIPTEXT_NAMESPACE_H
#define SCRIPTEXT_NAMESPACE_H

#include <QString>
#include <QStringList>

namespace ScriptExt
{
    class Node;
    class Symbol;
    class SymbolManager;
    class Tree;
    class Toolkit;

    extern QStringList collectiveLog; // Global variable used as log by all ScriptExt classes - definition is in scriptext_namespace.cpp
    inline static const QStringList& getCollectiveLog() { return collectiveLog; }
    inline static const QString getCollectiveLogAsString() { return collectiveLog.join("\n"); }
};

```

```
#endif // SCRIPTEXT_NAMESPACE_H
```

**Listing 3.1**: The ScriptExt namespace.

### 3.3.2 class Node

Class `Node` is the fundamental element of the tree built by the preprocessor. Each `Node` object that is instantiated represents a different piece of content from the script. As anticipated in Section 3.2, a node can be one of the following types: Command, Assignment, While, For, If, Head (with the root node being the only "Head"-type node in the tree). The type determines what data a node holds and how the node is traversed by the preprocessor in the traversal stage. Apart from the type, each node possesses a "node statement", a string that defines their purpose (e.g. a CLI command or a "while" loop). Depending on the type, a node may also hold a "block", i.e. the statements within its scope (for example, a "while" loop may contain a block consisting of three CLI commands), in which case it also has child nodes. More specifically:

- a Command node only holds a statement, which in its case is a CLI command;
- an Assignment node, similarly, only holds a statement, which - as the name "assignment" implies - is an assignment to a variable (e.g. [i] = [i] + 1);
- a While node holds a statement (e.g. "while [num] < 3"), a block (the scoped statements) and the child nodes corresponding to the scoped statements in the block;
- a For node has the same attributes of a While node, but the statement is different as it holds information about the initial assignment to the loop variable, the condition to evaluate at each iteration and the variable increment or decrement at the end of each iteration (e.g. "for [k] = 0; [k] <= 4; [k] = [k] + 1");
- an If node possesses multiple statements, with the top-priority one being the if-statement (e.g. "if [i] < 2") followed by any number of elif-statements (e.g. "elif [i] >= 2 and [i] < 4") - which can be none - and, if present, an else-statement; consequently, the node holds as many blocks as the aforementioned conditional

statements, plus a block representing the else-scoped statements (if present), and as many groups of child nodes as the blocks;

- a Head node possesses a placeholder statement and a block consisting of the script contents, which are assigned to it by the Tree constructor when creating a Tree object (which is described in Sub-section 3.3.6).

### 3.3.3 class Symbol

Class `Symbol` is the internal representation that the preprocessor employs for the integer variables defined by the user within a script. A Symbol has two attributes: a name and an integer value; the latter can be incremented, decremented or set to an arbitrary value.

### 3.3.4 class SymbolManager

Class `SymbolManager` is in charge of creating and managing `Symbol` objects, including updating their values when required; additionally, it is tasked with replacing symbol names (encapsulated in square brackets) occurring in strings with their current values; in other words, it is tasked with assignment evaluation and token replacement. Examples for each of these two functionalities are illustrated in the following paragraphs.

#### Example of symbol replacement in a string

In this example, `SymbolManager` already holds a symbol with name "k" and value "0".

If the function `SymbolManager::replaceSymbol()` is called with string "compile circuit[k].vhd" as argument, `SymbolManager` will detect the encapsulated symbol name "[k]", search the list of existing symbols for one with name "k" and finally replace "[k]" with its current value 0 within the string. Thus, the return value will be string "compile circuit0.vhd".

#### Example of symbol assignment evaluation

In this example, `SymbolManager` already holds a symbol with name "num" and value "2".

If the function `SymbolManager::symbolAssignment()` is called with string "[num] = [num] + 1" as argument:

- `SymbolManager` splits the string into left-hand-side substring (abbreviated to "lhs") and right-hand-side substring (abbreviated to "rhs");
- in the rhs, `SymbolManager` detects the encapsulated symbol name "[num]", searches the list of existing symbols for one with name "num" and finally replaces "[num]" with its current value 2;
- the resulting rhs "2 + 1" is evaluated as an arithmetic operation, yielding the integer value 3;
- in the lhs, `SymbolManager` detects the encapsulated symbol name "[num]", searches the list of existing symbols for one with name "num" and sets its value to the integer 3 obtained from the arithmetic operation.

As a result, symbol "num" is incremented by 1, from a value of 2 to a value of 3.

### 3.3.5 class Toolkit

Class `Toolkit` - as the name implies - has a number of functions that serve as "tools" for the preprocessor to perform simple but key operations with: trimming leading and trailing whitespace from a string (including tabs and newlines), checking if a string starts or ends with the contents of another string (e.g. checking if it starts with "while" or "for"), removing a certain amount of characters from the left side or right side of a string, evaluating an inequation (e.g. "5 > 3", which resolves to true) and evaluating an addition or subtraction (e.g. "4 - 1", the result of which is 3).

### 3.3.6 class Tree

Class `Tree` is tasked with executing the three stages of preprocessing described at the beginning of Section 3.2: detection of variable definitions, tree creation, tree traversal.

### Detection of variable definitions

In the first stage, `Tree` scans the script contents looking for variable definitions, i.e. lines with the structure "var [name] = [value]" (e.g. "var k = 0"); every such line prompts `Tree` to invoke `SymbolManager`, which creates a Symbol object with the name and value contained in the line and adds it to the list of existing symbols. Variable definitions are found at the very beginning of the script, therefore once a line which is not a definition is found `Tree` concludes the variable-detection stage and starts the next one: tree creation.

### Tree creation

As Section 3.2 briefly mentioned, tree creation consists of scanning the script contents (except for the variable definitions, which are parsed in the previous stage) and making nodes out of them.

The tree creation stage is handled by function `Tree::bloom()`, which takes a node as argument (and refers to it as "parent"); in particular, the first call to `bloom()` has the Head node as argument. The function checks whether "parent" is a node of type Command or Assignment, in which case it returns immediately due to nodes of these two types possessing no scoped block to parse. If "parent" is of neither type, it then checks whether its type is While, If, For or Head. If the type is While, For or Head, then `bloom()` fetches the scoped block from "parent" and scans it:

- if a CLI command is found, a Command node is created with the command as "node statement";
- if an assignment to variable is found, an Assignment node is created with the assignment as "node statement";
- if a while-statement is found then all lines between such while-statement and the while-loop termination statement "endwhile", i.e. all scoped lines, are packaged into a block; then a While node is created with the while-statement as "node statement" and the block is assigned to it;
- if a for-statement is found then, just like for the While node, all lines between such for-statement and the for-loop termination statement "endfor" are packaged into

a block; then a For node is created with the for-statement as "node statement" and the block is assigned to it;

- if an if-statement is found then all lines between such if-statement and the if-construct termination statement "endif" are packaged into a block; then (if present) all elif-statements within said block are appended to a list, and the block itself is split into the if-scoped sub-block, the elif-scoped sub-block(s) (if present) and the else-scoped sub-block (if present); finally, an If node is created with the if-statement as "node statement" and both the list of elif-statements and the sub-blocks obtained from the splitting of the initial block are assigned to it.

Whatever the type, once a node is created it is added as child node to "parent"; immediately after that, `bloom()` is recursively called with the new node as argument (i.e. as "parent"), thus going "one level deeper" into the script.

When the entirety of the script contents has been successfully scanned, the tree is complete; thus the next (and last) stage may begin: tree traversal.

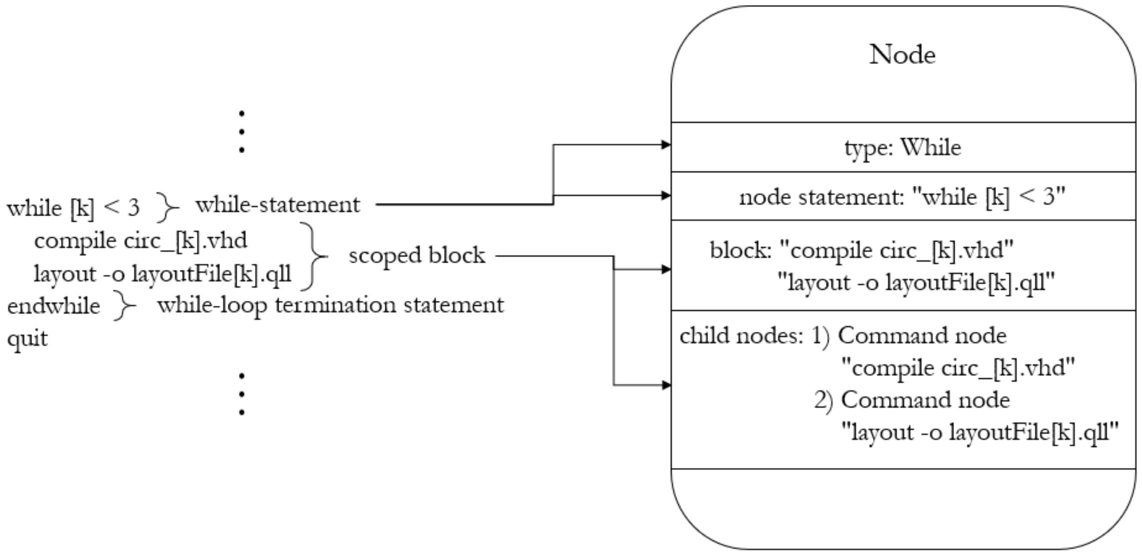

while [k] < 3
while-statement
compile circ_[k].vhd
layout -o layoutFile[k].qll
scoped block
endwhile
while-loop termination statement
quit
Node
type: While
node statement: "while [k] < 3"
block: "compile circ_[k].vhd"
"layout -o layoutFile[k].qll"
child nodes: 1) Command node
"compile circ_[k].vhd"
2) Command node
"layout -o layoutFile[k].qll"

**Figure 3.2**: Naming conventions used for the attributes of class Node.

### Tree traversal

The tree built by `bloom()` is traversed by means of the `Tree::traverse()` function, which - like `bloom()` - takes a node as argument (referred to as "parent" in this case too) and is given the Head node when it is called for the first time.

Each node used as "parent" is traversed differently depending on their type:

- if "parent" is a Command node, its "node statement" is fetched, any symbols embedded in it are replaced by means of `SymbolManager::replaceSymbol()` and the resulting string is appended to a `QStringList` object called "traversalTrace", which constitutes (once completed) the preprocessed plain list of commands;

- if "parent" is an Assignment node, its "node statement" is fed into `SymbolManager::symbolAssignment()`, which evaluates the string and increments/decrements/changes the value of the symbol subjected to the assignment;

- if "parent" is a While node, its "node statement" is evaluated by `Tree::evaluateConditionalStatemt()`, which extracts the condition(s) from the statement itself (e.g. "[i] < 4 and [j] == 0" is taken from "while [i] < 4 and [j] == 0"), evaluates it/them and returns the boolean result; if it is true, then `traverse()` is recursively called with each of the child nodes of "parent" as arguments; once all child nodes have been traversed, the "node statement" of "parent" is re-evaluated and, if the result is true, the cycle starts over;

- if "parent" is a For node, the traversal procedure is similar to that of the While node; however, since the "node statement" is made of three pieces - the initial assignment, the condition and the variable increment or decrement - some key differences are present:

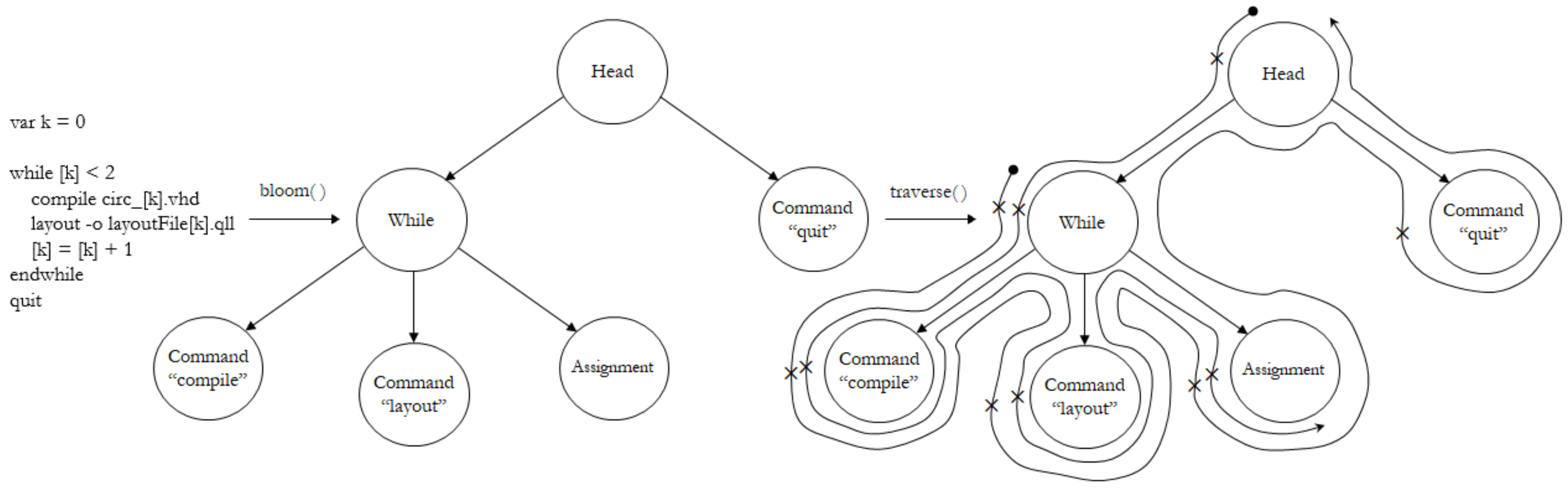
  - before the first evaluation of the condition by `Tree::evaluateConditionalStatemt()`, the initial assignment is performed;
  - every time all child nodes have been traversed, the variable increment/decrement is performed;

  apart from said differences, the cycle is the same;

- if "parent" is an If node, its "node statement" (the if-statement) is evaluated by `Tree::evaluateConditionalStatemt()`; if the result is true, then `traverse()` is recursively called with each of the child nodes related to the if-scoped sub-block as argument; if, instead, the result is false, then the elif-statements from the list (if present) are evaluated one after the other; as soon as one is found to be true, `traverse()` is recursively called using as argument each of the child nodes related to the sub-block of that elif-statement; if none of the elif-statements evaluate to true or if no elif-statement is present at all, `traverse()` is recursively called with each of the child nodes related to the else-scoped sub-block (if present) as argument.

- if "parent" is the Head node, its traversal is almost the same as that of a While node, with the key difference being that it only occurs once because there is no "node statement" to evaluate.

Once the tree has been traversed in its entirety, the preprocessed script (saved as "traversalTrace" within `Tree`) may be fetched via the `Tree::getTraversalTrace()` function or via the `Tree::getReverseTrace()` function (which returns a reversed version of "traversalTrace").



```
var k = 0

while [k] < 2
    compile circ_[k].vhd
    layout -o layoutFile[k].qll
    [k] = [k] + 1
endwhile
quit
```

**Figure 3.3**: The preprocessor builds a tree of nodes from a script, then traverses it following the conditions within the script itself.

### 3.3.7 Integration with the CLI

The preprocessor encapsulates independent logic from the CLI, thus integrating the former into the latter is a straightforward operation: in the script-execution algorithm (illustrated in Sub-section 2.2.4) only one small alteration is made: once the script file is opened and each line of text from first to last is appended to the local `scriptReqs` vector, `scriptReqs` is fed into a `Tree` object, which preprocesses it; after that, in place of the reversed `scriptReqs` vector the reversed version of the preprocessed script (fetched via `Tree::getReverseTrace()`) is appended to the global `scriptReqsMaster` vector. The rest of the CLI algorithm is left intact.

## 3.4 Testing the preprocessor

Since the preprocessor had to be built from scratch and had to perform relatively elaborate operations like token detection and replacement and execution of recursive algorithms, Test-Driven Development (TDD) was chosen as the development technique for this task. TDD consists of cycles, each of which starts by writing enough testing code to make one test and is followed by writing enough program code to make that test pass, then a new cycle is started.

As the preprocessor makes no use of Qt's signal/slot system, a testing framework different from QtTest could be employed: for its ease of use and macro-based approach, the Catch2 framework[3] was chosen.

### 3.4.1 Testing class Node

Class `Node` produces the building blocks of a tree, therefore testing focused on making sure that the class constructors created `Node` objects correctly and that addition of child nodes to nodes of each type worked as it should; If-type nodes, in particular, due to their complexity (as they are used to represent entire if/elif/.../else constructs, if needed) required special care during testing.

[3]https://github.com/catchorg/Catch2

### 3.4.2 Testing class Symbol

Class `Symbol` has a very simple structure and its member functions are all inline; tests were therefore comparably uncomplicated as they only had to verify that these functions behaved properly.

### 3.4.3 Testing class SymbolManager

Testing of class `SymbolManager` focused on verifying that its management of `Symbol` objects had no issues and that its functions, which have the key tasks of performing token replacement (where the "tokens" are symbol names), evaluation of an assignment and creation of a `Symbol` object from a definition string (e.g. "var count = 0") return the expected results.

### 3.4.4 Testing class Toolkit

Class `Toolkit` handles many smaller tasks, each of which expects a large variety of inputs (e.g. `Toolkit::trim()` may receive a string with leading tabs or whitespace or even both, or a string with a trailing newline); testing, therefore, was concerned with exhaustively covering such variety. An example of testing code related to this class is shown in Listing 3.2.

```
SECTION( "Trimming strings with leading and/or trailing whitespace" )
{
    QString statement("       layout -f 3");

    REQUIRE(Toolkit::trim(statement).toStdString() == "layout -f 3" );

    statement = "simulate tb.vhd   \n        ";

    REQUIRE( Toolkit::trim(statement).toStdString() == "simulate tb.vhd" );

    statement = "    quit  ";

    REQUIRE( Toolkit::trim(statement).toStdString() == "quit" );

    statement = "\t\tcompile 01.vhd\n\t\t";

    REQUIRE( Toolkit::trim(statement).toStdString() == "compile
        01.vhd" );
}
```

**Listing 3.2**: Testing code for function `Toolkit::trim()`.

### 3.4.5 Testing class Tree

Class `Tree` is by far the most complex of all the classes that together make up the preprocessor, due to the algorithms employed by its functions `Tree::evaluateConditionalStatemt()`, `Tree::bloom()` and `Tree::traverse()`. Extensive testing was performed to ensure that:

- all kinds of conditional statements were evaluated without mistakes;
- scripts with increasing complexity, from plain lists of commands with neither if/elif/else constructs nor loops to scripts with nested loops were correctly translated into trees;
- scripts with increasing complexity were traversed as expected, obeying conditions and applying value changes to variables.

Due to the inherent intricacy of recursive algorithms, which makes it difficult to track their progress and depth at any given time, several logging instructions were placed within `bloom()` and `traverse()` which write messages containing diagnostic information into collectiveLog (in the preprocessor namespace). This built-in debugging feature allowed to verify, for each test script, not only that the results were as expected but also that each and every step of execution of the recursive algorithms was correct, by comparing the actual log messages written by `bloom()` and `traverse()` into collectiveLog with the ones expected from a faultless execution.

# Chapter 4

# No-gui mode

With the addition of a preprocessor the CLI had become a powerful tool, but the user was still forced to interact with it from the GUI of ToPoliNano. Such inflexibility would partly defeat the purpose of offering control of the program via text commands, therefore a new "no-GUI" mode was developed for ToPoliNano, in order to let the user access the program directly via CLI without also having to load the GUI.

## 4.1 Functional overview

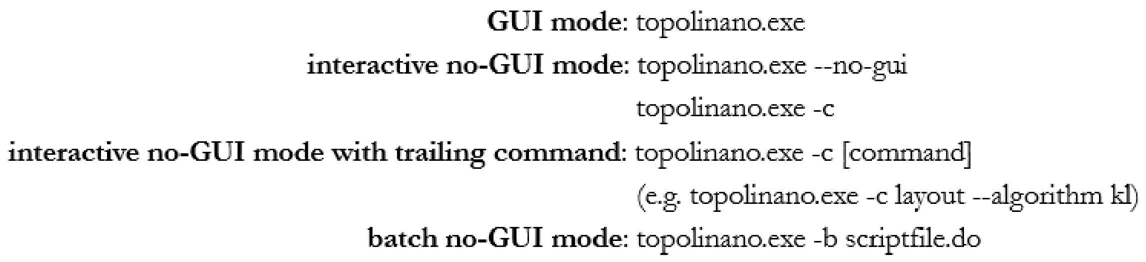
The no-GUI mode allows the user to launch from a terminal window ToPoliNano without GUI in three different ways:

- interactive no-GUI mode, which loads ToPoliNano into the terminal window it is launched from, where the user can write text commands for it to execute;
- interactive no-GUI mode with trailing command, which lets the user launch ToPoliNano and have it automatically execute a text command;
- batch no-GUI mode, which loads ToPoliNano, makes it automatically execute a script and then closes it once it is done.

The launch flags related to each flavour of no-GUI mode are shown in Figure 4.1.

## 4.2 Designing the no-GUI mode

In order to separate the logic for the no-GUI mode from the pre-existing logic in charge of the GUI mode, a new class called `MainCore` was developed, designed to mirror the class that takes care of the GUI mode, called `MainWindow`. Depending



| | |
|---|---|
| **GUI mode**: | topolinano.exe |
| **interactive no-GUI mode**: | topolinano.exe --no-gui |
| | topolinano.exe -c |
| **interactive no-GUI mode with trailing command**: | topolinano.exe -c [command] |
| | (e.g. topolinano.exe -c layout --algorithm kl) |
| **batch no-GUI mode**: | topolinano.exe -b scriptfile.do |

**Figure 4.1**: The various flavours of no-GUI mode and their related launch flags.

on which command-line flags ToPoliNano is launched with (if any), either of the two aforementioned classes will be employed.

### 4.2.1 Replacing the GUI elements of the CLI

In no-GUI mode the GUI elements of the CLI (i.e. the single-line text editor and the log box that displays output, both shown in Figure 2.5) are unavailable due to the GUI itself not being used. The terminal, where ToPoliNano is loaded in no-GUI mode, provides both a means for user input and a means for program output; however, dedicated code had to be written in order to correctly wire ToPoliNano to the terminal:

- as the user will be writing commands from the terminal itself rather than from GUI, a new `MainCore::SlotQueryUser()` function was created, tasked with collecting user input and then forwarding it for interpretation;

- all program output has been routed through a new function, called `MainCore::SlotPrint()`, which in turn outputs messages directly to terminal (after prepending the current date and time to them, in order to maintain consistency with the timestamping performed by the GUI).

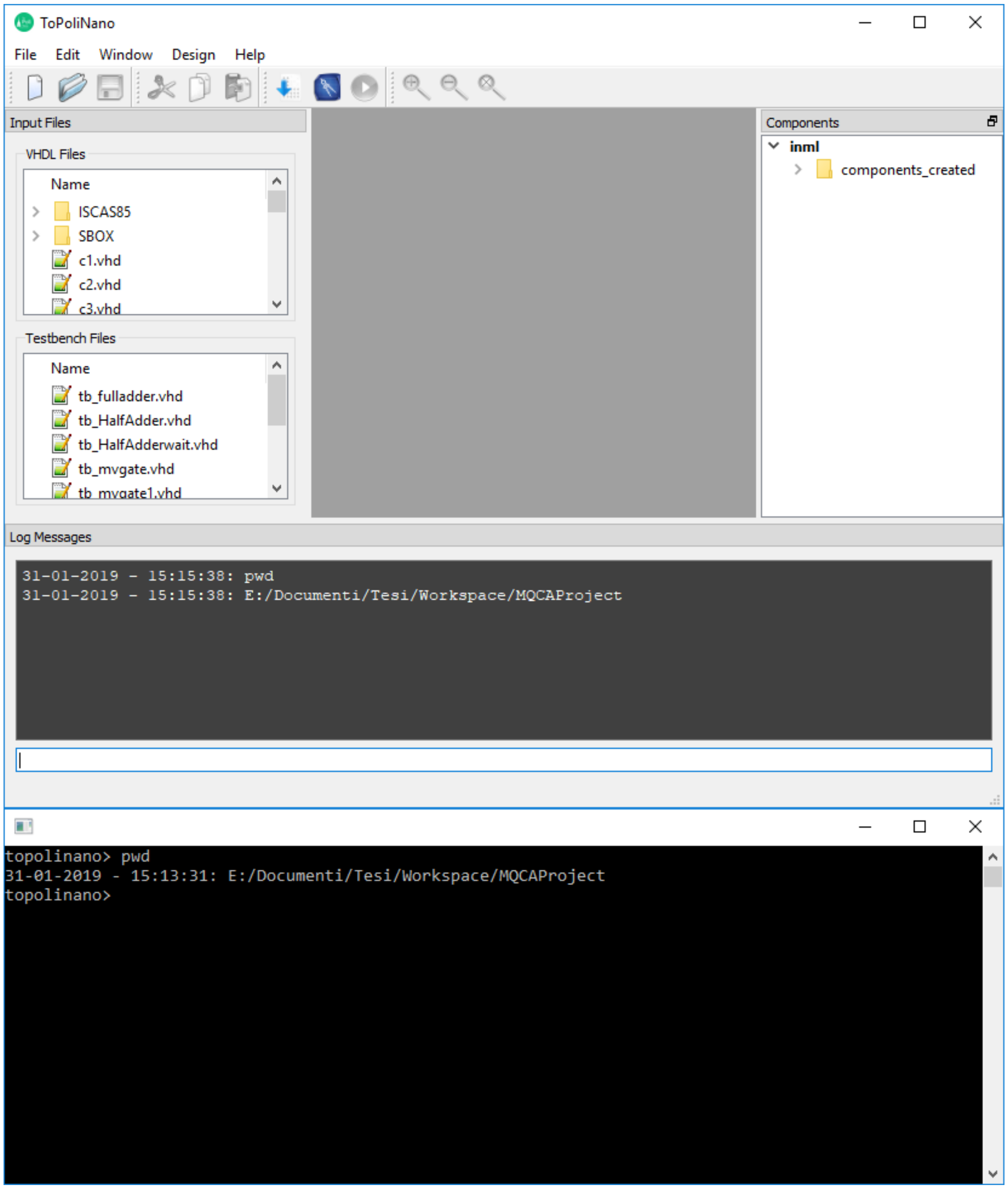
### 4.2.2 Adding support for terminal commands

Once ToPoliNano is loaded into the terminal, the latter becomes unable to recognize traditional terminal commands like "cd", "ls" and "pwd" due to hosting another program. These are valuable commands that enable the user to navigate files and directories without resorting to a GUI file explorer, therefore the no-GUI mode was given the ability to execute them itself.

A dedicated class, `SysCmdProcessor`, takes care of handling terminal commands by tapping into Qt's `QProcess` class; encapsulating this logic into a class separate from `MainCore` makes it possible to offer this helpful capability in GUI mode too, meaning that the user is able to execute terminal commands not only in no-GUI mode, but also from the GUI of ToPoliNano.

There is one major difference in how the output of terminal commands is handled, depending on whether the user is interacting with the no-GUI mode or with the GUI: in the first case, a terminal command is run by calling `QProcess::execute()`, a static function which outputs to terminal by default; in the second case, instead, a dedicated `QProcess` object is instantiated, it is given a terminal command to run and once it is done its output is fetched, converted to `QString` and finally printed to the log box (the `QTextEdit` object mentioned in Sub-section 2.2.2).

Figure 4.2 shows a call to terminal command "pwd" in both cases.



```
31-01-2019 - 15:15:38: pwd
31-01-2019 - 15:15:38: E:/Documenti/Tesi/Workspace/MQCAProject
```

```
topolinano> pwd
31-01-2019 - 15:13:31: E:/Documenti/Tesi/Workspace/MQCAProject
topolinano>
```

**Figure 4.2:** Calling "pwd" from GUI and in no-GUI mode.

# Appendix A

# CLI user manual

In order to give the user all the necessary information to use the CLI to its fullest potential and avoid making mistakes, a dedicated user manual was prepared. After an introduction to the text commands supported by ToPoliNano via the CLI itself, the user will learn about the program workflow and how to write scripts of increasing complexity, from simple lists of commands to nested loops.

## A.1 Supported text commands

Text commands in ToPoliNano follow the naming conventions shown in Figure A.1.

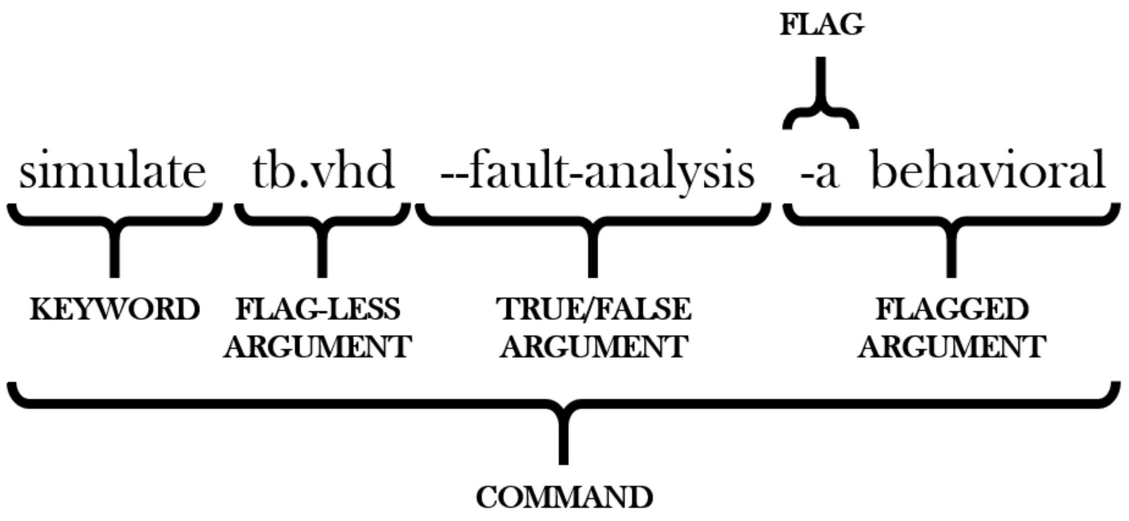


**Figure A.1**: Naming conventions used in this text for the elements of a CLI command

Nine keywords are supported:

- the "compile" keyword, followed by a file path (including the .vhd/.vhdl extension) triggers compilation of the VHDL file;
- the "layout" keyword orders to produce the layout of a compiled circuit; several arguments are at the user's disposal to fine-tune the process (e.g. the maximum fan-out, the design approach, etc.);
- the "simulate" keyword is used to initiate the simulation of a circuit layout via a testbench file, and many arguments are offered to specify the time parameters (duration, rise and fall times, clock cycle, etc.) and other details;
- the "save" keyword, followed by the desired file path, allows the user to save a layout in .qll format;
- the "export" keyword exports a screenshot of a layout; the output format may be specified with a dedicated argument;
- the "import" keyword, followed by the path of a .qll file, loads that layout into ToPoliNano;
- the "do" keyword, followed by the path of a script (which is actually a simple text file with .do extension), triggers the execution of that script;
- the "quit" keyword closes the program;
- the "clear" keyword clears the screen from all log messages.

Each keyword is discussed in more detail in its respective Sub-section.

### A.1.1 "compile" keyword

The "compile" keyword starts the compilation of a VHDL file containing a circuit.

This keyword only has one, flag-less, mandatory argument: the path of the VHDL circuit that the user wants to compile, including the .vhd/.vhdl extension.

### A.1.2 "layout" keyword

The "layout" keyword begins the layout generation process: the most recently compiled VHDL circuit is analyzed and a physical layout of said circuit is built by ToPoliNano. This keyword has a plethora of arguments (most of which are flagged, three of which are true/false arguments) that give the user fine control over the process:

- "-a/--algorithm", which tells ToPoliNano which optimization algorithm(s) to apply during the process (by default, no such algorithm is applied); the user may employ either a single algorithm or a sequence of (comma-separated) algorithms (e.g. "kl,barycenter");

- "-d/--design-approach", which - as the name implies - specifies the design approach taken by ToPoliNano; the available approaches are "flat" - i.e. the circuit uses no sub-modules -, "p_hierarchical" and "f_hierarchical" (the default) - i.e. the circuit uses sub-modules;

- "-l/--load", a true/false argument which - if specified by the user - tells ToPoliNano (in case it was ordered to employ a hierarchical design approach) to look for sub-modules within its own component library;

- "-g/--geometry", which specifies the magnet geometry (width, height, horizontal spacing, vertical spacing); this argument consists of four comma-separated numbers (the default is 50,100,20,20);

- "-f/--fanout", which defines the maximum allowed gate fan-out (default value is 2);

- "-v/--vertical-wire", which dictates the maximum allowed number of magnets for vertical interconnections (default value is 4);

- "-m/--magnet-clockzone", which specifies the maximum amount of magnets allowed in a clock zone (default value is 4);

- "-s/--sa-embedded", which orders ToPoliNano to employ an embedded version of the simulated annealing algorithm;

- "-w/--domain-wall", which tells ToPoliNano to use domain walls for vertical interconnections;
- "-t/--satimberwolf-parameters", which specifies the values of the key parameters (low temperature, high temperature, first alpha coefficient, second alpha coefficient, third alpha coefficient, first temperature, second temperature, number of iterations) used by the TimberWolf simulated-annealing algorithm; this argument consists of eight comma-separated numbers (the default is 0.1,40000,0.8,0.95,0.8,1000,10000,20);
- "-e/--saexponential-parameters", which specifies the values of the key parameters (low temperature, high temperature, alpha coefficient, number of iterations) used by the Exponential simulated-annealing algorithm; this argument consists of four comma-separated numbers (the default is 0.1,10000,0.95,20);
- "-o/--output", which specifies the path of an output file where ToPoliNano will save the layout once it's been successfully generated (by default, no output path is specified and thus no file is created); the path may or may not include the .qll extension (in the latter case, ToPoliNano will append it automatically).

### A.1.3 "save" keyword

The "save" keyword allows the user to save the most recent layout generated by ToPoliNano to file; the output file is in .qll format.

This keyword only has one, flag-less, mandatory argument: the path of the output file to be created, with or without the .qll extension (in the latter case, ToPoliNano will append it automatically).

### A.1.4 "export" keyword

The "export" keyword allows the user to export a screenshot of the most recent layout generated by ToPoliNano.

This keyword has the following arguments (all flagged):

- "-o/--output", which specifies the output directory (default value is "workspace/Results", with "workspace" being the default ToPoliNano workspace folder);

- "-n/--filename", which determines the output filename (default value is "top-level-name");
- "-f/--format", which dictates the file format; available formats are svg (the default), png, bmp, jpeg, pdf, ps;
- "-a/--area", which specifies the portion of layout to export between the scene (the default), the selected portion of the layout and the current view.

### A.1.5 "import" keyword

The "import" keyword allows the user to load a pre-existing layout from a layout file with .qll format.

This keyword only has one, flag-less, mandatory argument: the path of the file to load the layout from.

### A.1.6 "simulate" keyword

The "simulate" keyword starts the simulation of the most recently generated circuit layout via a VHDL testbench provided by the user.

This keyword has a flag-less argument, which is mandatory and must come immediately after "simulate", consisting of the path to the VHDL testbench file that the user wants to employ in the simulation (including the .vhd/.vhdl extension).

Apart from the flag-less one, many more arguments are available (most of which are flagged, one of which is a true/false argument) that allow the user to tweak the simulation details:

- "-t/--time", which defines the duration of the simulation; the user may specify both a number and a time unit (e.g. 20ns) or just a number (e.g. 40), in which case the default time unit ($\mu$s) will be used (if the user does not explicitly state a simulation time, ToPoliNano will use the default duration of 1$\mu$s); the available time units are ps, ns, $\mu$s and ms;
- "-r/--resolution", which specifies the time step (i.e. the resolution) of the simulation; the available resolutions are 1ps, 10ps, 100ps (the default), 1ns, 10ns, 100ns, 1$\mu$s, 10$\mu$s, 100$\mu$s, 1ms;

- "-c/--clock", which dictates the duration of the three-phase clock cycle; as with the "-t/--time" argument, the user may write both a number and a time unit or just a number, in which case the default time unit (ns) will be used (if the user does not explicitly state a clock period, ToPoliNano will use the default period of 3ns); the available time units are ps, ns, $\mu$s and ms;
- "-u/--risetime", which defines the clock rise-time; the user may employ both a number and a time unit or just a number, in which case the default time unit (ns) will be used (if the user does not explicitly state a rise-time, ToPoliNano will use the default duration of 0ns); the available time units are ps, ns, $\mu$s and ms;
- "-d/--falltime", which defines the clock fall-time; the user may write both a number and a time unit or just a number, in which case the default time unit (ns) will be used (if the user does not explicitly state a fall-time, ToPoliNano will use the default duration of 0ns); the available time units are ps, ns, $\mu$s and ms;
- "-a/--algorithm", which tells ToPoliNano what simulation algorithm to utilise; the available algorithms are "behavioral" (the default) and "single_domain";
- "-f/--fault-analysis", a true/false argument which - if specified by the user - enables fault analysis;
- "-v/--max-variations", which defines the maximum variations (on the X and Y axes) in magnet placement; this argument consists of two comma-separated numbers (the default is 3,2);
- "-i/--iterations", which defines how many simulation iterations to perform;
- "-o/--output", which specifies the path of an output file where ToPoliNano will save the simulation results once the simulation is complete (by default, no output path is specified).

### A.1.7 "do" keyword

The "do" keyword tells ToPoliNano to open a script file and execute its contents.

This keyword only has one, flag-less, mandatory argument: the path of the script file (including the .do extension).

### A.1.8 "quit" keyword

The "quit" keyword prompts ToPoliNano to close; any activities started before entering this keyword that are still being performed by the program (like, for example, a simulation) will be safely completed before ToPoliNano shuts down.

This keyword has no arguments.

### A.1.9 "clear" keyword

The "clear" keyword clears the output of all messages printed by ToPoliNano, leaving it blank.

This keyword has no arguments.

## A.2 Supported system commands

The CLI supports execution of all system commands, both on Windows and on Linux/UNIX; the user may seamlessly execute commands such as "cd", "pwd", "del"/"rm", "dir"/"ls" and many more both from GUI and from no-GUI mode; the user can therefore harness system commands to navigate and manage files without the need for an external file explorer.

## A.3 Path types

A CLI command may employ any of the following three types of file path:

- absolute/full path (e.g. "C:\workspace\newTestbench.vhd"), which is a complete path that starts from the root directory;

- relative path within ToPoliNano's workspace (e.g. "circuit3.vhd"), i.e. a relative path using a predefined sub-folder in ToPoliNano's workspace folder as the parent directory (e.g. the "Input_Files" sub-folder is used when a relative path is found in a "compile" command, the "Testbenches" sub-folder is used when a relative path is found in a "simulate" command, etc.);

- relative path within the current directory of the CLI - or of the terminal, in no-GUI mode - (e.g. "./layoutFile.qll"), which is a relative path using the current directory of the CLI or terminal (referenced with "./") as the parent directory.

## A.4 Program workflow

After ToPoliNano has been opened, not all commands can be executed right away: the program has a specific workflow that the user must be aware of in order to correctly make use of ToPoliNano's functionalities.

The first step in the workflow is the compilation of a VHDL circuit; this is achieved by means of a "compile" command. The most recently compiled circuit is referred to as "the current compiled circuit" in the paragraphs to follow.

The second step in the workflow consists of generating a layout from the current compiled circuit; a "layout" command serves this purpose. The most recently generated layout is referred to as "the current layout" from here on.

Once a layout has been generated, the user may either save to file a screenshot of the layout (obtained via an "export" command) or save to file the layout itself in .qll format (by using a "save" command). Since simulations are run on layouts, they can only be performed if a layout is created beforehand.

The last step in the workflow is the simulation of the current layout, which is started with a "simulate" command.

If, after any step, the user issues a new "compile" command referencing a different VHDL file (which overwrites the current compiled circuit), the current layout will be discarded (therefore ToPoliNano will not be able to perform simulations as long as a new layout is not generated). Similarly, if a new "layout" command with different argument values is entered, the resulting layout will overwrite the current layout; the same occurs for "simulate" commands.

If the user imports a pre-existing layout by means of an "import" command, such layout becomes the current layout and simulations may be run on it.

The "do", "quit" and "clear" commands have no restrictions: they may be used at any time.

## A.5 Preprocessor features

Before entering the topic of writing scripts, it is beneficial to explore the scripting features that the preprocessor (described in Chapter 3) makes available to the user.

### A.5.1 Variable definitions and usage

The feature that underpins the other, more complex ones is the ability to define variables and use them in the script. A variable definition is composed of the following four elements (in the sequence shown):

- the "var" keyword, which notifies the preprocessor that the line contains a variable definition;
- the variable name, which identifies the variable itself;
- the equals sign "=", which represents a value assignment;
- the integer value that is to be assigned to the variable.

Examples of variable definition are "var i = 0", "var num = 3", "var k = 12". Variable names cannot contain numbers, meaning that a variable cannot have a name like "num1", for example.

Variables are defined at the very top of a script; once the preprocessor reaches a line that does not start with "var", it will not accept any more definitions.

Beyond the definitions, in a script the preprocessor recognizes a variable only if its name is enclosed in square brackets; this means that a variable may be employed in conditions (described in Sub-section A.5.2) and filenames by enclosing its name in square brackets (e.g. "[i]", "[num]", etc.).

A variable can be given a new value at any time; taking the "i" variable from the previous example, if - at the end of a While loop (illustrated in Sub-section A.5.3) - the user wants to increase its value by 1, this is accomplished by means of an assignment, in this case "[i] = [i] + 1". An assignment is made of three parts:

- the variable receiving the assignment (left-hand side), enclosed in square brackets to notify the preprocessor that the line contains an assignment;

- the equals sign "=", which represents a value assignment;
- the value being assigned (right-hand side), which can be a number, a variable or an arithmetic addition/subtraction; if the addition/subtraction contains an enclosed variable, the latter is replaced with its current value before evaluating the expression itself.

In the previous example, if [i] has a current value of 3, the expression "[i] + 1" becomes "3 + 1", which is equal to 4, therefore the resulting assignment will be "[i] = 4".

### A.5.2 Conditions

A condition is an inequality containing (at least) one variable, which evaluates to true or to false. The supported relational operators are bigger-than (">"), bigger-than-or-equal-to (">="), equal-to ("=="), less-than ("<"), less-than-or-equal-to ("<=").

An example of condition involving one variable is "[i] > 3", which evaluates to true if the variable "i" has a current value bigger than 3, or to false if its current value is less than or equal to 3. An example of condition involving two variables is "[i] <= [j]", which evaluates to true if the current value of "i" is less than or equal to the current value of "j", or otherwise to false.

Two or more conditions may be combined via the logical operators "and" and "or" to form a single, larger condition; for example, the conditions shown in the two previous examples may be joined together, leading to "[i] > 3 and [i] <= [j]"; the resulting condition is true only if both its sub-conditions are true.

### A.5.3 While loops

A While loop consists of three parts:

- the while-statement (e.g. "while [i] < 4"), which holds the condition;
- the block of statements scoped to the loop, which must include an assignment to the variable utilised in the condition (or else the While loop becomes an infinite loop);
- the "endwhile", i.e. the while-loop termination statement.

An example While loop is shown in Listing A.1.

```
while [i] < 4
    compile file_[i]_top.vhdl
    layout -o output[i].qll
    [i] = [i] + 1
endwhile
```

**Listing A.1**: While loop.

Indentation of scoped statements (either via tabs or via whitespace) is suggested for improved readability but it's not mandatory.

### A.5.4 For loops

Just like the While loop presented in Sub-section A.5.3, the For loop is made up by three parts:

- the for-statement (e.g. "for [i] = 0; [i] < 4; [i] = [i] + 1"), which itself consists of three elements, namely - from left to right - the initial assignment ("[i] = 0"), the condition ("[i] < 4") and the end-of-iteration assignment ("[i] = [i] + 1");
- the block of statements scoped to the loop;
- the "endfor", i.e. the for-loop termination statement.

It is worth mentioning that the initial assignment contained in the for-statement replaces whatever value was held by the variable.

Listing A.2 illustrates an example of For loop.

```
for [i] = 0; [i] < 4; [i] = [i] + 1
    compile file_[i]_top.vhdl
    layout
    save layout_[i]
endfor
```

**Listing A.2**: For loop.

As for While loops, indentation of scoped statements is suggested but not obligatory.

### A.5.5 If constructs

In contrast to While and For loops, If constructs are more complex structures and can vastly differ in composition from each other. The only part that is common to all If constructs is the If section, consisting of:

- the if-statement (e.g. "if [i] == 1"), which contains the condition;
- the block of statements scoped to the if-statement (also referred to as "if-block");
- the "endif", i.e. the If construct termination statement.

The shortest possible If construct consists of just the If section. An If construct may be extended beyond that by adding:

- one or more Else-if (abbreviated as "Elif") sections, each constituted by an elif-statement (e.g. "elif [i] == 2") which contains a condition and by a block of statements scoped to it;
- an Else section, comprising of an else-statement ("else", with no conditions) and a block of statements scoped to it.

Elif sections and the Else section are not mutually exclusive, meaning that the user is given great freedom of choice when building an If construct: if, if-else, if-elif, if-elif-elif-..., if-elif-else, if-elif-elif-...-else are all valid If constructs. Many examples showing the variety of If constructs that the user may enjoy are shown in Listing A.3.

```
if [i] == 0
    save output_[i]
endif

/////////////////////////

if [i] == 0
```

```
    save output_[i]
else
    export
endif

////////////////////////

if [i] == 0
    save output_[i]
elif [i] == 1
    simulate -o results.txt
endif

////////////////////////

if [i] == 0
    save output_[i]
elif [i] == 1
    simulate -o results.txt
elif [i] == 2
    export
endif

////////////////////////

if [i] == 0
    save output_[i]
elif [i] == 1
    simulate -o results.txt
else
    quit
endif

```

```
/////////////////////////

if [i] == 0
    save output_[i]
elif [i] == 1
    simulate -o results.txt
elif [i] == 2
    export
else
    quit
endif
```

**Listing A.3**: If constructs.

Indentation of scoped statements is here too only suggested.

## A.6 Writing scripts

With Sections A.4 and A.5 serving as guidelines, it is easy to automate the workflow by means of scripts. This Section illustrates how to write scripts of increasing complexity, starting from the simplest ones.

Be aware that comments are not supported inside scripts.

### A.6.1 Script 1 - Plain list

Script 1 exemplifies the simplest kind of script that ToPoliNano can execute: a plain list of commands.

```
compile circuit.vhd
layout -a kl,barycenter -f 3
save layoutFile.qll
simulate -t 10us -o simResults.txt
quit
```

**Listing A.4**: Script 1.

As shown in Listing A.4, Script 1 automates the typical workflow of ToPoliNano, executing one after the other the compilation, layout and simulation steps, together with saving to file the layout itself and quitting the program at the end.

### A.6.2 Script 2 - Plain list with a nested script

Script 2 is similar to Script 1 but it makes use of the nested scripting capabilities of the CLI by employing a "do" command which calls another script file.

```
compile circuit.vhd
layout -d flat
do otherscript.do
simulate -t 100ns
quit
```

**Listing A.5**: Script 2.

```
save output.qll
export --format png
```

**Listing A.6**: otherscript.do, used by Script 2.

The contents of Script 2 are equivalent to those of the comparison script shown in Listing A.7.

```
compile circuit.vhd
layout -d flat
save output.qll
export --format png
simulate -t 100ns
quit
```

**Listing A.7**: Comparison script, equivalent to Script 2.

### A.6.3 Script 3 - While loop

Script 3 introduces a modicum of complexity: a While loop (described in Sub-section A.5.3) iterating over a series of commands.

```
var i = 0

while [i] < 3
    compile circ_[i].vhd
    layout
    save layoutFile_[i].qll
    [i] = [i] + 1
endwhile
quit
```

**Listing A.8**: Script 3.

The contents of Script 3 are equivalent to those of the comparison script shown in Listing A.9.

```
compile circ_0.vhd
layout
save layoutFile_0.qll
compile circ_1.vhd
layout
save layoutFile_1.qll
compile circ_2.vhd
layout
save layoutFile_2.qll
quit
```

**Listing A.9**: Comparison script, equivalent to Script 3.

### A.6.4 Script 4 - For loop

Script 4 employs a slightly more complicated loop: the For loop (illustrated in Subsection A.5.4).

```
var i = 0

for [i] = 0; [i] < 3; [i] = [i] + 1
    compile circ_[i].vhd
    layout
    export -n screenshot[i] --format jpeg
endfor
```

**Listing A.10**: Script 4.

### A.6.5 Script 5 - For loop with nested If construct

Script 5 is more complex than the previous ones: it makes use of a For loop which in turn contains an If construct.

```
var i = 0

compile circuit.vhd
for [i] = 0; [i] < 3; [i] = [i] + 1
    if [i] < 2
        layout -d flat
        simulate tb[i].vhd -r 10ps -o res[i].txt
    else
        layout -l
        simulate tb[i].vhd -r 1ps -o res[i].txt
    endif
    export -n screen_[i] -f png
endfor
```

**Listing A.11**: Script 5.

# List of figures