# POLITECNICO DI TORINO

**Master's Degree in Electronic Engineering**


**Master's Degree Thesis**

# Development of Advanced CAD Tools for the Design and Simulation of Digital Circuits Based on Emerging Nanotechnologies

**Supervisors**

**Prof. Maurizio ZAMBONI**

**Ph.D. Fabrizio RIENTE**

**Ph.D. Umberto GARLANDO**

**Candidate**

**Manuel DAMIANI**

**July 11, 2022**

# Summary

Currently available electronic products and circuits are heavily based on the CMOS technology, first invented in 1963. Over the course of the past decades, integration of electronic circuits has become one of the determining factors in the technological advances of our time. It was originally theorized by Moore's Law that the number of transistors in an integrated circuit (IC) would double every year. However, today this statement is not necessarily true anymore, due to various causes such as transistor scaling reaching critical limits and extremely high density of ICs, both challenging today's production processes, and more. Today, it is more apparent than ever that continuing to only rely on CMOS technology is leading to a slowdown in technological progress, and different alternative solutions are currently being researched more heavily as time goes on.

To deal with these increasing issues, and to ensure that constant improvements can be achieved, new technologies that go by the name of Beyond CMOS are being researched and studied. One of the most important is QCA (Quantum-dot Cellular Automata), which has been explored throughout the course of this work. It is a technology capable of achieving improvements in various areas where the CMOS might be limited moving forward, such as reduced power consumption and faster transmission of signals. The QCA technology is based on the interaction between the electrons of quantum cells, enabling logic operations. These quantum cells are bi-stable, meaning they possess two possible states at equilibrium which can be associated to logic '0' and '1'. The coupling effects due to the interactions between the magnetic and electric fields of adjacent cells are exploited in order to propagate binary data through the cells. Different possible implementations of QCA are currently being studied: the most prominent ones are Magnetic QCA (specifically iNML and pNML), in which the elementary cells are nanomagnets, and MolQCA, whose cells are made of molecules. These technologies are also referred to as Field-Coupled Nanocomputing (FCN).

In order to design and perform simulations of QCA circuits, following a design flow similar to the one found in commercial tools for CMOS design, new experimental software is required. The ToPoliNano framework developed by the

VLSI Group at Politecnico di Torino is a set of EDA tools that allows to design FCN circuit layouts, synthesize their HDL description, simulate the circuit and visualize the results in a 3D viewer. The framework consists of three main software, namely ToPoliNano, a CAD tool which is able to design, test and simulate circuits; MagCAD, a stand-alone software which enables custom circuit design, and FCNS Viewer, a 3D viewer that allows to view the behavior of each circuit element throughout the simulation. The ToPoliNano framework is continuously updated with new features: during the course of this work, ToPoliNano and FCNS Viewer have been modified to fully support MolQCA simulations in the former, and to insert a waveform viewer in the latter.

ToPoliNano has been extensively modified in order to integrate the full design flow also for Molecular QCA, whereas previously it was present only for iNML. To do this, after some preliminary work to integrate new command line functionalities, MolQCA has been activated by importing its shared library, creating a new class of simulation settings to allow a full customization by the user, and inserting the required parameters. The molecular clock has been changed to a four-phase clock, and the internal simulation controller was refactored to improve the passing of the parameters between classes; in addition, the simulator has been modified to support a verbose simulation, providing a more detailed representation of the system evolution by analyzing it in smaller time steps. The result is a fully fledged molecular simulator, capable of analyzing the pin placement to ensure that the output cell is correctly connected to its pin by analyzing all possible orientations, and providing results that can be studied for MolQCA research.

As far as the Waveform Viewer is concerned, the FCNS Viewer software has also been extensively updated to provide a feature that can import the results stored in a table by the ToPoliNano simulation, containing the signal values at each time step, and translate those values in a wave chart. The new plot allows to visually analyze the behavior of each signal during the simulation, such as value and amplitude changes; it supports both iNML and MolQCA simulation waveforms alike, and provides features such as horizontal and vertical zoom, and a line that intercepts the signals and updates the signal labels with their values at that time instant.

The new software implementations further expand the ToPoliNano framework, moving it even closer in terms of feature completeness to the commercial tools available for the CMOS technology. Future developments will allow to study and research an evolution toward hybrid systems that could be capable of reaping the benefits of both CMOS and FCN technologies, achieving technological progress and improvement. The Waveform Viewer could also be updated with an increasing number of features, to further improve its quality.

# Acknowledgements

To my teachers and professors, who taught me everything I know.

To my friends and colleagues, who shared the journey with me.

To my family, who raised me and always supported me.

To E.M.
My star, my perfect silence.

# Table of Contents

# Listings

# List of Figures

# Acronyms

**CMOS**
Complementary Metal-Oxide Semiconductor

**QCA**
Quantum-dot Cellular Automata

**iNML**
in-plane Nano Magnetic Logic

**pNML**
perpendicular Nano Magnetic Logic

**FCN**
Field-Coupled Nanocomputing

**EDA**
Electronic Design Automation

**CAD**
Computer-Aided Design

**HDL**
Hardware Description Language

**GUI**
Graphical User Interface

**CLI**
Command Line Interface

# Chapter 1

# Introduction

Today's electronic products and circuits are mainly based on the CMOS technology, originally invented in 1963 by Frank Wanlass at Fairchild Semiconductor. Over the course of the past decades, integration of electronic circuits has become one of the determining factors in the technological advances of our time. Thanks to technologies like CMOS, huge progress has been made in terms of miniaturizing and speeding up circuits, leading to a progressively wider distribution and availability of electronic chips, nowadays integrated in most everyday life consumer products, as well as in many different industrial sectors.
The integration of transistors, governed by Moore's Law, has been highly regarded as the guiding star for technological progress since 1965. It was originally theorized that the number of transistors in a dense integrated circuit (IC) would double every year [1]; this was later revised to be every two years, in 1975.

However, as challenges faced by modern technologies become increasing in number and complexity, this statement originally formulated almost 60 years ago is not necessarily true anymore. There are many different causes for this, ranging from transistor scaling reaching way below deep submicron, challenging today's production processes to reach gate lengths approaching the nanometer, to power consumption becoming an increasing issue, to the extremely high density of integrated circuits, which have led to studies on new solutions, like the implementation of logic in memory and 3D stacked ICs. Today, it is more apparent than ever that continuing to only rely on CMOS technology is leading to a slowdown in technological progress, and different alternative solutions are currently being researched more heavily as time goes on.

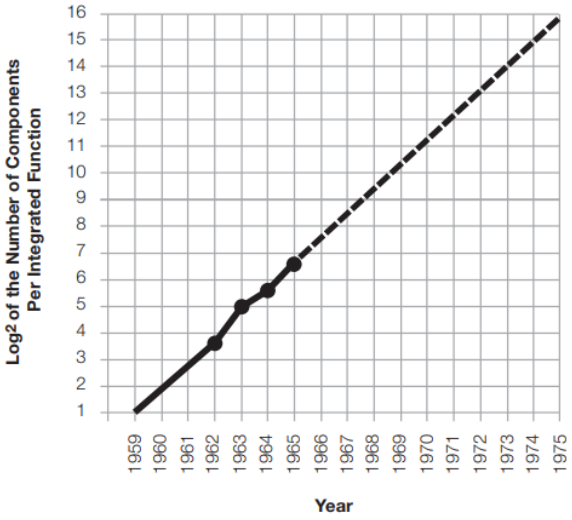



**Figure 1.1:** Gordon Moore's theorization of transistor increment [1].

## 1.1 The Search for Beyond CMOS Technologies

To deal with the technological progress slowing down, and to make sure that constant improvements can be achieved, new technologies that go by the name of *Beyond CMOS* are being researched and studied. One of the most important is QCA (Quantum-dot Cellular Automata), which has been explored throughout the course of this work. It is a technology capable of achieving improvements in various areas where the CMOS might be limited moving forward, such as reduced power consumption and faster transmission of signals. Thanks to QCA, different possibilities for achieving bit transmissions through a circuit even while it is not powered on, and for implementing logic-in-memory to increase memory speed even further, are being explored.

### 1.1.1 QCA

The QCA technology is based on the interaction between electrons of different quantum cells, which enables logic operations [2]. These quantum cells are bi-stable, meaning they possess two possible states at equilibrium; the two free electrons in a cell can occupy two of the four available quantum dots, which are arranged in a square area. Each square contains a limited number of charges that interact with its neighboring cells via Coulomb effects. QCA cells operate in a regime where Coulomb effects dominate over tunneling effects [3]. As shown in Figure 1.2, the two equilibrium states of a quantum cell can correspond to binary states, '0' and

'1', depending on the position of the two electrons: naturally, they repulse each other due to Coulomb forces, which is why the two possible states consist in the electrons being located in the opposite corners of the QCA cell. The association of the two states with the binary logic values is defined by convention [4]. Due to the charge distribution, the net charge of the individual cell is equal to zero.

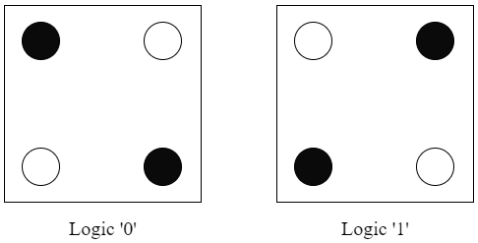



**Figure 1.2:** The two equilibrium configurations of the basic QCA cell.

The coupling effects generated by the interactions between the magnetic and electric fields of adjacent QCA cells are exploited in order to propagate binary information through the cells, thus creating a digital circuit. For this reason, these kinds of Beyond CMOS technologies are also referred to as FCN (Field-Coupled Nanocomputing). FCN computations are made without any electric current flow, allowing for a very low-power utilization [5], much lower than traditional CMOS applications.
Different types of circuits can be achieved by placing multiple cells in sequence. The most basic example is a simple wire (Figure 1.3), which is necessary in order to transmit bits through a circuit: this can be achieved by creating a line of QCA cells, and considering for instance the leftmost cell as the input, and the rightmost one as the output.

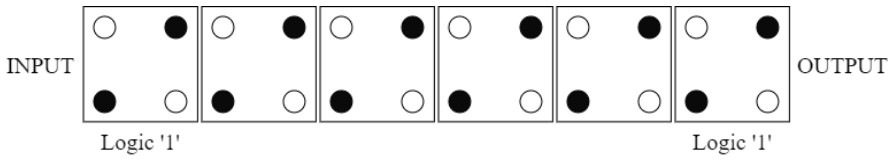
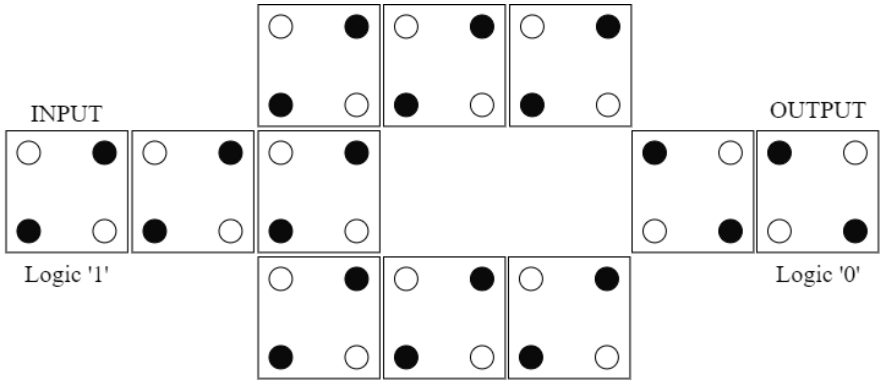


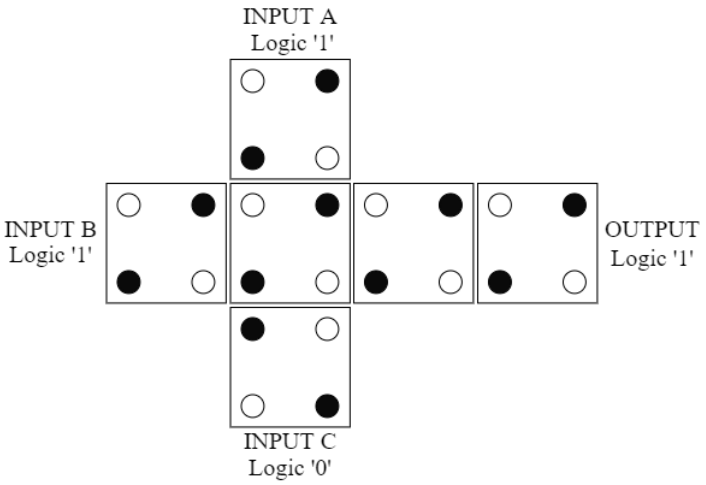

**Figure 1.3:** A simple QCA wire.

By creating QCA circuits that implement the basic logic functions like NOT, AND and OR, implementations that can substitute, or co-exist with, traditional CMOS circuits can be obtained. For instance, a NOT function (an inverting gate) can be accomplished by putting two cells adjacent to each other diagonally [3], which causes the position of the quantum dots of these two cells to be inverted as shown in Figure 1.4a. Another building block that can be used to implement boolean functions in QCA technology is the majority voter (Figure 1.4b), a gate

with three inputs whose output is equal to the majority of the inputs. The majority voter itself can also be used to implement the AND and OR, by simply wiring one of the inputs to a '0' or to a '1' respectively [5].




**(a)** A QCA inverter.




**(b)** A QCA majority voter.

**Figure 1.4:** Two essential QCA structures.

The different combinations obtainable from these elements allow to achieve electronic circuits equivalent to their CMOS counterparts. However, in order to actually propagate the information through a QCA circuit, an external field known as *clock* is mandatory. The effect of a quantum cell field is not sufficiently strong to influence the neighboring cells by itself: for this reason, the clock field is used to force the cells in an unstable state called *reset* state. After this state is reached, the clock is disabled and the new equilibrium state of the circuit will depend on the interaction between the cells in the so-called *active* state [5].

The circuit is typically divided in multiple clock zones (Figure 1.5), where each one is composed of a fixed maximum number of cells, and these clock zones are reset one at a time by alternating their phases [2]. Thus, the various solutions proposed in literature consist of different types of multi-phase clocking mechanisms. As it becomes apparent from this explanation, one of the big advantages is the fact that the only power consuming step is the application of the clock signals: all of the interactions between the QCA cells, propagating data through the circuit, require no energy, as they occur while in the active state where no external influence is necessary.

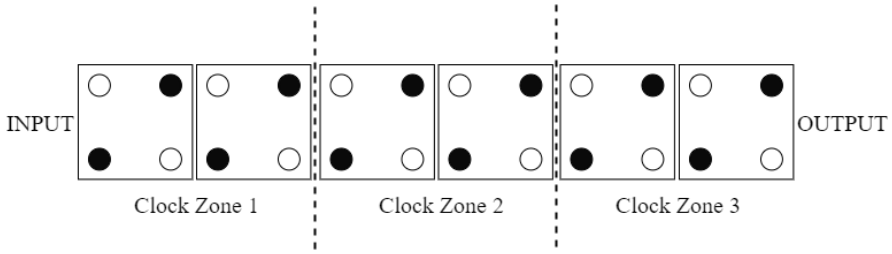



**Figure 1.5:** A QCA wire split into three clock zones.

Having detailed the possible theoretical implementations of QCA technology, the physical implementations can be analyzed. There are multiple different possibilities, and in this work mainly two of them have been studied and faced: MQCA (magnetic QCA) and MolQCA (Molecular QCA).

## 1.2 MQCA

Magnetic QCA, usually referred to by its more specific name of NML (Nano Magnetic Logic), is a technological implementation that consists in the use of rectangular nanomagnets as quantum cells. The main benefits that QCA can provide with respect to traditional CMOS circuits are present in NML: this technology has no static power consumption, as the clock field is only applied in the reset phase, it is immune to radiations and its elementary cells are capable of being both a logic element and a memory [2].

Two types of MQCA can be distinguished: iNML (in-plane NML), and pNML (perpendicular NML); the difference between the two technologies is related to the plane where the magnetization vector is located. In iNML technology, the magnetization vector is on the same plane as the cells, while in pNML the magnetization vector is perpendicular to the plane where the cells are located. In regards to MQCA, this work focused on iNML technology, whereas the pNML implementation was outside of the scope of this Thesis.

### 1.2.1 iNML

The iNML implementation of Magnetic QCA is based on rectangular-shaped nanomagnets used as bistable elements, whose binary nature is related to the two possible orientations of the in-plane magnetization vector, ultimately able to encode a logic '0' or a logic '1'. Since these cells are nanomagnets, their physical dimensions are in the order of tens of nanometers, usually (50 × 100 × 20) nm or (60 × 90 × 20) nm [6].

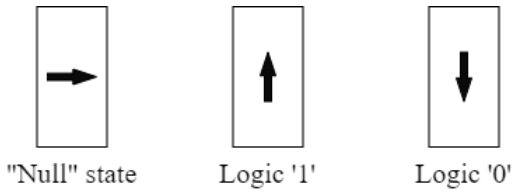



**Figure 1.6:** The three magnetization states of an iNML cell.

This technology is part of the Field-Coupled Nanocomputing family, in this particular case referring to the coupling interaction between magnetic fields of the neighboring cells that can constitute an iNML circuit. As it is observed for QCA technologies in general, iNML nanomagnets also need to be excited by an external source before their interactions take place, leading to the data propagation through the circuit. This external source is the clock field, which can jumpstart the interactions between the magnets by causing an instability in the system; the instability is obtained because the clock field rotates the magnetization vector of 90°, along the short axis of the magnet, which represents a "null" state [7]. Then, through the active phase with the clock removed, the magnetization vector of each cell assumes one of the two possible configurations, after said interactions occur with no external influence and no static power consumption.

Like in the QCA theorization, the clock field is a multi-phase clock also in iNML technology. A circuit could be split in multiple clock zones, where each phase of the clock corresponds to a zone containing a limited number of magnets, usually up to four or six to avoid problems related to thermal noise [6]. In this way, the clock zones are activated sequentially, allowing for the interaction between adjacent cells to propagate the binary data from the input to the output.

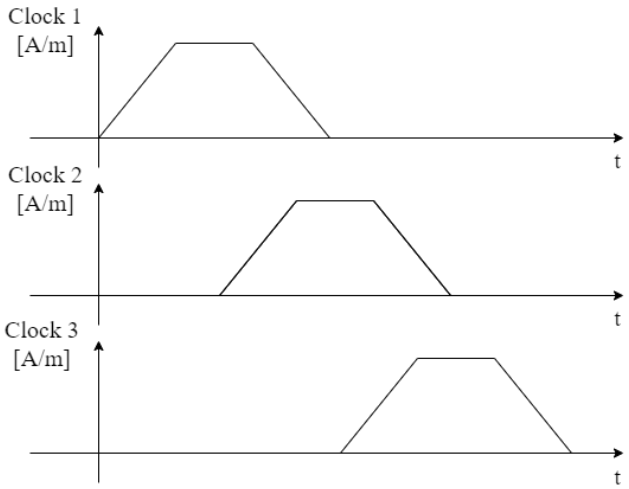



**Figure 1.7:** A three-phase clock system.

The two types of magnetic coupling that can occur between adjacent nanomagnets are F (Ferromagnetic) or AF (Anti-Ferromagnetic), depending on their placement (Figure 1.8a). Placing two magnets next to each other, in a row, causes the magnetization vectors to be antiparallel (AF coupling); instead, placing them in a column leads to a parallel orientation of the two magnetization vectors (F coupling).

Let us now go over the elementary gate examples previously listed in the generic QCA case, this time referring to iNML logic. The most basic element, which allows for the propagation of digital information through the circuit, is a simple wire. A wire can be obtained with a row of magnets: depending on their number, the resulting circuit could either be a wire or an inverter. If we consider an input magnet connected to a row of magnets, with the last cell being the output, then a wire is obtained with an even number of nanomagnets. On the contrary, an odd number of magnets results in an inverting circuit (Figure 1.8b).

The previously explained majority voter is another gate that can be efficiently implemented in iNML technology, by combining both types of magnetic coupling between nanomagnets to obtain the expected majority voter function. Figure 1.8c shows an iNML majority voter as generated by ToPoliNano, a software that will be later detailed in Chapter 2.

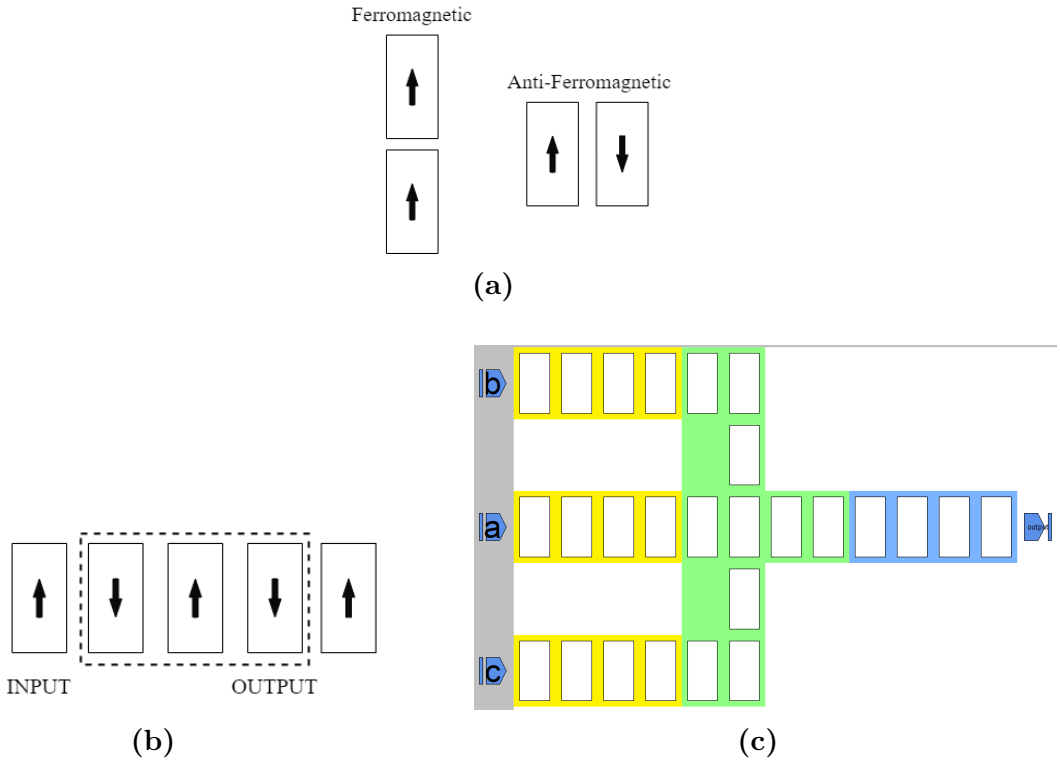
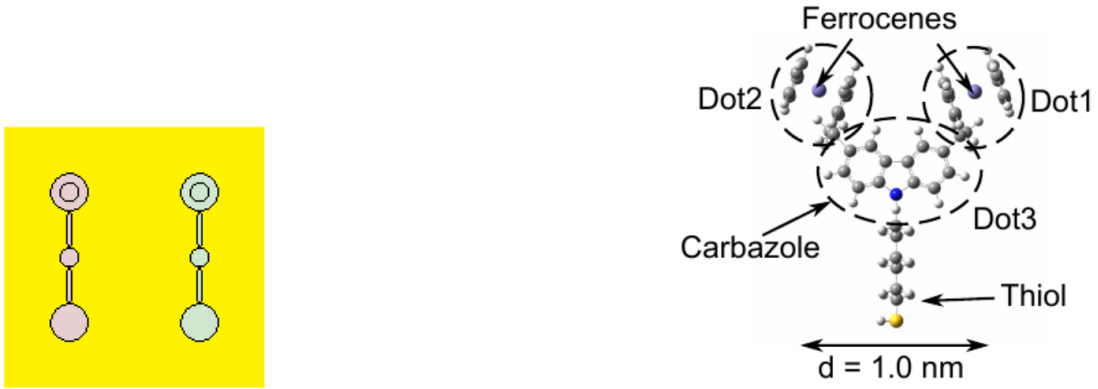



**Figure 1.8:** (a) Types of magnetic coupling in iNML cells. (b) An iNML inverter. (c) An iNML majority voter.

## 1.3 MolQCA

Another type of QCA technology that is subject to research and very promising is Molecular QCA. This implementation makes use of molecules as elementary cells. Different possible choices exist: the MolQCA technology considered in this work uses artificially synthesized Bisferrocene molecules [8]. In particular, two of these molecules constitute an elementary cell (Figure 1.9a), providing a total of six quantum dots that can allow for two possible configurations, resulting in the binary behavior typical of digital information, plus an additional null state that is assumed during the reset state, when the clock field is applied. More specifically, as shown in Figure 1.9b, there are two ferrocene units that correspond to Dot1 and Dot2, used as the two logic states for '0' and '1'; instead, the carbazole group in the middle is the Dot3 responsible for the null state [9].

As explained in the iNML case, there are different basic structures and gates that can be obtained in MolQCA technology as well. To transfer data along a circuit, a basic wire is necessary: when the logic information is being transmitted from the input towards the output, the adjacent cells will interact with each other,




**(a)** **(b)**

**Figure 1.9:** (a) A MolQCA elementary cell. (b) The Bisferrocene molecule [9].

propagating the data. The clock field that is used in MolQCA technology can be implemented as a multi-phase clock split in three or four clock zones.

## 1.4 The Need for QCA Design Tools

The purpose of studying *Beyond CMOS* technologies is not simply to reach a complete substitution of the CMOS technology. Instead, these new, emerging technologies could be integrated into current CMOS implementations, reaching a hybrid solution that can reap the benefits of both CMOS and QCA.

Design of electronic circuits today is done resorting to EDA (Electronic Design Automation) and CAD (Computer-Aided Design) tools, which allow to automate a lot of design steps, facilitating the designer's work. However, no commercial tools that manage these emerging QCA technologies currently exist, to allow for their design and implementation. For this reason, the VLSI Group at Politecnico di Torino has been researching and working on new EDA tools that can serve this purpose. These new tools can be used to develop circuits using NML and MolQCA implementations, and eventually to reach a Hybrid technology. The result of these efforts is the ToPoliNano Framework.

# Chapter 2

# The ToPoliNano Framework

The ToPoliNano framework is a set of EDA tools that allows the designer to create circuits in MQCA and MolQCA technologies, compile a corresponding HDL (Hardware Description Language) description, simulate the circuit either behaviorally or physically, and visualize the results of the simulation throughout all time instants in a 3D viewer. The framework has been developed and is in constant update at the VLSI Group in Politecnico di Torino; it is entirely written in C++ using the Qt environment and libraries, and supported on multiple operating systems, namely Windows, Linux and macOS. Thanks to the tools in the ToPoliNano framework, designers can follow a top-down strategy similar to the one provided by the commercially available software for CMOS technology [5].

ToPoliNano is a CAD tool which is able to design, test and simulate circuits based on iNML and MolQCA technologies: the structural description of the circuit can be given through a VHDL file, then compiled using a library of QCA components, or by importing a layout designed in MagCAD [2]. MagCAD is a stand-alone software which enables custom circuit design in the aforementioned QCA technologies, allowing the user to decide the position of the elementary cells, set their physical parameters, and place the input and output pins. The third element of the framework is FCNS Viewer, containing a 3D viewer that can visualize the QSS files generated by a ToPoliNano simulation, in order to check the behavior of each individual cell of the circuit at every time step.

Throughout the course of this work, ToPoliNano and FCNS Viewer have been modified to fully support MolQCA simulations in the former, and to insert a waveform viewer in the latter.

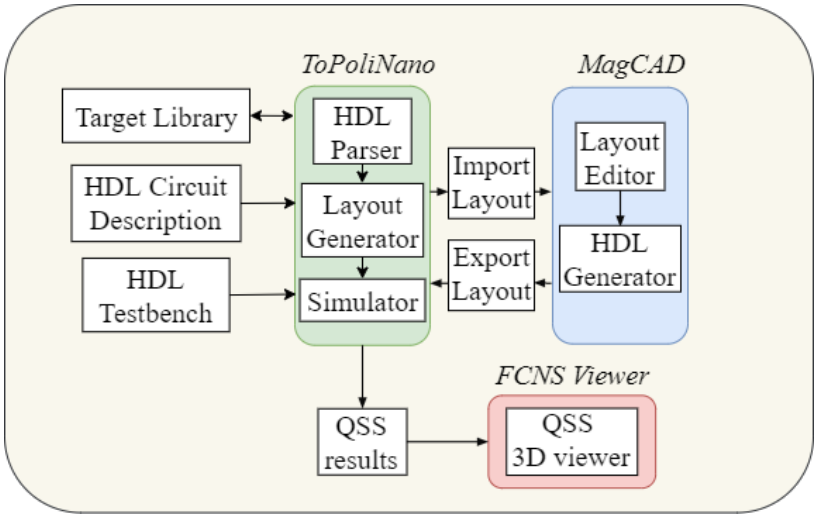



**Figure 2.1:** The ToPoliNano framework.

## 2.1 ToPoliNano

The centerpiece of the framework is ToPoliNano. The workflow of this software consists of three main steps: the compile phase performs the compilation of an HDL circuit description, mapping the components thanks to a library of QCA components, the layout phase allows to generate an actual circuit layout following one of the available algorithms, and the simulation phase lets the user decide the parameters for the simulation and run it, obtaining as outputs a table file listing all the signal values at every step, and a QSS file for each time step. If the designer has already obtained a QLL circuit layout from MagCAD, it can be imported directly, bypassing the compile and layout steps. As far as technologies are concerned, ToPoliNano supports the iNML implementation of Magnetic QCA, and as it will be shown in the next chapters, this work has been largely focused on the full insertion and support of Molecular QCA as well.

Figure 2.2 shows the main window of the ToPoliNano GUI (Graphical User Interface), which also contains a CLI (Command Line Interface) that allows to insert written commands and to read log messages printed during the execution. On the left side, the HDL files for the currently selected technology are shown, both input files and testbenches. The top part of the window contains all the elements necessary to access the features of the program: creating or opening an HDL file, saving it, accessing one of the three simulation steps, and so on. The bottom part contains the previously mentioned command line, while the right side shows the list of components that are available for the selected technology. The following descriptions will focus on the already supported iNML technology.

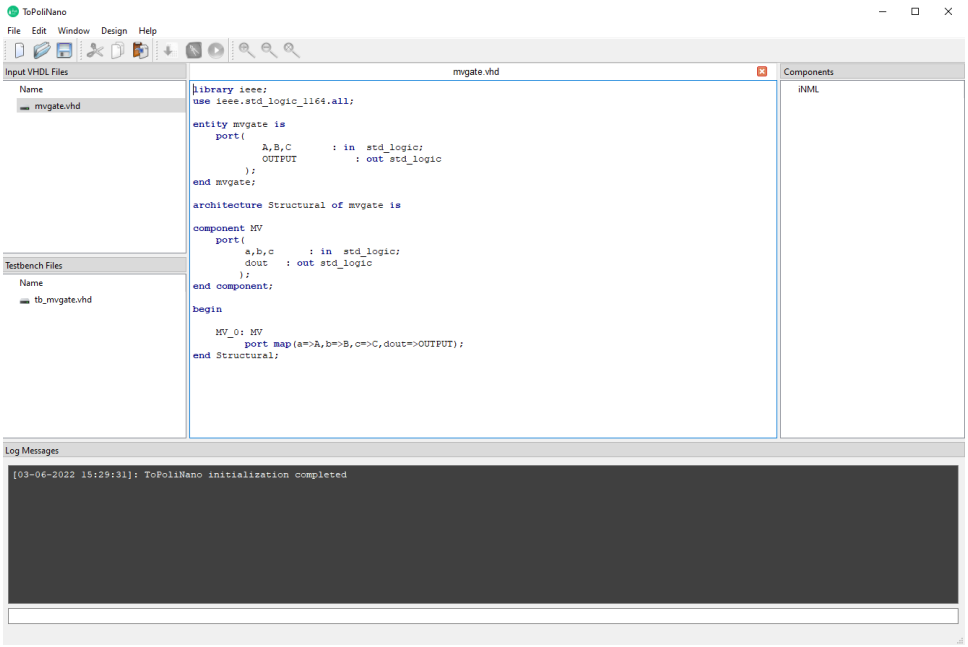



**Figure 2.2:** ToPoliNano user interface.

### 2.1.1 Compilation Phase

The first part of the ToPoliNano workflow is the compilation phase. The available HDL files have been either previously synthesized, or have been written using the available text editor: the user is allowed to select one to be used as top-level entity, and all the elements in the hierarchy are compiled; an internal data structure is also written, to aid with the layout phase that comes afterwards [5]. The process is performed by an HDL parser. All the necessary information about input and output ports of the entities, their interconnections, and the technological details are saved: this is all fed to ToPoliNano's place and route engine, used in the layout phase [2].

To be successfully compiled, the input files must be organized in a structural way, down to the four fundamental gates described in the previous chapter, which are available in the target library: inverter, AND gate, OR gate and majority voter. The hierarchical, top-down structure is maintained in the obtained graph, which is the end result of the compilation phase.

### 2.1.2 Layout Phase

The second phase consists in the generation of a circuit layout, representing the placement of all the elementary cells, connections and input/output ports. The

starting point of the layout phase is the aforementioned graph, which is elaborated according to the selected optimization algorithm before the physical mapping of the circuit elements takes place. Different design approaches can also be selected. Before the layout settings can be opened, the user must select one of the files listed in the testbench section as the active testbench.

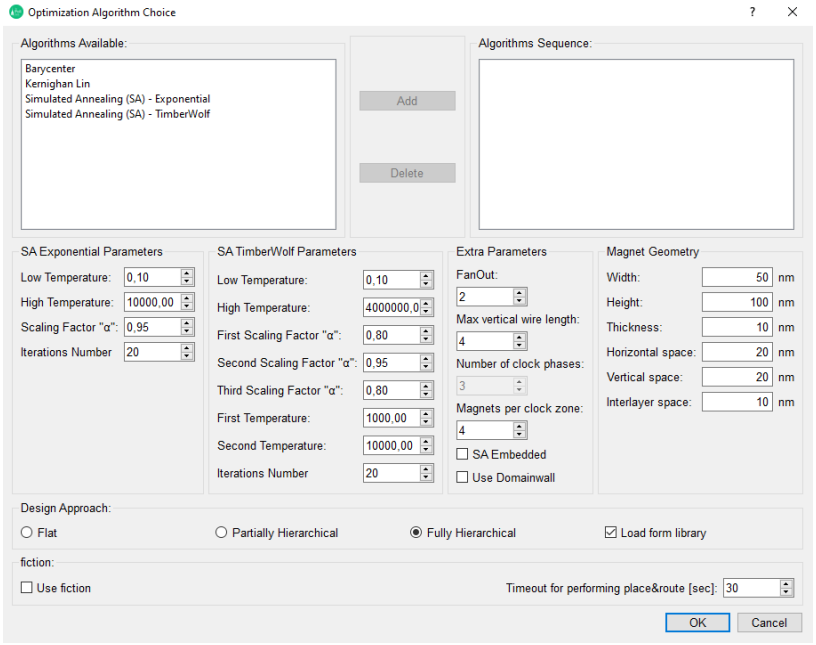



**Figure 2.3:** Layout phase customizable settings.

The first possible selection is among the different optimization algorithms: the choice can be made between Barycenter, Kernighan-Lin, and two kinds of Simulated Annealing (Exponential and TimberWolf); more than one can be selected if necessary, to be executed sequentially.

Below the algorithm choice, there is a list of parameters than can be modified to suit the designer's needs. These include, but are not limited to, temperature levels and scaling parameters for Simulated Annealing, the maximum fan-out, the number of magnets per clock zone and the geometrical parameters of the magnets, namely width, height, thickness and the distance between each magnet.

Another choice that can be made is that of the design approach. There are three possible choices: flat, partially hierarchical or fully hierarchical. The flat approach consists in flattening the hierarchy of the circuit, while the fully hierarchical method exploits components which are already available in the target library, so as to speed up the layout generation [2]. Instead, the partially hierarchical method constitutes a trade-off between the previous two, by using only the first level of the hierarchy

and flattening the ones below [5]. By checking the adjacent box, labeled "Load from library", ToPoliNano can save time by loading components that have already been designed.

There is also the option to use *fiction* during the layout generation. It is an open source tool [10] developed at Bremen University: it has been integrated inside ToPoliNano to provide an exact algorithm for cell placement, as opposed to the previously described algorithms, which are heuristic. Being an exact algorithm, *fiction* could be very time consuming, and for this reason a timeout can be specified. After this maximum time has been reached, if the placement has not yet been completed, ToPoliNano resumes with its heuristics.

When the layout generation starts, ToPoliNano first elaborates the graph, in order to optimize the netlist and evaluate the maximum fan-out of each node, then performs the physical mapping of the nodes to the nanomagnets. The resulting layout is displayed in the main window area, with some simulation details printed in the message log area, including the total elapsed time, the number of cells, and more. If a layout has already been generated through MagCAD, the first two phases of the workflow can be omitted by simply importing the layout QLL file, allowing to move forward with the final step, which is the simulation itself.

### 2.1.3 Simulation Phase

With a completed layout, it is possible to perform the last step: simulating the circuit by applying the testbench stimuli, and obtaining the evolution of the system at every time step.

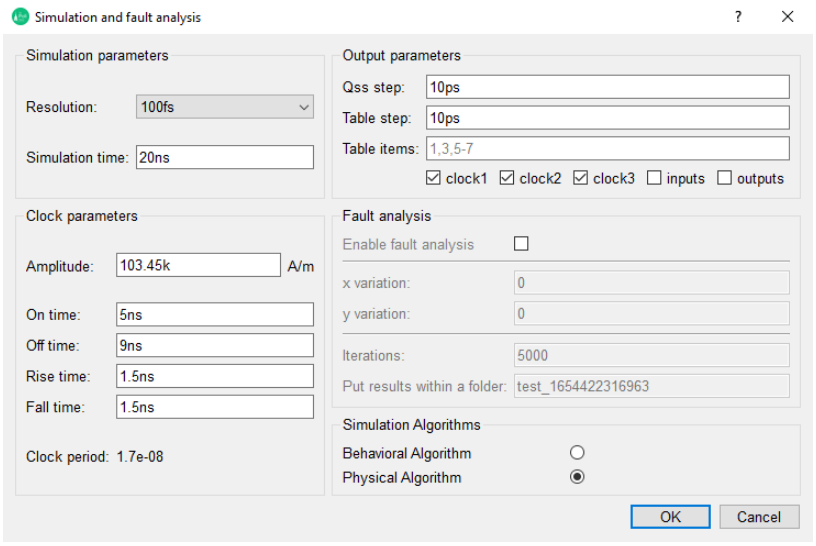



**Figure 2.4:** Simulation phase customizable settings.

The simulation settings can be tailored to the specific case. The user can input the total simulation time, and select the resolution from a drop down menu. The clock is also customizable: both its amplitude, in A/m, and its timing characteristics, namely on time, off time, rise time and fall time, can be individually selected. The window shows the automatically computed clock period, based on the inserted time values.

Relative to the outputs of the simulation, it is possible to choose both the QSS step and the table step. The QSS step infers how many QSS files, containing the magnetization values of all cells at a certain QSS step, should be generated. Similarly, the table step indicates which time instants should be stored in the output *table.txt* file. The available checkboxes allow to decide what if any clock signals or inputs/outputs should be listed in the table.

The final choice is the type of simulation algorithm to be used. The behavioral algorithm does not consider the physical properties of the cells, and uses ideal clock signals with sharp rising and falling edges. Instead, the physical algorithm is the Single Domain one, which provides a physical simulation. Once the simulation starts, ToPoliNano's embedded simulator applies the testbench stimuli, computes the magnetization values for every magnet, and stores the results only for the requested QSS steps and table steps.

## 2.2 MagCAD

MagCAD allows the designer to create a new layout from scratch, or to modify an already existing QLL file. It is a layout editor which provides the ability to choose the preferred technology (iNML, pNML or MolQCA), and to customize many different physical parameters. For iNML (Figure 2.5), the width, height and thickness of the magnets, as well as the distances between them, and the number of clock phases and elements per clock zone can be selected; for MolQCA layouts, the intermolecular distance, the number of clock zones and the maximum elements per zone are available for customization.
Once the target technology has been chosen along with its settings, the main window and all its options become accessible (Figure 2.6). From the item menu on the left, the basic cells and some basic structures, like the majority voter, the slanted magnets for the AND and OR gates, a cross-wire and more, can be selected and freely placed on the canvas. The user can also assign a clock zone, rotate the elements, or change the previously selected technological settings.

Another crucial element is the placement of the input and output pins, which can be rotated, attached to a cell and named. Once the layout is completed, it can either be simply saved, or it can be exported. If only the QLL is generated, it can be imported into ToPoliNano, but the user is still required to create a testbench

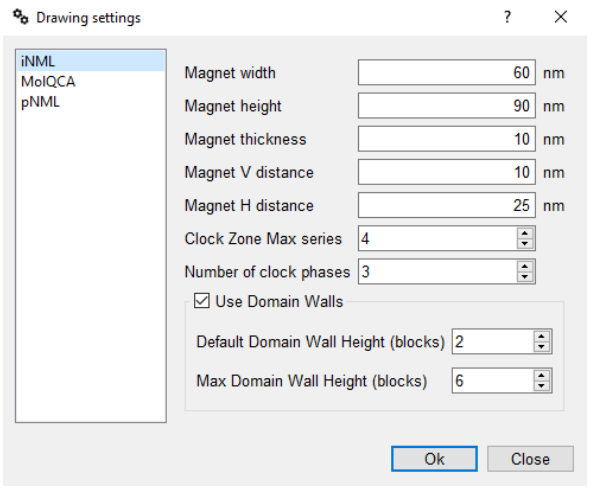



**Figure 2.5:** MagCAD drawing settings.

from scratch. By exporting the layout as a component, MagCAD automatically generates a VHDL file with the circuit description, and a testbench template which can be easily customized. The generated netlist is checked for design rules and constraints, and it can be used for simulation with commercial HDL simulators, namely QuestaSim and ModelSim [11].

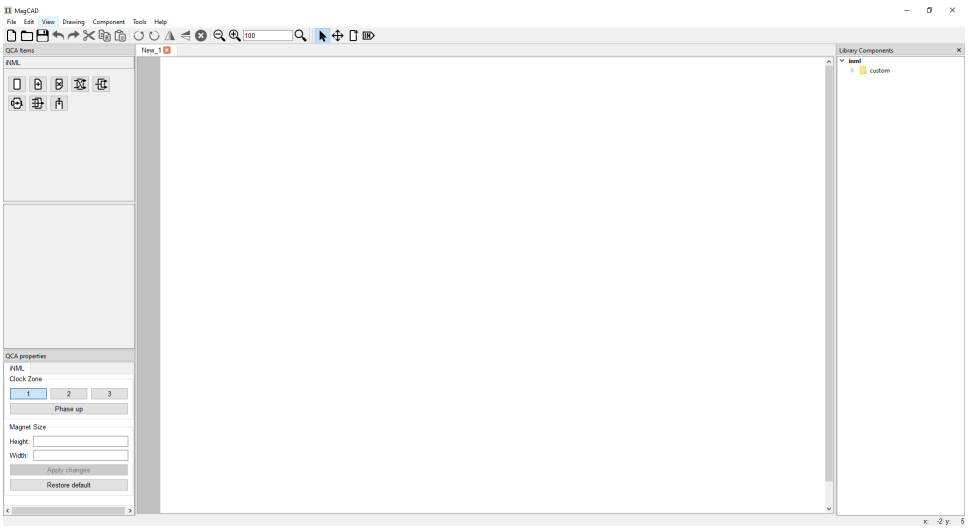
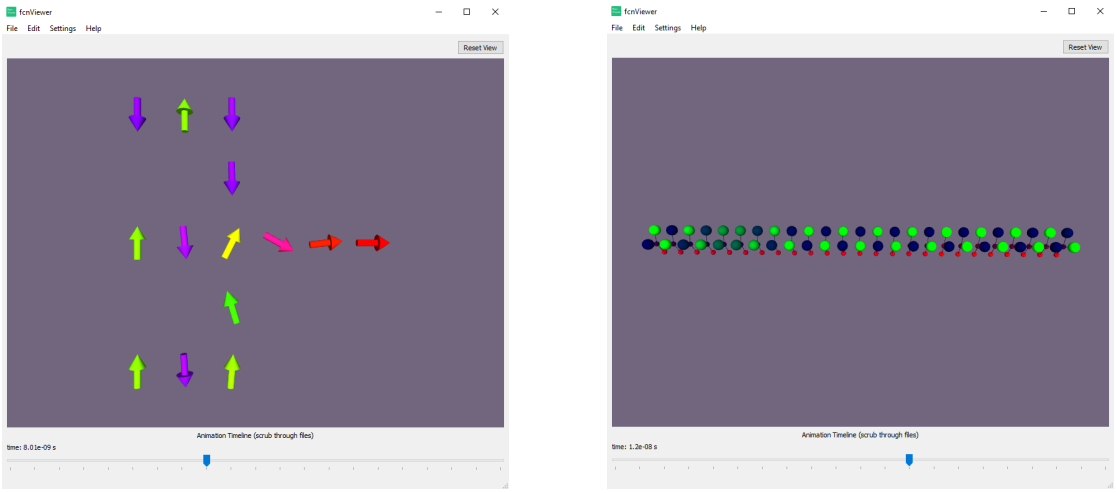


**Figure 2.6:** MagCAD user interface.

## 2.3 FCNS Viewer

To conclude the workflow, FCNS Viewer is a software that can be used to visualize the QSS files produced by the ToPoliNano simulation, providing a visual representation of how all the elementary cells of the circuit have evolved during the course of the simulation. FCNS Viewer can be used to view single QSS files, or to select a folder with the simulation results: a manual slider allows to very easily move in time to verify the circuit behavior at each time instant.

It supports both Magnetic QCA and Molecular QCA: in particular, for iNML circuits, it is possible to either view the nanomagnets, which are color coded based on the value of the magnetization vector, or to use a vector view, where the vectors themselves are visible and can be more easily evaluated.




**(a)** **(b)**

**Figure 2.7:** (a) An iNML majority voter. (b) A MolQCA wire.

The work performed in this Thesis integrated a new functionality: a waveform viewer, used to visualize the contents of the *table.txt* file generated by ToPoliNano at the end of the simulation by representing each signal as a waveform; it will be detailed in Chapter 4.

# Chapter 3

# Enabling Molecular QCA Simulations

In order to provide a wider support for QCA technologies, Molecular QCA must also be fully implemented in the totality of the ToPoliNano framework. The last step for this goal is to enable the MolQCA simulation controller, which had already been designed, to make it fully functional and allow the choice between iNML and MolQCA technologies. In order to perform this task, some preliminary work is necessary, starting with the implementation of a few missing CLI commands relative to the recent implementation of new data structures and of the internal FCN Solver in a previous work.

## 3.1 New CLI Commands Implementation

The workflow inside ToPoliNano can be handled mainly in two ways, which are also interchangeable as the user sees fit: using the GUI buttons and menus, or typing commands through the integrated CLI. The previously mentioned three phases of the ToPoliNano simulation can be run by using the three corresponding commands, meaning *compile*, *layout* and *simulate*; each of these commands can receive additional arguments that act as the settings one would select in the graphical interface.

The class that is in charge, among other things, of interpreting the written commands, is the CLI Backend. The command interpreter inserts the received arguments in a vector, so that they can be passed to the structs containing parameters for the various phases; if an argument is not chosen by the user, the passed value is a default one.

The implementation of the FCN Solver inside ToPoliNano required the addition of new possible parameters for the CLI commands, in both the layout and simulation

steps. The new parameters that can be specified in the layout phase include the thickness of the iNML nanomagnets, the inter-layer space and the number of clock phases (Listing 3.1). The declaration of the command line arguments allows to specify the type, the letter that can be used to input an argument before its value (for instance, if the chosen letter is "k", the argument can be inserted by the user by typing "-k" followed by the desired value), the full name of the parameter, a brief description that appears when the help command is summoned, and the default value to be applied in case the argument is not specified when the layout command is called.

```
// Magnet thickness
magnet_thickness_arg_ = new TCLAP::ValueArg<unsigned int>("k", "magnet-thickness",
     "set the magnet-thickness value", false, 10, "num", *cmdParser);
arg_list_.push_back(magnet_thickness_arg_);
// Interlayer space
interlayer_space_arg_ = new TCLAP::ValueArg<unsigned int>("z", "interlayer-space",
     "set the interlayer-space value", false, 10, "num", *cmdParser);
arg_list_.push_back(interlayer_space_arg_);
// Number of phases
phase_number_arg_ = new TCLAP::ValueArg<unsigned int>("p", "phase-number", "set
    the phase-number value", false, 3, "num", *cmdParser);
arg_list_.push_back(phase_number_arg_);
```

**Listing 3.1:** New arguments for the layout phase.

Furthermore, there were also parameters missing from the simulation command, which allow the user to further customize this step by specifying the desired timing resolution, the clock on-time and off-time values and its amplitude. The creation of the QSS files and the table.txt as outputs of the simulation could also allow the user to specify the QSS step, the time step for the table file, and which elements to print in the table. Listing 3.2 shows the addition of these new parameters into the argument list.

```
// Resolution
resol_arg_ = new TCLAP::ValueArg<std::string>("r", "resolution", "set the time
    resolution (1fs, 10fs, 100fs(default), 1ps, 10ps, 100ps, 1ns, 10ns, 100ns, 1us
    , 10us, 100us, 1ms)", false, "100fs", "res", *cmdParser);
arg_list_.push_back(resol_arg_);
// Amplitude
amplitude_arg_ = new TCLAP::ValueArg<double>("z", "amplitude", "set the amplitude
    [num] (default 103.45e3)", false, 103.45e3, "amplitude", *cmdParser);
arg_list_.push_back(amplitude_arg_);
// Time-on
ontime_arg_ = new TCLAP::ValueArg<std::string>("x", "ontime", "set the on time [
    num][unit (ps, ns, us, ms)] (default 5ns); unit is optional", false, "5ns", "
    period", *cmdParser);
arg_list_.push_back(ontime_arg_);
// Time-off
offtime_arg_ = new TCLAP::ValueArg<std::string>("l", "offtime", "set the off time
    [num][unit (ps, ns, us, ms)] (default 9ns); unit is optional", false, "9ns", "
    period", *cmdParser);
arg_list_.push_back(offtime_arg_);
// Output QSS step
```

```
14 QssStep_arg_ = new TCLAP::ValueArg<std::string>("q", "qssstep", "set the output
       simulation step (default = 10ns); unit is optional", false, "10ps", "QssStep",
        *cmdParser);
15 arg_list_.push_back(QssStep_arg_);
16 // Output table step
17 TableStep_arg_ = new TCLAP::ValueArg<std::string>("b", "tablestep", "set the
       output table step (default = 10ns); unit is optional", false, "10ps", "
       TableStep", *cmdParser);
18 arg_list_.push_back(TableStep_arg_);
19 // Output item list
20 TableItems_arg_ = new TCLAP::ValueArg<std::string>("e", "tableitems", "set the
       list of table items to display(ck1,ck2,ck3,inputs,outputs) (default is empty);
        do not separate numbers and/or commas with spaces", false, "noItems", "
       TableItems", *cmdParser);
21 arg_list_.push_back(TableItems_arg_);
```

**Listing 3.2:** Additional arguments for the simulation phase.

Once all the arguments have been received, they must be passed to the structures containing all the layout and simulation parameters. To do this, they must also be checked for correctness: for instance, the clock timing parameters and its amplitude must be positive. As soon as one of the values does not meet the requirements, an error message is generated and shown in the command log, interrupting the execution of the current phase (Listing 3.3). If no errors are found, before the current phase can begin a summary of all the parameters is printed in the command log section.

Among the previously selectable parameters, the clock period and the simulation step have been removed: the former has been removed because the user is now able to freely select the rise time, on time, fall time and off time; the latter is no longer present as it has been substituted with the resolution, which is more significant.

```
1  // RESOLUTION
2  std::string resolStd = resol_arg_->getValue();
3  QRegExp regex("10{0,2}[f,p,n,u,m]s");
4  if ( !regex.exactMatch(QString::fromStdString(resolStd)))
5  {
6      usage_and_errors_gen_->interpreterError("invalid resolution", command_arg_->
       shortID(), resol_arg_->shortID() );
7      return false; // Abort interpretation without executing the request
8  }
9  double resolution = convertTimeString( QString::fromStdString(resolStd) );
10 // AMPLITUDE
11 double amplitude = amplitude_arg_->getValue();
12 if(amplitude < 0) // If the user wrote an invalid amplitude, generate an error
13 {
14     usage_and_errors_gen_->interpreterError("invalid amplitude", command_arg_->
       shortID(), time_arg_ ->shortID() );
15     return false; // Abort interpretation without executing the request
16 }
17 // ON-TIME
18 std::string ontimeStd = ontime_arg_->getValue();
19 double ontime = convertTimeString( QString::fromStdString(ontimeStd), "ns" );
20 if (ontime < 0) // If the user wrote an invalid time unit, convertTimeString()
       returns a negative value
21 {
```

```
22      usage_and_errors_gen_->interpreterError("invalid time unit", command_arg_->
        shortID(), time_arg_->shortID() );
23      return false; // Abort interpretation without executing the request
24  }
25  std::string TableItemsStd = TableItems_arg_->getValue();
26  QString items = QString::fromStdString(TableItemsStd);
27  if (TableItemsStd.compare("noItems") == 0) // If the user did not use the "--
        tableitems" option...
28  {
29      items = ""; // Empty table is set
30  }
31  QList<QString> TableList = items.split(",", Qt::SkipEmptyParts); // Split the
        string into a list of substrings
```

**Listing 3.3:** Some of the new parameter correctness checks.

In order to ensure that all the compilation, layout and simulation arguments work correctly, before ToPoliNano is even launched for testing, a tester integrated in the code can be launched independently. The tester can launch some dummy ToPoliNano executions, by loading examples of commands written in a few test text files, and report whether the execution was successful or not.

The CLI controller tester has been updated to include the new commands, to check for operational correctness. Its workflow consists in loading the text files containing the dummy command lines, grabbing each individual argument, and comparing it with the expected one. If each parameter passes the tests, every test is marked as a PASS; otherwise, whichever tests were unsuccessful are marked as FAIL. Listing 3.4 contains some lines of code showing the workflow of the tester, in particular for the new parameters.

```
1   QFETCH(int, expected_magnet_thickness);
2   QFETCH(int, expected_interlayer_space);
3   QFETCH(int, expected_phase_number);
4
5   int magnThickness = argParams.magnet_thickness;
6   int interlayerSpace = argParams.magnet_layer_space;
7   int phaseNum = argParams.phaseNumber;
8
9   QCOMPARE(magnThickness, expected_magnet_thickness);
10  QCOMPARE(interlayerSpace,   expected_interlayer_space);
11  QCOMPARE(phaseNum,  expected_phase_number);
12
13  QVector<int> layMagnThickness;
14  QVector<int> layinterlayerSpace;
15  QVector<int> layphaseNumber;
16
17  if(valueSet.contains("error", Qt::CaseInsensitive)) // If no signal will be called
        because a faulty request was made...
18  {
19      layMagnThickness.append( 1 );
20      layinterlayerSpace.append( 1 );
21      layphaseNumber.append( 1 );
22  } else
23  {
24      layMagnThickness.append( valueList.at(10).toUInt() );
25      layinterlayerSpace.append( valueList.at(11).toUInt() );
26      layphaseNumber.append( valueList.at(12).toUInt() );
```

```
}

if(( layoutReqs.size() != layMagnThickness.size() ) || ( layoutReqs.size() !=
    layinterlayerSpace.size() ) || ( layoutReqs.size() != layphaseNumber.size() )
{
    QString errorQString = "Error: request file and replies file for 'layout' 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 "layout"
    functionality
} else
{
    QTest::addColumn<int>("expected_magnet_thickness");
    QTest::addColumn<int>("expected_interlayer_space");
    QTest::addColumn<int>("expected_phase_number");

std::string stdLabel = labelsVector.at(i).toStdString(); const char* cLabel =
    stdLabel.c_str();
QTest::newRow(cLabel) << layMagnThickness.at(i) << layinterlayerSpace.at(i) <<
    layphaseNumber.at(i); // Create a row in the testing table
}
```

**Listing 3.4:** Snippets of the CLI controller tester.

When the tester is launched, all the available tests are performed and the results are obtained. Figure 3.1 shows the results at the end of the testing procedure: the output log contains the list of performed tests, their description and their outcome.

```
******** Start testing of CLIbackendTest ********
Config: Using QtTest library 5.15.2, Qt 5.15.2 (x86_64-little_endian-llp64 shared (dynamic) debug build; by MSVC 2019), windows 10
PASS   : CLIbackendTest::initTestCase()
PASS   : CLIbackendTest::testTokenizeAndParseCmd(empty string)
PASS   : CLIbackendTest::testTokenizeAndParseCmd(complete help request)
PASS   : CLIbackendTest::testTokenizeAndParseCmd(complete help request)
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [compile])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [layout])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [save])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [export])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [import])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [quit])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [simulate])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [do])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(invalid command [do])
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][-r])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(interpreter error [do])
PASS   : CLIbackendTest::testTokenizeAndParseCmd(arg value missing [do])
PASS   : CLIbackendTest::cleanupTestCase()
Totals: 27 passed, 0 failed, 0 skipped, 0 blacklisted, 26ms
********* Finished testing of CLIbackendTest *********
================================================================================
```

**Figure 3.1:** CLI Backend test results.

## 3.2 First Iteration of Molecular QCA Simulation

In order to achieve the goal of fully implementing MolQCA simulations in ToPoliNano, many preparations are required. The first necessary addition is importing the MolQCA plugin: the software uses shared libraries in DLL format, which need to be added to the project first so they can be loaded at the program startup.

Since the molecular plugin library has already been created, the only necessary action is to include it in the Qt project file (Listing 3.5). The plugin is copied from the source folder into the correct tech folder, and added to the QMake targets; QMake is the utility implemented by the Qt development environment to automate the generation of the Makefile for the compiled project. In this way, the plugin is correctly activated and linked to ToPoliNano.

```
MOLQCAPLUGIN_SOURCE_PATH = $$DESTDIR/../libs/tech/molqca_001/plugin/windows/
    $$BUILD_TYPE/*
MOLQCAPLUGIN_DEST_PATH = $$DESTDIR/tech/molqca_001/plugin
createFolderMolQCA.target = $$DESTDIR/tech/molqca_001/plugin
createFolderMolQCA.commands = $(MKDIR) \"$$shell_path($$MOLQCAPLUGIN_DEST_PATH)\"
molqcaPluginCopy.commands = $(COPY_FILE) \"$$shell_path($$MOLQCAPLUGIN_SOURCE_PATH
    )\" \"$$shell_path($$MOLQCAPLUGIN_DEST_PATH)\"
QMAKE_EXTRA_TARGETS += first sslCopy createFolderInml createFolderMolQCA
    inmlPluginCopy molqcaPluginCopy molqcaCharacteristicsCopy fictionCopy
```

**Listing 3.5:** MolQCA Plugin import.

ToPoliNano contains a wizard, which can be opened by selecting "New workspace" under the file drop-down menu header. The wizard allows the user to choose a new workspace directory and select the desired technology. However, the only available technology was Magnetic QCA, and in particular the iNML implementation. In order to allow the selection of MolQCA technology, some changes have been made to the wizard.

Adding the choice between MQCA and MolQCA can be achieved by inserting a new button in the window (Listing 3.6). When the user clicks the desired button, the field is saved so it can be accessed by the Wizard, which would then set the active target technology and technological node by passing it to the Application Controller (Listing 3.7).

```
m_MQCAButton = new QRadioButton(tr("Magnetic QCA"));
m_MQCAButton->setChecked(false);
m_MolQCAButton = new QRadioButton(tr("Molecular QCA"));
m_MolQCAButton->setChecked(false);

// record fields to be retreived later on acceptance.
registerField("MQCA", m_MQCAButton);
registerField("MolQCA", m_MolQCAButton);

QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(m_MQCAButton);
layout->addWidget(m_MolQCAButton);
setLayout(layout);
```

**Listing 3.6:** Creation of the MolQCA selection button.

```
AppController::Instance().SetTargetTechnologyName("MolQCA");
AppController::Instance().SetTechnologyNode("MolQCA");
```

**Listing 3.7:** Passing the selected technology to the App Controller.

Once the technology has been set, the ToPoliNano main window is loaded successfully. Out of the three main phases of the software, when MolQCA technology is selected only the simulation phase should be available: a molecular technology layout can be obtained only through MagCAD and imported into ToPoliNano for simulation. For this reason, it is necessary to disable the possibility to compile HDL files if the selected technology is molecular; this has to be done in both the CLI and the GUI. As far as the command line is concerned (Listing 3.8), the user should simply receive a warning message explaining that compilation is disabled for MolQCA technology.

```
if(AppController::Instance().GetTargetTechnologyName() == "MolQCA")
{
    text = "MolQCA technology selected: compilation is disabled";
    Logger::instance()->logMessage(text, Logger::WarningLog);
}
```

**Listing 3.8:** Disabling HDL compilation in the CLI.

Concerning the ToPoliNano GUI, the compile button must be disabled, making it greyed out. To do so, in addition to the warning message, the compile action needs to be made unavailable, so that no HDL file can be selected as top-level entity for its compilation, and the button is deactivated (Listing 3.9).

Additionally, in iNML, when a layout file is opened the compile button is enabled, as the user might want to change entity and possibly obtain a new layout: this possibility is removed if MolQCA is the active technology.

```
toggleActionAvailability(disableAction, compile_action_, &hdl_compile_completed_);
```

**Listing 3.9:** Disabling HDL compilation in the GUI.

### 3.2.1 MolQCA Simulation Settings

With the first two phases being definitively handled, implementing the simulation itself is the final step. When the user launches the simulation, ToPoliNano opens a window where the simulation parameters can be customized as needed. While this feature had already been implemented for iNML, it was missing for Molecular QCA. The first step in this direction consists in creating the simulation settings window, in a new class called *MolQCASimulationSettings*. This class is called when the simulation is requested by the user with MolQCA as the active technology (Listing 3.10). The struct containing the simulation data is updated with the settings selected by the user, a new simulation folder is created, the previously imported molecular layout is copied into the simulation folder, and an internal signal that

activates the simulation task is asserted. If an error has occurred, the process is aborted.

```
if(AppController::Instance().GetTechnologyNode() == "MolQCA")
{
    molqcaform = new MolQCASimulationSettings(AppController::Instance().
    GetWorkspacePath(), this, floorplan_parameters_.value("PhaseNumber").toInt());

    if (molqcaform->registerParameters())
    {
        molqcaform->updateParameters(simulation_data_container_);
        m_gui_state = SIMULATE;

        // create simulation folder
        QString simulation_folder = AppController::Instance().GetWorkspacePath() +
     "/Simulations/" + QFileInfo(circuit_toplevel_name_).baseName() + "_" +
    QDateTime::currentDateTime().toString("yyyyMMdd_HHmmss");

        // make unique folder
        QString tmp = simulation_folder;
        int i = 1;
        while(QDir(simulation_folder).exists())
        {
            simulation_folder = tmp + QStringLiteral("_(%1)").arg(i++);
        }
        simulation_folder += "/";
        QDir().mkpath(simulation_folder);

        // save the layout inside the simulation folder
        SaveLayout(simulation_folder + QFileInfo(circuit_toplevel_name_).baseName
    () + ".qll");

        // Starts simulation task
        emit SignalSimulateCircuit(simulation_data_container_,
    circuit_toplevel_name_, input_waveform_filename_, simulation_folder);
    }else
    {
        text = "Warning - The simulation process has been aborted";
        Logger::instance()->logMessage(text, Logger::WarningLog);
        if(execSem->available() == 0) // If the CLIcontroller had acquired the
    lock...
            execSem->release(1); // ...Release it
    }
}
```

**Listing 3.10:** Setting up the molecular simulation.

In addition, when MolQCA is the target technology, once the layout is imported the simulate action must be activated, provided that the user selects a valid testbench first (Listing 3.11). Once the testbench has been selected, when the simulation button is pressed, the parameter settings window pops up.

```
if(AppController::Instance().GetTargetTechnologyName() == "MolQCA")
{
    if (selected)
    {
        if (layout_completed_)
        {
            input_waveform_filename_ = value.toString();
            if (!simulate_action_->isEnabled())
```

```
                simulate_action_->setEnabled(true);

            input_testbench_selected_ = true;
            text = "Input Waveform selected: " + value.toString();
            Logger::instance()->logMessage(text);
        }else
        {
            text = "Simulation cannot start, no layout available";
            Logger::instance()->logMessage(text, Logger::WarningLog);
        }
    } else
    {
        text = "Error, no testbench selected";
        Logger::instance()->logMessage(text, Logger::ErrorLog);
    }
}
```

**Listing 3.11:** Ensuring a testbench has been selected.

As previously mentioned, the window with the molecular settings is handled by the *MolQCASimulationSettings* class. The parameters are grouped in separate sections of the same window, distinguishing between timing parameters, physical parameters and output parameters. Over the course of this work, different parameters have been modified, and new ones have been added; their effective implementation will be detailed later. To ensure that the numeric parameters are reasonable, each of them implements a *QRegExp* function. This function allows the data to be validated only if it follows the specified contraints, for example only three digits between zero and nine (0-999). A default value is also written in each field.

The first section of the settings is called "Simulation Parameters" (Listing 3.12): it contains the timing parameters of the simulation, namely the number of time instants to simulate in a single clock cycle, the total simulation time, and the clock amplitude measured in V/nm. The timing parameters are handled differently in the case of MolQCA simulations: instead of explicitly specifying the time and its unit of measurement, discrete time instants are considered; the total amount of simulation steps specified by the user is a dimensionless quantity, where each step is handled internally by the simulator as one nanosecond.

Similarly, the clock signal is obtained by querying the user about the number N of instants each part of the clock should have, which are equal in this case: rise time, on time, fall time and off time are all identical, each consisting of N steps (handled internally as N nanoseconds). Therefore, the total clock period is $4N$. A picture depicting how the clock is managed has also been included, for added clarity.

Once all the options have been created, they are added to the window layout as separate widgets, in different positions.

```
void MolQCASimulationSettings::createTimeParametersBox()
{
```

```
3     // GroupBox Simulation initialization
4     m_GroupSimulation.setTitle("Simulation parameters");
5     QRegExp rx("[0-9]{1,3}");
6     QRegExp rxAmp("[0-9]*\\.?[0-9]*[k]?");
7     QPixmap m_TimestepImage;
8
9     m_TimestepImage.load(":/images/molqca_timesteps.png");
10    QSize image_size = m_TimestepImage.size();
11    int scaling_factor = 2.5;
12    m_TimestepImage = m_TimestepImage.scaled(QSize(image_size.width()/
      scaling_factor, image_size.height()/scaling_factor), Qt::KeepAspectRatio);
13    m_TimestepLabel.setText("Select the number N of instants to simulate:");
14    m_TimestepLabel.setWordWrap(true);
15    m_TimestepImageLabel.setPixmap(m_TimestepImage);
16    m_Timesteps.setValidator(new QRegExpValidator(rx));
17    m_Timesteps.setText("5");
18
19    m_SimTimeLabel.setText("Total amount of steps:");
20    m_SimTime.setValidator(new QRegExpValidator(rx));
21    m_SimTime.setText("100");
22
23    m_ClockAmplitudeLabel.setText("Clock amplitude:");
24    m_ClockAmplitude.setValidator(new QRegExpValidator(rxAmp));
25    m_ClockAmplitude.setText("2");
26    m_ClockAmplitude_suffix.setText("V/nm");
27
28    // GridLayout creation
29    m_GridLayoutSimulation.addWidget(&m_SimTimeLabel,0,0);
30    m_GridLayoutSimulation.addWidget(&m_SimTime,0,1);
31
32    m_GridLayoutSimulation.addWidget(&m_TimestepLabel, 1, 0);
33    m_GridLayoutSimulation.addWidget(&m_Timesteps, 1, 1);
34    m_GridLayoutSimulation.addWidget(&m_TimestepImageLabel, 2, 0, 1, 3, Qt::
      AlignCenter);
35
36    m_GridLayoutSimulation.addWidget(&m_ClockAmplitudeLabel, 3, 0);
37    m_GridLayoutSimulation.addWidget(&m_ClockAmplitude, 3, 1);
38    m_GridLayoutSimulation.addWidget(&m_ClockAmplitude_suffix, 3, 2);
39
40    // GroupBox set layout
41    m_GroupSimulation.setLayout(&m_GridLayoutSimulation);
42 }
```

**Listing 3.12:** Simulation parameters.

The section called "Physical Parameters" shown in Listing 3.13 handles settings that impact the physical simulation depending on their choice. These are the damping factor, the stability threshold, which is a value that indicates (when reached) the stability condition under which the simulator will move to the next simulation step; the interaction radius, which is the maximum distance from a molecular cell that is considered when taking into account interactions with neighboring elements. To conclude, the input's low and high physical levels, measured in V, indicate the maximum and minimum value of the input signals: the lower half of the range could then be considered as logic '0', while the top half corresponds to a logic '1'.

```
1 void MolQCASimulationSettings::createPhysicalParametersBox()
```

```
{
    m_GroupPhysicalParam.setTitle("Physical Parameters");

    QRegExp rxdamping("[0-9]*\\.?[0-9]?");
    QRegExp rxstability("[0-9]*\\.?[0-9]?[e]?[-]?[1-9]?");
    QRegExp rxintradius("[0-9]*\\.?[0-9]?[e]?[-]?[1-9]?");
    QRegExp rxvoltagelow("[+/-]?[0-9]*\\.?[0-9]?");
    QRegExp rxvoltagehigh("[+]?[0-9]*\\.?[0-9]?");

    m_DampingFactorLabel.setText("Damping Factor");
    m_DampingFactor.setValidator(new QRegExpValidator(rxdamping));
    m_DampingFactor.setText("0.6");

    m_StabilityThresholdLabel.setText("Stability Threshold");
    m_StabilityThreshold.setValidator(new QRegExpValidator(rxstability));
    m_StabilityThreshold.setText("2e-7");

    m_InteractionRadiusLabel.setText("Interaction Radius");
    m_InteractionRadius.setValidator(new QRegExpValidator(rxintradius));
    m_InteractionRadius.setText("2e-6");

    m_VoltageLowLabel.setText("Input Low Physical Level");
    m_VoltageLow.setValidator(new QRegExpValidator(rxvoltagelow));
    m_VoltageLow.setText("-4.5");
    m_VoltageLow_suffix.setText("V");

    m_VoltageHighLabel.setText("Input High Physical Level");
    m_VoltageHigh.setValidator(new QRegExpValidator(rxvoltagehigh));
    m_VoltageHigh.setText("+4.5");
    m_VoltageHigh_suffix.setText("V");

    m_GridPhysicalParam.addWidget(&m_DampingFactorLabel, 0, 0);
    m_GridPhysicalParam.addWidget(&m_DampingFactor, 0, 1);
    m_GridPhysicalParam.addWidget(&m_StabilityThresholdLabel, 1, 0);
    m_GridPhysicalParam.addWidget(&m_StabilityThreshold, 1, 1);
    m_GridPhysicalParam.addWidget(&m_InteractionRadiusLabel, 2, 0);
    m_GridPhysicalParam.addWidget(&m_InteractionRadius, 2, 1);
    m_GridPhysicalParam.addWidget(&m_VoltageLowLabel, 3, 0);
    m_GridPhysicalParam.addWidget(&m_VoltageLow, 3, 1);
    m_GridPhysicalParam.addWidget(&m_VoltageLow_suffix, 3, 2);
    m_GridPhysicalParam.addWidget(&m_VoltageHighLabel, 4, 0);
    m_GridPhysicalParam.addWidget(&m_VoltageHigh, 4, 1);
    m_GridPhysicalParam.addWidget(&m_VoltageHigh_suffix, 4, 2);

    m_GroupPhysicalParam.setLayout(&m_GridPhysicalParam);
}
```

**Listing 3.13:** Physical parameters.

While the iNML simulation allows the user to choose between physical and behavioral simulation, the molecular one is purely physical. Instead, the possible choice is between disabling or enabling a setting called *verbose mode*. In short, enabling this mode allows for a more complex simulation, analyzing not only the behavior of the system at every time step specified by the user, but instead going in more detail by considering even smaller microsteps. Its actual implementation in the molecular simulator will be tackled later; in regards to the settings window, the Verbose section allows to either enable it or disable it, and pass this information to

the simulator (Listing 3.14).

```
void MolQCASimulationSettings::createVerboseBox()
{
    m_GroupVerboseMode.setTitle("Verbose Mode");

    m_enableVerboseLabel.setText("Enable");
    m_disableVerboseLabel.setText("Disable");
    m_enableVerboseRadio.setChecked(true);

    m_GridVerboseMode.addWidget(&m_enableVerboseLabel, 0, 0);
    m_GridVerboseMode.addWidget(&m_enableVerboseRadio, 0, 1);
    m_GridVerboseMode.addWidget(&m_disableVerboseLabel, 1, 0);
    m_GridVerboseMode.addWidget(&m_disableVerboseRadio, 1, 1);

    m_GroupVerboseMode.setLayout(&m_GridVerboseMode);

    QObject::connect(&m_enableVerboseRadio, &QRadioButton::clicked, this, &
    MolQCASimulationSettings::RadioVerboseModeToggled);
    QObject::connect(&m_disableVerboseRadio, &QRadioButton::clicked, this, &
    MolQCASimulationSettings::RadioVerboseModeToggled);
}

void MolQCASimulationSettings::RadioVerboseModeToggled()
{
    if(m_enableVerboseRadio.isChecked())
    {
        m_is_debug = true;
    }else if(m_disableVerboseRadio.isChecked())
    {
        m_is_debug = false;
    }
}
```

**Listing 3.14:** Verbose mode selection.

The last section (Listing 3.15) of the settings window allows the user to choose the desired fields to be stored in the output table file. Not only can individual items be selected, but the checkboxes also allow to choose which of the clock signals to save, as well as the inputs and outputs. The molecular clock consists of at most four phases, as opposed to iNML's three-phase clock: therefore, if the selected circuit has four clock zones, a selection box for it appears. Since the molecular simulation does not explicitly handle time units, but instead works with a discrete number of time instants, the QSS step and table step (which are customizable in the iNML case) are also fixed to 1 *ns*.

```
void MolQCASimulationSettings::createOutputParamBox(int clock_zones)
{
    m_GroupOutputParam.setTitle("Output parameters");

    // GroupBox elements initialization
    QRegExp rx_list("(?:\\d+\\-\\d+,?|\\d+[a-z]?,?)*");

    m_TableItemsLabel.setText("Table items:");
    m_TableItems.setValidator(new QRegExpValidator(rx_list));
    m_TableItems.setPlaceholderText("1,3,5-7");

    m_checkBox_ck1.setText("clock1");
```

```
    m_checkBox_ck1.setCheckState(Qt::CheckState::Checked);
    m_checkBox_ck2.setText("clock2");
    m_checkBox_ck2.setCheckState(Qt::CheckState::Checked);
    m_checkBox_ck3.setText("clock3");
    m_checkBox_ck3.setCheckState(Qt::CheckState::Checked);
    m_checkBox_ck4.setText("clock4");
    if(clock_zones > 3)
    {
        m_checkBox_ck4.setCheckState(Qt::CheckState::Checked);
    }
    else
    {
        m_checkBox_ck4.setDisabled(true);
    }
    m_checkBox_in.setText("inputs");
    m_checkBox_out.setText("outputs");
    m_hboxlayout_items.addWidget(&m_checkBox_ck1);
    m_hboxlayout_items.addWidget(&m_checkBox_ck2);
    m_hboxlayout_items.addWidget(&m_checkBox_ck3);
    m_hboxlayout_items.addWidget(&m_checkBox_ck4);
    m_hboxlayout_items.addWidget(&m_checkBox_in);
    m_hboxlayout_items.addWidget(&m_checkBox_out);

    m_GridLayoutOutputParam.addWidget(&m_TableItemsLabel,0,0);
    m_GridLayoutOutputParam.addWidget(&m_TableItems,0,1);
    m_GridLayoutOutputParam.addLayout(&m_hboxlayout_items,1,1);

    // GroupBox set layout
    m_GroupOutputParam.setLayout(&m_GridLayoutOutputParam);
}
```

**Listing 3.15:** Output parameters.

Once the user has decided all the parameters and presses the "OK" button, all of them are double checked for validity, providing an error in case of issues; otherwise, they are passed to all the structures that need them in other sections of the code, such as the simulator itself. In particular, the parameters are passed to the *SimulationData* class through individual set methods, so that they are stored and passed to the *MainWindow* class, and can be retrieved as necessary through the corresponding get methods before the simulation begins, and when the simulation controller is instantiated (Listing 3.16). The timing resolution is also fixed to 1 *ns*.

```
void MolQCASimulationSettings::updateParameters(SimulationData &simData)
{
    simData.setStopTime(m_simTimeDouble);
    simData.setClock(m_ClockAmplitudeDouble, m_TimeOnDouble, m_TimeOffDouble,
    m_TimeRiseDouble, m_TimeFallDouble);
    simData.setPhysicalParams(m_DampingFactorDouble, m_StabilityThresholdDouble);
    simData.setResolution(1e-9);
    simData.setQssStep(m_QssStepDouble);
    simData.setTableStep(m_TableStepDouble);
    simData.setTableItems(m_TableItemsList);
    simData.setVerboseMode(m_is_debug);
    simData.setInteractionRadius(m_InteractionRadiusDouble);
    simData.setLogicVoltageValues(m_VoltageLowDouble, m_VoltageHighDouble);
    simData.setMQCASimulationAlgorithm("physical");
    simData.setNumberIterations(1);
```

```
15      simData.setResultsFolderPath(AppController::Instance().GetResultsPath());
16  }
```

**Listing 3.16:** Updating the simulation parameters.

The final result is the simulation settings window shown in Figure 3.2. When the "OK" button is pressed, the *Actions* class instantiates either the behavioral iNML simulation controller, the physical iNML controller, or the molecular one; once the desired controller is instantiated, the *setParameters* function, which sets the parameters chosen by the user, and the *startSimulation* function, which launches and executes the actual simulation, are called sequentially (Listing 3.17). Their implementation, performed subsequently to additional changes related to both iNML and MolQCA, will be described later.

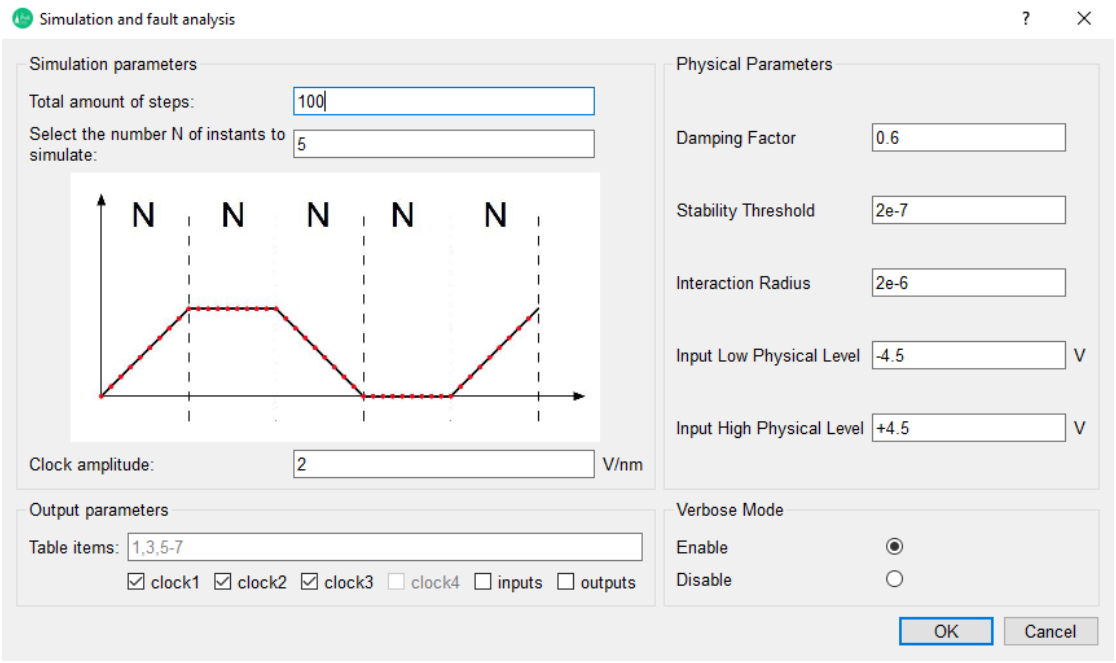



**Figure 3.2:** Simulation settings window.

```
1  if(sim_algorithm == 0) // behavioral
2  {
3      QElapsedTimer timer;
4      timer.start();
5      iNMLBehavioralSimulationController controller(layout_->getMQCAItemRoot()->
       allItems(), simulation_->getMQCAFloorplanParameters(), 1);
6      Logger::instance()->logMessage(QStringLiteral("The system generation took %1
       milliseconds").arg(timer.restart()));
7      controller.createTimeline(transition_list, sample_number);
8      controller.setParameters(simulation_data);
9      Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
       milliseconds").arg(timer.restart()));
```

```
10      controller.startSimulation(layout_qll, "Description", Format::Text,
        table_values, outputPath);
11      Logger::instance()->logMessage(QStringLiteral("The simulation took %1
        milliseconds").arg(timer.elapsed()));
12  }else if(sim_algorithm == 1) // physical
13  {
14      QElapsedTimer timer;
15      timer.start();
16      if(AppController::Instance().GetTechnologyNode() == "iNML")
17      {
18          iNMLSimulationController controller(layout_->getMQCAItemRoot()->allItems()
        , simulation_->getMQCAFloorplanParameters(), 2e-6) ;
19          Logger::instance()->logMessage(QStringLiteral("The system generation took
        %1 milliseconds").arg(timer.restart()));
20          controller.createTimeline(transition_list, sample_number);
21          controller.setParameters(simulation_data);
22          Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
        milliseconds").arg(timer.restart()));
23          controller.startSimulation(layout_qll, "Description", Format::Text,
        table_values, outputPath);
24          Logger::instance()->logMessage(QStringLiteral("The simulation took %1
        milliseconds").arg(timer.elapsed()));
25      } else if(AppController::Instance().GetTechnologyNode() == "MolQCA")
26      {
27          MolecularSimulationController molcontroller(layout_->getMQCAItemRoot()->
        allItems(), simulation_->getMQCAFloorplanParameters(), simulation_data.
        getInteractionRadius()) ;
28          Logger::instance()->logMessage(QStringLiteral("The system generation took
        %1 milliseconds").arg(timer.restart()));
29          molcontroller.createTimeline(transition_list, sample_number);
30          molcontroller.setParameters(simulation_data);
31          Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
        milliseconds").arg(timer.restart()));
32          molcontroller.startSimulation(layout_qll, "Description", Format::Text,
        table_values, outputPath);
33          Logger::instance()->logMessage(QStringLiteral("The simulation took %1
        milliseconds").arg(timer.elapsed()));
34      }
35  }
```

**Listing 3.17:** Simulation controller instantiation.

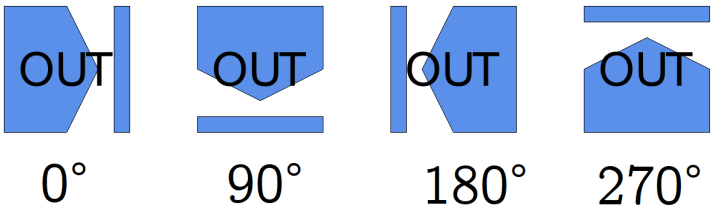
### 3.2.2 Preliminary Simulation Management

Before the already existing but unimplemented molecular simulation controller could be modified, corrected and activated, a few more preliminary steps were required. In order to analyze the behavior of the molecules during the simulation, their transcharacteristics must be known. To do so, some extra files containing those details must be imported into ToPoliNano in the same way the molecular plugins were; they are three text files which list the values of the charges in the Bisferrocene molecule depending on the value of the clock they are subjected to. In order to include these files, they are added to the project (Listing 3.18). In this way, when the molecular simulation controller is launched, the molecules can be read and their transcharacteristics imported.

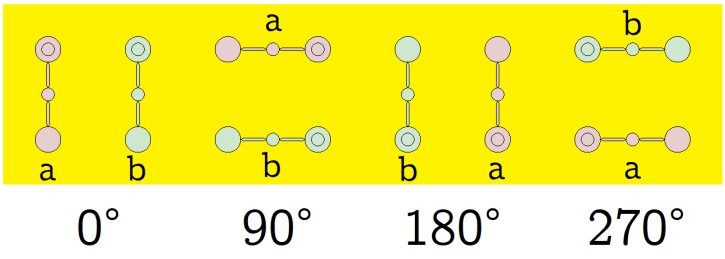
```
MOLQCA_TRANSC_SOURCE_PATH = $$DESTDIR/../libs/tech/molqca_001/plugin/
    transcharacteristics/*
MOLQCA_TRANSC_DEST_PATH = $$DESTDIR/tech/molqca_001/plugin
molqcaCharacteristicsCopy.commands = $(COPY_FILE) \"$$shell_path(
    $$MOLQCA_TRANSC_SOURCE_PATH)\" \"$$shell_path($$MOLQCA_TRANSC_DEST_PATH)\"
```

**Listing 3.18:** Importing the molecular transcharacteristics.

The next required feature to ensure that the simulation works correctly is related to checking that the pin placement in the layout is valid. MagCAD doesn't perform a pin orientation check, which is why ToPoliNano must be able to determine if the placement is valid: if not, some corrective action should be performed. MagCAD allows to freely select the placement of magnets and molecules and the pins connected to them; concerning the output pins, the only constraint is that the pin has its left side connected to a circuit element, otherwise the pin is disconnected altogether. In iNML technology, only the pin angle is evaluated, because the pin can be attached to any side of the magnet. Instead, there is a more complex distinction to be made when considering the molecular technology: for the output pin, we could distinguish between valid and invalid cases. Both the output pins and the molecules can be placed in four different angles: 0°, 90°, 180° and 270° (Figure 3.3 shows all possible orientations).




**(a)**




**(b)**

**Figure 3.3:** (a) Possible pin orientations. (b) Possible molecule orientations.

As mentioned earlier, each MolQCA cell contains two molecules, that can be referred to as molecule A and molecule B. As seen above, the standard orientation consists in molecule A being on the left side and molecule B on the right side. The output pin that is attached to a cell must be placed on a side where the pin is adjacent to either molecule A or molecule B: these cases are considered valid, as only one of the two molecules is influencing the output. However, in the two remaining sides, where the output would be influenced by both molecules, since the pin is exposed to both, it wouldn't be possible to determine which molecule of the cell is responsible for the interaction, leading to an unknown output value.

For this reason, all possible combinations of adjacency between output pin and molecule are considered: in this way we can determine whether the behavior is due to the interaction with molecule A, molecule B, or the behavior is unknown.

In order to handle all these cases inside ToPoliNano, the orientation of both the output pin and its adjacent cell must be determined, so as to evaluate which of the two molecules is responsible for the interaction. To perform this analysis and determine which cell to consider as output when the table file is generated, a new switch statement has been implemented (Listing 3.19). The *Actions* class contains the instantiation of the simulation controller and its functions for the execution of the simulation. Before launching it, the selected testbench must be parsed in order to obtain the applied input stimuli and the timing information. Another operation to be performed is related to the output table: depending on the selection made by the user in the simulation settings window, the corresponding elements of the circuit must be appended to a list, so that these items can be printed in the text file.

If the user had previously decided they would like the inputs to be shown, the layout is traversed in order to determine which are the input cells. In case the outputs must be printed, a more thorough analysis is required. The layout is parsed to find an output pin; once the pin has been found, the adjacent cell which is responsible for the output behavior must also be located. This search, however, depends on the selected technology: as previously mentioned, in case the iNML technology is being considered, only the pin angle is evaluated, since the pin can be attached on any side of a nanomagnet. If the working technology is Molecular QCA, both the pin and output cell orientation have to be determined, as different cases lead to different molecules influencing the output, or even to an undefined outcome.

```
1 switch(item->angle()){
2     case 0: // pos().x+1
3         foreach(QcaItem* find_item, layout_->getMQCAItemRoot()->allItems()){
4             if(find_item->pos().x()+1 == item->pos().x() && find_item->pos().y()
      == item->pos().y()){
5                 if(AppController::Instance().GetTechnologyNode()=="iNML"){
6                     table_values.append(QString::number(find_item->id()));
```

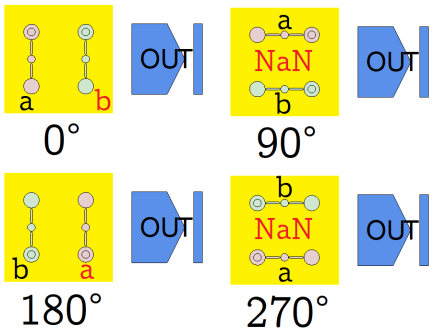
```
                }else if(AppController::Instance().GetTechnologyNode()=="MolQCA"){
                    if(find_item->angle() == 0){
                        table_values.append(QString::number(find_item->id())+"b");
                    }else if(find_item->angle()==90 || find_item->angle()==270){
                        table_values.append("invalid_output");
                    }else if(find_item->angle() == 180){
                        table_values.append(QString::number(find_item->id())+"a");
                    }
                }
            }
        }
        break;
    case 90: // pos().y+1
        foreach(QcaItem* find_item, layout_->getMQCAItemRoot()->allItems()){
            if(find_item->pos().x() == item->pos().x() && find_item->pos().y()+1
    == item->pos().y()){
                if(AppController::Instance().GetTechnologyNode()=="iNML"){
                    table_values.append(QString::number(find_item->id()));
                }else if(AppController::Instance().GetTechnologyNode()=="MolQCA"){
                    if(find_item->angle() == 90){
                        table_values.append(QString::number(find_item->id())+"b");
                    }else if(find_item->angle()==0 || find_item->angle()==180){
                        table_values.append("invalid_output");
                    }else if(find_item->angle() == 270){
                        table_values.append(QString::number(find_item->id())+"a");
                    }
                }
            }
        }
        break;
    case 180: // pos().x-1
        foreach(QcaItem* find_item, layout_->getMQCAItemRoot()->allItems()){
            if(find_item->pos().x()-1 == item->pos().x() && find_item->pos().y()
    == item->pos().y()){
                if(AppController::Instance().GetTechnologyNode()=="iNML"){
                    table_values.append(QString::number(find_item->id()));
                }else if(AppController::Instance().GetTechnologyNode()=="MolQCA"){
                    if(find_item->angle() == 180){
                        table_values.append(QString::number(find_item->id())+"b");
                    }else if(find_item->angle()==90 || find_item->angle()==270){
                        table_values.append("invalid_output");
                    }else if(find_item->angle() == 0){
                        table_values.append(QString::number(find_item->id())+"a");
                    }
                }
            }
        }
        break;
    case 270: // pos().y-1
        foreach(QcaItem* find_item, layout_->getMQCAItemRoot()->allItems()){
            if(find_item->pos().x() == item->pos().x() && find_item->pos().y()-1
    == item->pos().y()){
                if(AppController::Instance().GetTechnologyNode()=="iNML"){
                    table_values.append(QString::number(find_item->id()));
                }else if(AppController::Instance().GetTechnologyNode()=="MolQCA"){
                    if(find_item->angle() == 270){
                        table_values.append(QString::number(find_item->id())+"b");
                    }else if(find_item->angle()==0 || find_item->angle()==180){
                        table_values.append("invalid_output");
                    }else if(find_item->angle() == 90){
                        table_values.append(QString::number(find_item->id())+"a");
```

```
65                          }
66                      }
67                  }
68              }
69              break;
70          default:
71              break;
72  }
```

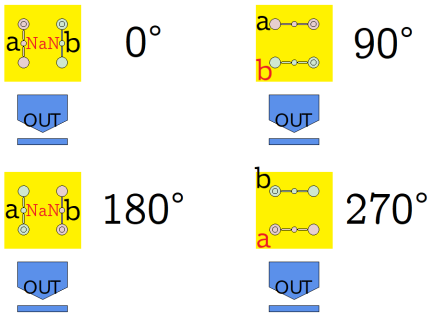
**Listing 3.19:** Analysis of all possible output pin and cell angle combinations.

The position of the elements on the circuit layout are treated as if they were on a cartesian plane, with X and Y coordinates. In order to determine where the output cell is located with respect to the pin, the latter's orientation must be considered first, hence why the cases of the switch statement refer to the pin angle. Depending on said angle, the relative position of the output cell could be on the right, left, above or below the pin. Naturally, in any of the possible cases, the cell is feeding the pin, meaning it would always be on its natural "left" so that the signal flows into the pin. Inside each case of the switch statement, the code contains a *for* loop that iterates over the elements in the circuit, looking for the output cell. Considering as an example a pin with angle equal to 0°, the QCA Item is the desired one if the pin is on the right of the cell ($pos().x() + 1$); if the pin orientation is 90°, the cell's vertical position incremented by one must coincide with the pin position, meaning that the pin is above the cell ($pos().y() + 1$), and so forth.

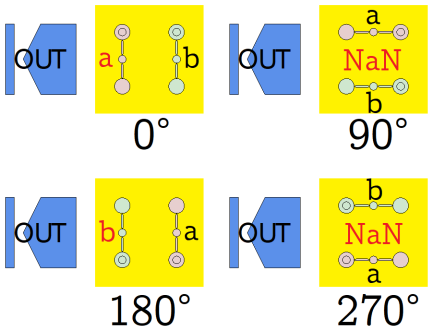
Once the cell has been located, if ToPoliNano is working with iNML technology, the cell is simply added to the list of values to print in the output table; in case of Molecular QCA, the orientation of the cell must be analyzed, to determine which of its molecules is influencing the output. There are three possible outcomes following this analysis: either molecule A is adjacent to the pin, molecule B is adjacent, or both of them are. The last case is the invalid one, and it is handled by adding to the table a fictional element called *invalid_output* instead of the cell name. As it will be shown later, if a table element has this name, the output is declared as NaN (Not a Number), because it cannot be inferred which of the two molecules is responsible for the output behavior. Figure 3.4 shows all possible combinations; the molecules with the name in red are the ones responsible for the interaction in that case.



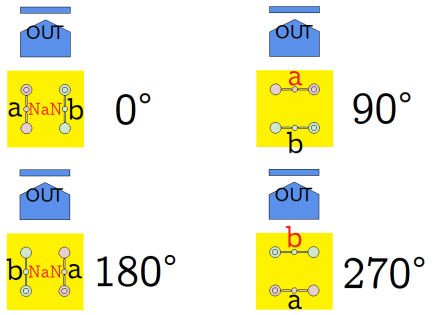

**(a)** Combinations for pin angle 0°.




**(b)** Combinations for pin angle 90°.




**(c)** Combinations for pin angle 180°.




**(d)** Combinations for pin angle 270°.

**Figure 3.4:** All possible pin-cell combinations.

Having established which elements to include in the table, managing how the text file itself is printed is the following step. This functionality is handled inside the *Data Manager* class, which is also responsible for the generation of the QSS files. The part of the class that handles the generation of the table file has been extensively modified in order to support the implementation of MolQCA simulations and the addition of the previously mentioned *invalid_output*. The structure of the text file has also been revamped in preparation for its management in the new Waveform Viewer (see Chapter 4).

The generation of the column titles and the print of the values are handled separately. The names of the elements that the user has chosen to display are used as titles; after they have been fetched, they are printed according to how many elements each signal contains and the technology type. For instance, in iNML technology, each clock element is composed of three magnetization values, along the X, Y and Z axes. This means that every clock phase has three columns in total:

*clock.X*, *clock.Y* and *clock.Z*. The same three magnetization values are printed for each magnet, so the size of each value is always three columns.

Instead, each clock phase in the MolQCA implementation has only one value, namely its amplitude expressed in $V/nm$, therefore occupying just one column per clock phase. The cell values are however four per cell, because each one is made of four charge dots (A, B, C and D). In addition, when handling MolQCA simulations there is also the possibility of encountering the aforementioned *invalid_output*. All the possible cases have been handled in Listing 3.20.

```
QFile file(path_name_ + "table.txt");
if (file.open(QIODevice::WriteOnly))
{
    QTextStream stream(&file);
    stream.setFieldAlignment(QTextStream::AlignLeft);
    stream.setFieldWidth(1);
    stream << "#time";
    for(auto value : table_fields_)
    {
        if(AppController::Instance().GetTechnologyNode() == "iNML")
        {
            for (int i=0; i<valuedim; ++i)
            {
                stream << "," + value + "." + valuelabels_short.split(' ')[i];
            }
        } else if(AppController::Instance().GetTechnologyNode() == "MolQCA")
        {
            if(value == "clock1" || value == "clock2" || value == "clock3" ||
    value == "clock4" || value == "invalid_output")
            {
                stream << "," + value;
            } else
            {
                for (int i=0; i<valuedim; ++i)
                {
                    stream << "," + value + "." + valuelabels_short.split(' ')[i];
                }
            }
        }
    }
    file.close();
}
```

**Listing 3.20:** Printing the titles of the table columns.

The previously described distinctions have been made: if iNML is the active technology, each field is split in different columns corresponding to the magnetization along the X, Y and Z axes; if MolQCA is selected, the clock signals and the *invalid_output* all possess one value each, while the molecules have multiple columns and each one is iterated on. Another new addition is how the table is formatted: previously, all the columns were separated by enough spaces to have a very ordered view of each value. With the implementation of a program to visualize the table values as waveforms, the table does not need to be opened as a simple text file anymore. Therefore, a more practical solution has been achieved by separating the columns with commas instead of aligning them with multiple

spaces, for an optimal management when parsing the table files in the Waveform Viewer described in Chapter 4.

Having handled the titles, the next function to be modified reflecting the new changes is *tableWriteOnFile*(), responsible for printing the actual values of the table elements (Listing 3.21). This function is called regularly by the simulator, which sends the values to a buffer table until it's full; when this happens, the values are written to the table, and the buffer is emptied and fed again with new values.

```
void DataManager::tableWriteOnFile(){
    QFile file(path_name_ + "table.txt");
    if(file.open(QIODevice::Append)){
        QTextStream stream(&file);
        stream.setFieldAlignment(QTextStream::AlignLeft);
        for (int i=0; i <= index_table_; ++i, table_step_counter+=step_table_){
            stream << qSetFieldWidth(0) << "\n" << qSetFieldWidth(1) <<
    buffer_table_[i]["CurrentTime"]["CurrentTime"] << ",";
            int j = 0;
            for(QString value : table_fields_){
                if(buffer_table_[i].contains(value)){
                    int index = 0;
                    int max = buffer_table_[i][value].values().size();
                    for(double value : buffer_table_[i][value].values()){
                        if((j == table_fields_.size() - 1) && (index == max - 1)){
                            stream << QString::number(value);
                        } else{
                            stream << QString::number(value) + ",";
                        }
                        index++;
                    }
                } else if(value == "invalid_output"){
                    if(j == table_fields_.size() - 1){
                        stream << "NaN";
                    }else{
                        stream << "NaN,";
                    }
                }else{
                    for(int i=0; i<valuedim_; ++i){
                        if((j==table_fields_.size() - 1) && (i==valuedim_ - 1)){
                            stream << "";
                        } else{
                            stream << ",";
                        }
                    }
                }
                j++;
            }
        }
        file.close();
    }
}
```

**Listing 3.21:** Printing the table values.

The first element of each row is the simulation step. The current time was initially computed at every iteration of this function; it has been updated to simply obtain the simulation time stored in the buffer table, for convenience purposes.

The program loops at every value found in the data structure before moving on to the next simulation step, and a distinction is made based on the value that is being read and on its position in the row. If a value is found in the current element, the iteration occurs on all its values: for instance, if a clock value is found, all three magnetization values are printed, separated by a comma. If the element to be printed is the last in the row, no comma at the end should be printed.

If the current field is an *invalid_output*, due to a molecular layout in which the output is in the undefined case, the printed value is $NaN$, with or without a comma based on its position in the current row. The last case is the possibility of an empty element, which may occur when the user would like to read the value of a clock signal that is not present in the circuit, *i.e.* trying to print *clock*2 and *clock*3 when only one clock zone is available. When this possibility occurs, an empty element is printed by simply writing a comma to the output stream, or an empty character if the end of the row has been reached.

In order to effectively implement the choices of the new parameters in the previously described *MolQCASimulationSettings* class, all the necessary get and set methods have been implemented. Inside class *SimulationData*, responsible for managing through various functions the retrieval and passing of the simulation parameters, new methods (Listing 3.22) have been introduced for the selection of the verbose mode, damping factor, stability threshold, interaction radius and input high and low physical levels; all of these parameters are necessary to perform the molecular simulation.

```
1  void   setVerboseMode(bool is_debug);
2  bool   getVerboseMode();
3  void   setPhysicalParams(double damping_factor, double stability_threshold);
4  double getDampingFactor();
5  double getStabilityThreshold();
6  void   setInteractionRadius(double interaction_radius);
7  double getInteractionRadius();
8  void   setLogicVoltageValues(double voltage_low, double voltage_high);
9  double getVoltageLow();
10 double getVoltageHigh();
```

**Listing 3.22:** List of new get and set methods.

The actual usage of these methods will be tackled when discussing the execution of the molecular simulation. However, in regards to the low and high voltage levels, these parameters have been added to the molecular simulation controller in order to set the value of the input charge to either the high or the low level depending on whether its logic value is '1' or '0' respectively (Listing 3.23).

```
1 double input = (value["Logic"] == 1) ? voltage_high_ : voltage_low_;
2 QVector<double> state = molecules.value(system_.at(*node_it.first)->
      getMoleculeName()).getChargesValue(input, 2);
```

**Listing 3.23:** Assigning the value of the input charge depending on the logic level.

## 3.3 Improving the Propagation of Parameters

An additional improvement to the propagation of the simulation parameters through the ToPoliNano classes would allow for a further optimization of the code. To achieve this, a new data structure has been implemented. The new type, called $MQCAFloorplanParameters$, is a $QHash < QString, QVariant >$: this data structure is a hash, it can contain multiple entries composed of a key, which is an identifier of the value (in this case, a string containing the name of the parameter), and a value. The value in this particular hash is of type QVariant, which allows to have parameters of different types (integer, double and more), depending on which is necessary.

The layout files in QLL format are xml files that are elaborated by ToPoliNano to obtain a graphical representation. Inside these files, the position and characteristics of pins and cells are listed, as well as the technological parameters (see example in Figure 3.5).

```
<settings tech="iNML">
    <property name="layersEnabled" value="false"/>
    <property name="DWMaxSize" value="6"/>
    <property name="Layoutwidth" value="4"/>
    <property name="Thickness" value="10"/>
    <property name="Height" value="90"/>
    <property name="HDistance" value="25"/>
    <property name="DWDefSize" value="2"/>
    <property name="DWEnabled" value="false"/>
    <property name="CZSequence" value="4"/>
    <property name="Width" value="60"/>
    <property name="Layoutheight" value="5"/>
    <property name="PhaseNumber" value="3"/>
    <property name="VDistance" value="25"/>
    <property name="InterlayerSpace" value="10"/>
</settings>
```

**Figure 3.5:** Technology settings in an iNML layout.

In order to elaborate the circuit, the parameters must be read directly from the selected QLL file and stored as data. After these parameters have been read, the new method shown in Listing 3.24 inserts them in the previously mentioned QHash. The saved values can then be retrieved by the active simulation controller, whether it is the iNML behavioral, iNML physical or molecular one, and thus used in the execution of the simulation: this is possible because the function is technology-independent, and only the parameters required for the currently active technology will be retrieved.

```
1 void SimulationData::setMQCAFloorplanParameters( NMLLayoutOptParameters& OptParam,
      int width, int height)
2 {
3     m_MQCAFloorplanParameters.insert("maxVertMag", OptParam.
      maximum_vertical_magnets);
```

```
4     m_MQCAFloorplanParameters.insert("CZSequence",OptParam.magnets_per_clockzone);
5     m_MQCAFloorplanParameters.insert("PhaseNumber", OptParam.phaseNumber);
6     m_MQCAFloorplanParameters.insert("Height", OptParam.magnet_height);
7     m_MQCAFloorplanParameters.insert("Width", OptParam.magnet_width);
8     m_MQCAFloorplanParameters.insert("Thickness", OptParam.magnet_thickness);
9     m_MQCAFloorplanParameters.insert("HDistance", OptParam.magnet_horizontal_space
      );
10    m_MQCAFloorplanParameters.insert("VDistance", OptParam.magnet_vertical_space);
11    m_MQCAFloorplanParameters.insert("InterlayerSpace", OptParam.
      magnet_layer_space);
12    m_MQCAFloorplanParameters.insert("numMagPhase1", OptParam.
      magnets_per_clockzone);
13    m_MQCAFloorplanParameters.insert("numMagPhase2and3", OptParam.
      magnets_per_clockzone);
14    m_MQCAFloorplanParameters.insert("numRows", 1);
15    m_MQCAFloorplanParameters.insert("numPhaseBlocksPerRow", width / OptParam.
      magnets_per_clockzone);
16    m_MQCAFloorplanParameters.insert("Layoutheight", height);
17    m_MQCAFloorplanParameters.insert("Layoutwidth", width);
18 }
```

**Listing 3.24:** The method that stores the layout parameters.

These are only some of the layout parameters used in the simulation controllers, and they are passed through the corresponding get method to the desired controller in its constructor, when it is called. This occurs in the *Actions* class when the simulation is requested, after the layout parameter has been read and the output pin and cell orientations have been determined. Depending on the chosen technology, and on the algorithm if iNML is active, one of three controllers can be called: the iNML behavioral controller, the iNML physical controller or the molecular simulation controller (Listing 3.25). Once the desired controller has been instantiated, more methods are executed to create the timeline, set all the simulation parameters, and finally execute the requested simulation. A timer is also used to send information to the user through the command log, regarding the elapsed time for the system generation, the timeline generation and the total simulation time.

```
1  if(sim_algorithm == 0){ // behavioral
2     QElapsedTimer timer;
3     timer.start();
4     iNMLBehavioralSimulationController controller(layout_->getMQCAItemRoot()->
      allItems(), simulation_->getMQCAFloorplanParameters(), 1);
5     Logger::instance()->logMessage(QStringLiteral("The system generation took %1
      milliseconds").arg(timer.restart()));
6     controller.createTimeline(transition_list, sample_number);
7     controller.setParameters(simulation_data);
8     Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
      milliseconds").arg(timer.restart()));
9     controller.startSimulation(layout_qll, "Description", Format::Text,
      table_values, outputPath);
10    Logger::instance()->logMessage(QStringLiteral("The simulation took %1
      milliseconds").arg(timer.elapsed()));
11 }else if(sim_algorithm == 1){ // physical
12    QElapsedTimer timer;
13    timer.start();
14    if(AppController::Instance().GetTechnologyNode() == "iNML") {
```

```
15          iNMLSimulationController controller(layout_->getMQCAItemRoot()->allItems()
    , simulation_->getMQCAFloorplanParameters(), 2e-6) ;
16          Logger::instance()->logMessage(QStringLiteral("The system generation took
    %1 milliseconds").arg(timer.restart()));
17          controller.createTimeline(transition_list, sample_number);
18          controller.setParameters(simulation_data);
19          Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
    milliseconds").arg(timer.restart()));
20          controller.startSimulation(layout_qll, "Description", Format::Text,
    table_values, outputPath);
21          Logger::instance()->logMessage(QStringLiteral("The simulation took %1
    milliseconds").arg(timer.elapsed()));
22      } else if(AppController::Instance().GetTechnologyNode() == "MolQCA") {
23          MolecularSimulationController molcontroller(layout_->getMQCAItemRoot()->
    allItems(), simulation_->getMQCAFloorplanParameters(), simulation_data.
    getInteractionRadius()) ;
24          Logger::instance()->logMessage(QStringLiteral("The system generation took
    %1 milliseconds").arg(timer.restart()));
25          molcontroller.createTimeline(transition_list, sample_number);
26          molcontroller.setParameters(simulation_data);
27          Logger::instance()->logMessage(QStringLiteral("The timeline generation %1
    milliseconds").arg(timer.restart()));
28          molcontroller.startSimulation(layout_qll, "Description", Format::Text,
    table_values, outputPath);
29          Logger::instance()->logMessage(QStringLiteral("The simulation took %1
    milliseconds").arg(timer.elapsed()));
30      }
31  }
```

**Listing 3.25:** Simulation execution.

The constructor receives all the layout properties and circuit parameters, so they can be used during the system computations. As an example, the different parameters for the physical iNML simulation (Listing 3.26) are shown below. These include spatial information, namely distances between elements on the X and Y axes, the number of clock zones, how many magnets are placed at most in each clock zone and the total width and height of the circuit layout; information about the magnet dimensions are included as well: width, height and thickness.

```
1  physical_parameters_.x_distance = settings.value("HDistance").toDouble() * 1e-9;
2  physical_parameters_.y_distance = settings.value("VDistance").toDouble() * 1e-9;
3  physical_parameters_.z_distance = settings.value("InterlayerSpace").toDouble() * 1
       e-9;
4  physical_parameters_.x_axis = settings.value("Width").toDouble() * 1e-9;
5  physical_parameters_.y_axis = settings.value("Height").toDouble() * 1e-9;
6  physical_parameters_.z_axis = settings.value("Thickness").toDouble() * 1e-9;
7  physical_parameters_.offset = 200e-9;
8  physical_parameters_.clock_zone_number = settings.value("PhaseNumber").toInt();
9  physical_parameters_.magnet_per_clock_zone = settings.value("CZSequence").toInt();
10 physical_parameters_.layout_width = settings.value("Layoutwidth").toDouble();
11 physical_parameters_.layout_height = settings.value("Layoutheight").toDouble();
12 physical_parameters_.grid_x_size = physical_parameters_.x_distance +
       physical_parameters_.x_axis;
13 physical_parameters_.grid_y_size = physical_parameters_.y_distance +
       physical_parameters_.y_axis;
14 physical_parameters_.grid_z_size = physical_parameters_.z_distance +
       physical_parameters_.z_axis;
```

**Listing 3.26:** List of parameters passed to the physical iNML controller.

A similar approach has been chosen for the molecular simulations (Listing 3.27): in this case, the information about the layout are the total width and height, the number of molecules per clock zone and the total amount of clock zones. Instead, the molecule parameters used in the calculations are the intermolecular distance, which is the distance between the two molecules inside a cell, and the cell distance, equal to double the intermolecular distance.

```
physical_parameters_.offset = settings.value("Intermolecular Distance").toDouble()
    * 1e-12;
physical_parameters_.cell_distance = physical_parameters_.offset * 2;
physical_parameters_.cz_number = settings.value("PhaseNumber").toInt();
physical_parameters_.molecule_per_cz = settings.value("CZSequence").toInt();
physical_parameters_.layoutWidth = settings.value("Layoutwidth").toDouble();
physical_parameters_.layoutHeight = settings.value("Layoutheight").toDouble();
```

**Listing 3.27:** List of physical parameters passed to the MolQCA controller.

The physical parameters just described are related to the layout information and to the selected technology. Before the simulation can be executed, however, the active controller must also be fed with the user-defined simulation parameters described in section 3.2.1. To do this, a new method called *setParameters* has been introduced.

The individual simulation controllers are all classes derived from the so-called *FcnsSimulationController*; the classes derived from it implement in different ways the virtual methods declared in the generic controller, and *setParameters* is among them. This is because, depending on the active technology, the simulation settings that can be selected by the user are very different, hence why it is more convenient to have the same method implemented in different ways. Another reason why this is possible is that the parameter passed to the function is the same *SimulationData* structure, containing different settings depending on the controller type. All the previously mentioned get and set methods for the simulation settings have been used inside *setParameters* to pass the values to the controllers.

For behavioral iNML simulations (Listing 3.28), the necessary simulation parameters are the timing resolution, the simulation step, the output steps for the QSS and table files and the total simulation time (*i.e.* the stop time). As far as the clock is concerned, due to the fact that this is not a physical simulation, the rise and fall times are ideal and set to zero, and the amplitude range is unitary; the only clock parameters are the on time and the off time.

```
void iNMLBehavioralSimulationController::setParameters(SimulationData
    simulation_data){
    simulation_step_ = simulation_data.getResolution().toDouble();
    data_manager_.setTime_step(simulation_step_);
    data_manager_.SetExportStep(simulation_data.getTableStep(), simulation_data.
    getQssStep());
    stop_time_ = simulation_data.getStopTime();
    min_amplitude_ = 0.0;
    max_amplitude_ = 1.0;
```

```
8     rise_time_ = 0.0;
9     on_time_ = simulation_data.getOnTime();
10    fall_time_ = 0.0;
11    off_time_ = simulation_data.getOffTime();
12 }
```

**Listing 3.28:** Behavioral iNML simulation settings.

Similarly, physical iNML simulations require the previously listed timing parameters, with the addition of the clock rise and fall times, because the clock is not an ideal one (Listing 3.29). The maximum amplitude is also retrieved from the user-selected value, while the minimum is set to zero.

```
1  void iNMLSimulationController::setParameters(SimulationData simulation_data){
2     simulation_step_ = simulation_data.getResolution().toDouble();
3     data_manager_.setTime_step(simulation_step_);
4     data_manager_.SetExportStep(simulation_data.getTableStep(), simulation_data.
      getQssStep());
5     stop_time_ = simulation_data.getStopTime();
6     min_amplitude_ = 0.0;
7     max_amplitude_ = simulation_data.getClockAmplitude();
8     rise_time_ = simulation_data.getRiseTime();
9     on_time_ = simulation_data.getOnTime();
10    fall_time_ = simulation_data.getFallTime();
11    off_time_ = simulation_data.getOffTime();
12 }
```

**Listing 3.29:** Physical iNML simulation settings.

The simulation type that requires the largest amount of parameters is the molecular one (Listing 3.30). In addition to the timing values mentioned for the iNML controllers, the clock signal is different in this case. Instead of having an amplitude from 0 to *Amplitude*, the molecular clock can reach negative values, hence why it is set from -*Amplitude* to +*Amplitude*.

The remaining user-selected parameters required for the MolQCA simulation are the already mentioned damping factor, stability threshold, verbose mode selection and input physical voltage levels.

```
1  void MolecularSimulationController::setParameters(SimulationData simulation_data){
2     damping_factor_ = simulation_data.getDampingFactor();
3     stability_threshold_ = simulation_data.getStabilityThreshold();
4     voltage_low_ = simulation_data.getVoltageLow();
5     voltage_high_ = simulation_data.getVoltageHigh();
6     is_debug_ = simulation_data.getVerboseMode();
7     simulation_step_ = simulation_data.getResolution().toDouble();
8     data_manager_.setTime_step(simulation_step_);
9     data_manager_.SetExportStep(simulation_data.getTableStep(), simulation_data.
      getQssStep());
10    stop_time_ = simulation_data.getStopTime();
11    min_amplitude_ = -(simulation_data.getClockAmplitude());
12    max_amplitude_ = simulation_data.getClockAmplitude();
13    rise_time_ = simulation_data.getRiseTime();
14    on_time_ = simulation_data.getOnTime();
15    fall_time_ = simulation_data.getFallTime();
16    off_time_ = simulation_data.getOffTime();
17 }
```

**Listing 3.30:** MolQCA simulation settings.

## 3.4 Molecular Time Steps Refactoring

The last modifications to ToPoliNano's simulator are related in particular to achieving a fully working molecular simulation. The first step consists in refactoring the way the simulator keeps track of the current time. As already mentioned during the description of Listing 3.21, for an ordered and more efficient time management, a new parameter has been added to the buffers containing the values to be printed in both the output QSS and table files. The previous implementation required the current time to be computed every time the output generating functions were accessed. Thanks to the addition of the new *CurrentTime* value directly in the simulation controllers, it is possible to simply add it to the buffer at every step; this allows for a simpler management, since the controller naturally uses and updates the current simulation time during its calculations.

For example, in the iNML simulations, every time the controller advances by one time step the current time is stored in a new data structure and, if that step should be stored for the use in the output files, the value is appended (Listing 3.31).

```
QPair<QString, Value> time;
time.first = "CurrentTime";
time.second = Value("CurrentTime", simulation_time_);
if(data_manager_.isToBeSaved()){
    data_manager_.appendElement(time);
}
```

**Listing 3.31:** Storing the current simulation time.

The new data structure is a $QPair < QString, Value >$. It is a struct that stores a pair of items: in this case, the first one is the name of the variable, "CurrentTime", while the second item is of type Value, defined inside ToPoliNano; similarly to a QHash, a variable of type Value consists of a QString that acts as a key (an identifier), and the associated value of type double.

The verbose mode for molecular simulations has also been handled in the following, enabling it to work as intended. As previously described, verbose mode introduces a much more detailed simulation execution, by iterating over time steps much smaller than the selected simulation step, called microsteps. When verbose mode is activated by the user, for every time step, time advances in multiples of the microstep, and at each of them the system is evaluated and the values are stored, to be printed in the output files. The condition that must be reached for the system to move on to the next step is either the stability of the system, which is related to the user-defined stability threshold, or the fact that 999 microsteps have passed. The reason for the latter is that the value of one microstep is equal to $0.001 \cdot simulation\_step$, and therefore after 1000 microsteps the next simulation step is reached. Listing 3.32 shows an excerpt from the *startSimulation* function,

which is called at the beginning of a simulation.

```
simulation_time_ = 0.0;
current_time_ = 0.0;
microstep_ = simulation_step_*1e-3;
while(simulation_time_ <= stop_time_)
{
    nmicrosteps_ = 0;
    current_time_ = simulation_time_;
    // loop while it is not stable
    do
    {
        evaluateInteraction();
        if(is_debug_)
        {
            nmicrosteps_++;
        }
    }while(!advanceStep());
    simulation_time_ += simulation_step_;
}
```

**Listing 3.32:** The simulation loop.

The simulation time, current time and microstep values are initialized. The simulation time is incremented by the step at every iteration of the loop. The current time is also updated with the simulation time, and its purpose is to be stored in the corresponding element of the buffer table used to save the output results. At every step, the interactions between all the molecular cells are evaluated. If verbose mode is active, which is symbolized by the boolean value of variable *is_debug_*, the number of microsteps elapsed in the current simulation step is also incremented. The inner do-while loop then keeps iterating until the return value of function *advanceStep* (Listing 3.33) is false: this means that the system is not stable yet, so the simulator continues evaluating the interactions between the circuit elements. When the system is stable, the simulation time is increased and a new iteration of the outer loop is performed. The exit condition occurs when the simulation time has reached the total requested time, stored in variable *stop_time_*.

```
bool MolecularSimulationController::advanceStep()
{
    bool isStable = true;
    Value difference;
    QHash<QString, Value> stepValues;
    int maxsteps = 999;
    for ( auto node_it = boost::vertices(system_.graph()); node_it.first !=
    node_it.second; ++node_it.first){
        auto node_desc = *node_it.first;
        QPair<QString, Value> diff = system_.at(node_desc)->updateValue(difference
    );
        stepValues[diff.first] = diff.second;
        // If one is above the threshold, the step is not stable.
        if((difference > stability_threshold_) && (nmicrosteps_ < maxsteps)){
            isStable = false;
        }
    }
    stepValues["CurrentTime"] = Value("CurrentTime", current_time_);
```

```
17      if(is_debug_){
18          current_time_ += microstep_;
19      }
20      if(is_debug_ || isStable){
21          for (auto ck : system_.clocks()){
22              stepValues[ck->getId()] = ck->getValue();
23          }
24          data_manager_.newTimeStep();
25          if(data_manager_.isToBeSaved()){
26              data_manager_.appendElements(stepValues);
27          }
28      }
29      return isStable;
30  }
```

**Listing 3.33:** Management of the simulation steps.

The *advanceStep* function is accessed multiple times during each simulation step, to determine if the system is stable. A QHash called $stepValues$ is used to store the cell and clock values, as well as the current time. In a single simulation step, if verbose mode has been selected, time can increase by microsteps until the maximum number, 999, is reached. If the stability threshold of the system has not yet been reached, and the number of elapsed microstep is still below the maximum, the stability is labeled as false. The most likely case is the system reaching stability before the total amount of microsteps is equal to the maximum.

The current simulation time is saved inside variable $stepValues$: then, if verbose mode is activated, the current time is incremented by a microstep; otherwise, the current time remains the same. The reason behind this is that, as mentioned earlier, the function is accessed multiple times per simulation step, and the circuit interactions are evaluated multiple times until the system is stable at that step. The system values are saved either when stability is reached, or at every microstep if verbose mode is on. A new time step is created in the data manager structures, necessary for both the QSS and the table files; then, if the buffers are full, the values are immediately appended so they can be printed.

The final required implementation to obtain a fully functional molecular simulation is a new four-phase clock system. Originally, all three types of simulation controller possessed the same function to generate the same clock, defined in the generic FCNS controller class. Function *createClock* is now defined as a virtual function in the FCNS interface, and implemented separately in each simulation controller: in this way, iNML and molecular simulations can have different clocks. Magnetic simulations have maintained the three-phase clock already documented in section 1.2.1, while the new four-phase clock for the molecular case has been added to ToPoliNano.

The new clock signal does not only add a new phase, but it is also slightly different from the iNML clock, particularly the third element. Figure 3.6 depicts the four-phase clock to be implemented inside ToPoliNano's molecular simulation

controller, with a standard range between -2 and +2 V/nm.

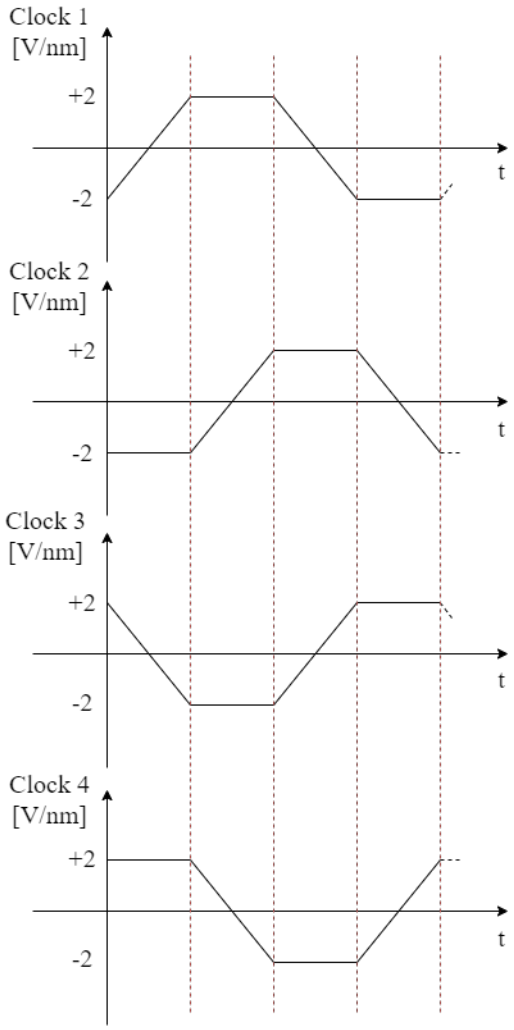



**Figure 3.6:** The four-phase clock for molecular simulations.

The function that defines and creates the clock, and inserts it in the simulation timeline, is called inside every controller's *startSimulation* function. The flow of the *createClock* function consists in first of all computing the steps when the clock reaches the on state, starts decreasing, and when it is in the off state. In addition, the number of steps per clock period, the time shifts between each clock element and the time step relative to the start of the period are also determined. A snippet of the *createClock* function is shown in Listing 3.34, where only the computation of the aforementioned values and the generation of the *Clock* 1 signal are shown.

```
1 QVector<QPair<QString, Level> > timeline_tmp;
2 // clock time segments
```

```
int startStepOn = rise_time_/simulation_step_; // the first step when the clock is
     on
int startStepFall = startStepOn + on_time_/simulation_step_; // the step at which
    the clock starts to go down
int startStepOff = startStepFall + fall_time_/simulation_step_; // the first step
    where the clock is off
int stepsPeriod = startStepOff + off_time_/simulation_step_; // number of steps
    per period
// clock shift
int shiftCk2 = startStepOn; // shift of the clock_2 wrt to the clock_1
int shiftCk3 = 0; // shift of the clock_3 wrt to the clock_1
int shiftCk4 = startStepOn; // shift of the clock_4 wrt to the clock_1
// calculate the time step relative to the start of the period
int relativeStep_1 = timeStep % stepsPeriod;
int relativeStep_2 = (timeStep+stepsPeriod-shiftCk2) % stepsPeriod;
int relativeStep_3 = (timeStep+stepsPeriod-shiftCk3) % stepsPeriod;
int relativeStep_4 = (timeStep+stepsPeriod-shiftCk4) % stepsPeriod;

Level level_ck;
// CLOCK 1
if(relativeStep_1 < startStepOn){
    level_ck.value_["Logic"] = min_amplitude_ + ((max_amplitude_-min_amplitude_) /
     startStepOn) * relativeStep_1;
    level_ck.start_ = timeStep  * simulation_step_;
    level_ck.stop_ = (timeStep+1) * simulation_step_;
    timeline_tmp.append(QPair<QString, Level>("clock1", level_ck));
}else if(relativeStep_1 == startStepOn){
    level_ck.value_["Logic"] = min_amplitude_ + (max_amplitude_-min_amplitude_);
    level_ck.start_ = timeStep  * simulation_step_;
    level_ck.stop_ = (timeStep+startStepFall - startStepOn) * simulation_step_;
    timeline_tmp.append(QPair<QString, Level>("clock1", level_ck));
}else if(relativeStep_1 >= startStepFall && relativeStep_1 < startStepOff){
    level_ck.value_["Logic"] = min_amplitude_ + (max_amplitude_-min_amplitude_) -
    (((max_amplitude_-min_amplitude_) / (startStepOff - startStepFall)) * (
    relativeStep_1 - startStepFall));
    level_ck.start_ = timeStep  * simulation_step_;
    level_ck.stop_ = (timeStep+1) * simulation_step_;
    timeline_tmp.append(QPair<QString, Level>("clock1", level_ck));
}else if(relativeStep_1 == startStepOff){
    level_ck.value_["Logic"] = min_amplitude_;
    level_ck.start_ = timeStep  * simulation_step_;
    level_ck.stop_ = (timeStep + stepsPeriod - startStepOff) * simulation_step_;
    timeline_tmp.append(QPair<QString, Level>("clock1", level_ck));
}
```

**Listing 3.34:** The molecular createClock() function.

As shown above only for the first clock phase, depending on where the current step falls (during the rise time, on time, fall time or off time), the amplitude value and the start and stop time for that step are computed. At the end, the clock is appended to the timeline, and the function moves on to the next clock phase. Once all the clock signals have been defined and generated, the physical simulation of the molecules can be performed.

# Chapter 4

# Waveform Viewer

The results of ToPoliNano simulations consist in a series of QSS files, each one containing the values of all cells in a layout at a certain time instant, and a table file, in which the values previously selected by the user, including the clock signals, the input cells and the output cells, are stored in a table where every row corresponds to a different simulation time step.

Thanks to the FCNS Viewer previously described in section 2.3, individual or complete folders of QSS files can be visualized in 3D, graphically showing the magnetizations or the charge values (depending on the selected technology) with a color-coded representation of the circuit elements. The feature that would complete the ToPoliNano Framework is one capable of representing the values of all the signals stored in a table file as waveforms, similar to what commercial simulation tools for CMOS technologies include. Thanks to the new implementation featured in this chapter, this task is now possible inside FCNS Viewer.

## 4.1 Opening and Parsing a Table File

The interface of FCNS Viewer allows the user to either open an individual QSS file, or an entire folder so as to visualize a full simulation timeline. By adding the possibility of selecting a table file, the first step of this implementation can be achieved. The Window class instantiates the main window and all its menu features, namely the 3D viewer section and the drop-down menus. A new voice has been added under the "File" header, allowing the user to search for a table file to open (Listing 4.1).

```
1 actionTable = new QAction(this);
2 actionTable->setObjectName(QString::fromUtf8("actionTable"));
3 menuFile->addAction(actionTable);
4
```

```
connect(actionTable, SIGNAL(triggered()), this, SLOT(openTable()));

actionTable->setText(QApplication::translate("Window", "Open Table", nullptr));
#ifndef QT_NO_SHORTCUT
    actionTable->setShortcut(QApplication::translate("Window", "Ctrl+T",nullptr));
#endif // QT_NO_SHORTCUT
```

**Listing 4.1:** Creating the Open Table menu option.

Therefore a new menu action is created, and its activation triggers a new function called *openTable*, shown in Listing 4.2; as the code above demonstrates, this feature can also be accessed through the $CTRL + T$ shortcut.

```
void Window::openTable()
{
    table_elements.clear();
    error_occurred = false;
    QString name;
    name = QFileDialog::getOpenFileName(this, tr("Open Table"), lastOpenedLocation
    .path(), tr("Normal Text Files (*.txt)"));
    if (name != "")
    {
        lastOpenedLocation = QDir(name.at(0));
        tablename = name;
        parseTable();
        if(error_occurred == true)
        {
            return;
        }
        nSignals = table_elements[0].size() - 1;
        // Create Simulation Subwindow
        SimulationSubwindow* tab = new SimulationSubwindow(this);
        tab->setAttribute(Qt::WA_DeleteOnClose);
        simulationSubwindow.setCentralWidget(tab);
        simulationSubwindow.resize(1280, 720);
        tab->createGraphs(nSignals);
        double max = table_elements[table_elements.size()-1][0].toDouble();
        double table_step = table_elements[2][0].toDouble(); // table_elements[i
    ][0] contains the timesteps. i = 2 is the second timestep, whereas i = 1 is t
    = 0. So index i = 2 gives us the table step.
        tab->setTime(max, table_step);
        tab->plotSignals(table_elements);
        simulationSubwindow.setWindowTitle("Waveform Viewer");
        simulationSubwindow.show();
        tab->init();
    }
}
```

**Listing 4.2:** The openTable() function.

When the *openTable* function is accessed, a new QFileDialog is activated, letting the user search and select a table to open from the file system. Once the table has been selected, if the name of the file is valid, its content can be parsed thanks to function *parseTable*, whose contents are present in Listing 4.3. If no error has occurred in this process, the function can move on, otherwise the software returns to the main window. In the former case, the total number of signals can be obtained by reading how many elements are present in the first row, which is the

row containing the names of the signals.

An instance of the new class called *SimulationSubwindow* can now be generated. This class opens a new subwindow of the FCNS Viewer software, in which the waveforms are represented. Its functions will be described in detail later: *openTable* passes to the class the number of signals, the total simulation time (by reading the last element of the time column, which is the final time instant) and the time step of the table file, and the matrix containing all the elements parsed from the table. To conclude, the new subwindow is opened.

```
void Window::parseTable(){
    QFile file(tablename);
    if (file.open(QIODevice::ReadOnly)){
        QTextStream instream(&file);
        QString firstline = instream.readLine();
        QStringList titlefields = firstline.split(",");
        if(titlefields[0] != "#time"){
            error_occurred = true;
            QMessageBox::critical(this, "An error has occurred", tr("Operation
    aborted: no time column found in\n%1").arg(tablename), QMessageBox::Close);
            return;
        }
        QVector<QString> tmp;
        for(int i=0; i < titlefields.size(); i++){
            tmp.push_back(titlefields[i]);
        }
        table_elements.push_back(tmp);
        QVector<bool> errors(titlefields.size(), false);
        while(!instream.atEnd()){
            QString line = instream.readLine();
            QStringList fields = line.split(",");
            if((fields.size() != titlefields.size()) || (fields.size() == 1)){ //
    Mismatch in amount of table elements
                error_occurred = true;
                QMessageBox::critical(this, "An error has occurred", tr("Operation
     aborted: wrong number of elements in\n%1").arg(tablename), QMessageBox::Close
    );
                return;
            }
            QVector<QString> tmp;
            bool ok;
            for(int i=0; i < fields.size(); i++){
                double numcheck = fields[i].toDouble(&ok);
                if(fields[i] == "" || fields[i] == " "){
                    tmp.push_back("U");
                    errors[i] = true;
                }else if(fields[i] == "NaN" || ok == true){
                    tmp.push_back(fields[i]);
                }else{
                    error_occurred = true;
                    QMessageBox::critical(this, "An error has occurred", tr("
    Operation aborted: invalid character found in\n%1").arg(tablename),
    QMessageBox::Close);
                    return;
                }
            }
            table_elements.push_back(tmp);
        }
```

```
43          // Table has been completely parsed. Now, check again and remove elements
        of the table corresponding to where the errors are.
44          for(int i = 0; i < table_elements.size(); i++){
45              int k = 0;
46              int max = table_elements[i].size();
47              for(int j = 0; j < max; j++){
48                  if(errors[j] == true){
49                      table_elements[i].remove(j-k);
50                      k++;
51                  }
52              }
53          }
54      }
55      file.close();
56  }
```

**Listing 4.3:** The parseTable() function.

The first step in parsing the table consists in opening the file selected by the user in read-only mode. The first line contains the titles of all the columns, which are the names of the signals found in the table. The titles are read separately from the values: since the values are comma-separated, the line is read, and the elements are split by considering them to be located between two commas; a first check for errors is performed by making sure that the first column contains the list of simulation steps, called *time*. If this is not the case, the procedure is aborted. The title row is then inserted in the *table_elements* structure, a matrix represented as a QVector of QVectors of strings.

All the values present in the table can now be read: each iteration of the loop reads one row from the table until the end of the file has been reached. An operation similar to the title row is performed by separating the elements between the commas and checking for errors. The first check makes sure that the number of elements in the current row is equal to the expected number of fields, which is known from the title row. It also ensures that there is not only one element, which would mean that the table only contains the simulation time column, but no actual signals to plot.

The individual values of a row are then evaluated: each element must be either a number or a NaN value. As described in section 3.2.2, when an output in molecular technology is invalid due to the output pin being influenced by both molecules of the adjacent cell, the table lists the value as "NaN". Instead, if the value is present but it is an empty field, it is marked as an error so it can be flushed from the matrix of elements. If one of the elements does not satisfy any of these requirements, and is instead an invalid character, *parseTable* generates an error.

Once all the elements have been added to the matrix, the ones marked as erroneous can be removed from it by means of a nested loop. Iterating over all the elements, if the "errors" array states that an element is empty, the whole column is flushed; at the end of the function, the file is closed.

The table has been completely parsed, and all the data have been added to the matrix. Therefore, the Simulation Subwindow can now be instantiated.

## 4.2 The Simulation Subwindow

The core element of the new Waveform Viewer functionality is the class named *SimulationSubwindow*, which contains all the necessary elements to plot and visualize the signals contained in the table files. After the window is created, as part of Listing 4.2 showed, the class receives all the timing parameters and the matrix of table elements before the plot can be generated. The constructor of the Simulation Subwindow class (Listing 4.4) instantiates two more necessary classes, called *Chart* and *ChartView*. *ChartView* is a widget in which a Chart class element can be instantiated in order to draw a graph, with many different graphical features that can either be custom implemented by overriding an existing method, or designed from scratch. The font to be used for the labels on the Y-axis is also initialized here.

The subwindow also contains a horizontal and a vertical scrollbar, which allow the user to move along the two axes after a zoom-in action has been performed.

```
1  SimulationSubwindow::SimulationSubwindow(QWidget* parent) : QWidget(parent){
2     m_chart = new Chart();
3     m_chartView = new ChartView(m_chart, this);
4     m_chartView->setBackgroundBrush(Qt::black);
5     m_horizontalScrollBar = new QScrollBar(Qt::Horizontal, this);
6     m_verticalScrollBar = new QScrollBar(Qt::Vertical, this);
7     zooming_vertical = false;
8     labelsFont.setPointSize(8);
9     labelsFont.setWeight(QFont::DemiBold);
10    // create connection between axes and scroll bars:
11    connect(m_horizontalScrollBar, SIGNAL(valueChanged(int)), this, SLOT(
       horzScrollBarChanged(int)));
12    connect(m_verticalScrollBar, SIGNAL(valueChanged(int)), this, SLOT(
       vertScrollBarChanged(int)));
13    QGridLayout* plotAreaLayout = new QGridLayout();
14    plotAreaLayout->addWidget(m_chartView, 0, 0);
15    plotAreaLayout->addWidget(m_horizontalScrollBar, 1, 0, 1, 1, Qt::AlignBottom);
16    plotAreaLayout->addWidget(m_verticalScrollBar, 0, 1, 1, 1, Qt::AlignRight);
17    setLayout(plotAreaLayout);
18 }
```

**Listing 4.4:** The *SimulationSubwindow* constructor.

To receive the total simulation time and the table step from the Window class, function *setTime* is called (Listing 4.5). Inside it, a few timing parameters are set. The timescale is set to nanoseconds for all values, including the table step and the total simulation time. A label is also applied to the X axis through a *ChartView* method, in order to make it clear that time is being represented, and its unit of measurement is the nanosecond. Therefore, *setTime* sets the properties related to

the horizontal axis, while the vertical axis containing all the signals will be defined in further functions.

```
void SimulationSubwindow::setTime(double max, double table_step){
   m_timeScale = 1e-9; // Standard timeScale set as nanoseconds.
   m_table_step = (QString::number(table_step/ m_timeScale, 'e', 1)).toDouble();
   m_tmin = 0;
   m_tmax = max / m_timeScale;   // convert to selected timeScale
   m_chartView->updateXaxisTimeScale("t [ns]");
}
```

**Listing 4.5:** The setTime() function.

The signal plot can now be generated through a function called *plotSignals*. This function is responsible for the manipulation of the table elements matrix, whose values are offset in order to be drawn one above the other on the same plot, with no graphical overlaps. The maximum values for each range and the overall largest range are also evaluated, for plotting purposes. Listing 4.6 contains the first part of this function.

```
for (int j = 1; j < table_elements_offset[0].size(); j++){
    if(table_elements_offset[0][j] == "invalid_output"){
        curr_min = 0;
        curr_max = curr_min;
    } else{
        curr_min = table_elements_offset[1][j].toDouble();
        curr_max = curr_min;
    }
    for (int k = 1; k < table_elements_offset.size(); k++){ // For each time
    instant
        if(table_elements_offset[0][j] == "invalid_output"){
            curr_y = 0;
        } else{
            curr_y = table_elements_offset[k][j].toDouble();
        }
        if(curr_y < curr_min){
            curr_min = curr_y;
        } else if(curr_y > curr_max){
            curr_max = abs(curr_y);
        }
    }
    curr_range = curr_max - curr_min;
    if(curr_range > max_range){ // New max range found
        max_range = curr_range;
    }
    max_values.append(curr_range);
}
for(int i = 0; i < max_values.size(); i++){
    if(max_values[i] == 0){
        max_values[i] = max_range;
    }
}
QVector<QString> signalNames;
for (int i = 1; i <= m_nSignals; i++){
    signalNames.append(table_elements_offset[0][i]); // Copy names into the new
    vector
}
```

**Listing 4.6:** Obtaining the range for each table signal.

The incoming data is assigned to a structure called *table_elements_offset*, a matrix that will ultimately be used to plot the signals in the Waveform Viewer. This first nested loop is a preliminary manipulation of the content that reads through every signal at every time instant to determine whether that signal fits the previously mentioned *invalid_output* case, in which the current signal value is set to zero in place of the "NaN" value, or to the matrix value obtained in that cycle. The current value is used to determine, for every column, which are the minimum and maximum values, so that the range for that signal can be evaluated; then, the maximum range out of all the computed ones is obtained, to be used as the same offset for all values of the matrix. If a signal has a null range, which means that it is constant, the maximum range is assigned to it. A vector containing all the signal names is then filled with the first row of the table matrix.

To conclude the *plotSignals* function, all the table values must be offset using the *max_range* value just computed (Listing 4.7). By accumulating an offset to each signal, their values on the Y-axis can be represented on the plot with no overlaps, one above the other.

```
1  double curr_offset = max_range;
2  double curr_element = 0;
3  for (int j = 1; j < table_elements_offset[0].size(); j++){ // For every signal
4      for (int k = 1; k < table_elements_offset.size(); k++){ // For every timestamp
5          if(table_elements_offset[0][j] == "invalid_output"){
6              curr_element = max_range/max_values[j-1] + 2*curr_offset;
7          } else{
8              curr_element = ((table_elements_offset[k][j].toDouble())*(2*max_range/
       max_values[j-1]) + 2*curr_offset); // shift I/O values
9          }
10         table_elements_offset[k][j] = QString::number(curr_element);
11     }
12     curr_offset = curr_offset + 2*max_range;
13 }
14 configurePlot(signalNames);
```

**Listing 4.7:** Computing the offset for all signal values.

The whole matrix is traversed once more, and for each signal a new offset is accumulated by adding the maximum range; in fact, the summed value is $2*max\_range$, a choice made to further distance the signals in order to achieve a clearer plot in the end. The current element is either set to the sum between the table value and the offset, or simply to the sum between the offset itself and a scaling factor, if the signal is an invalid output; in fact, the scaling factor is also multiplied by the table value in the first case, in order to normalize the signal amplitudes. The value is then stored in the matrix, replacing the original value. The last step of *plotSignals* consists in calling another function named *configurePlot*: the signal names are passed to it and, using the newly computed values in the matrix, the plot is generated.

The goal of function *configurePlot* is to fully setup the chart by adding all

the signals stored in the table, defining the labels on the Y-axis to distinguish the signals and defining the starting values for the scrollbars and the visible range. As the first part contained in Listing 4.8 shows, the first step consists in inserting the table elements in the chart.

```
double curr_x, curr_y;
QPen pen(Qt::green);
QPen err_pen(Qt::red);
pen.setWidth(2);
err_pen.setWidth(2);
for (int j = 1; j < table_elements_offset[0].size(); j++){ // Write the logic
    value of each I/O signal
    QLineSeries* series = new QLineSeries();
    for (int k = 1; k < table_elements_offset.size(); k++){
        curr_x = (table_elements_offset[k][0].toDouble()) / m_timeScale;
        curr_y = table_elements_offset[k][j].toDouble();
        *series << QPointF(curr_x, curr_y);
    }
    m_chart->addSeries(series);
    if(table_elements_offset[0][j] == "invalid_output"){
        series->setPen(err_pen);
    }else{
        series->setPen(pen);
    }
}
```

**Listing 4.8:** Adding all the line series to the chart.

The signals are represented as lines through *QLineSeries*. For each element of each signal, the value on the X-axis corresponds to the simulation time, converted to nanoseconds using the timescale, while the Y-coordinate is the signal value. The color of the signal line is then chosen by means of two different pens: the green pen is assigned to the valid signals, while the red pen is used for signals corresponding to *invalid_output*.

Once the signal series have been added to the chart, each of them must be labeled by using the corresponding name and placing it in the correct position on the Y-axis. This action is performed in Listing 4.9.

```
// Label as signal name using QCategoryAxis
QCategoryAxis* axisY = new QCategoryAxis();
axisY->setLabelsPosition(QCategoryAxis::AxisLabelsPositionCenter);
axisY->setLabelsColor(Qt::white);
m_chart->setTitleBrush(Qt::white);
axisY->setLabelsFont(labelsFont);
axisY->setTruncateLabels(false);
axisY->setGridLineVisible(false);
axisY->setLinePen(Qt::NoPen);
// Add labels for signals
double offset = max_range;
QString s;
for (int i = 0; i < signalNames.count(); i++){
    if(signalNames[i] == "invalid_output"){
        s = QString("%1 | <br> %2").arg(signalNames[i]).arg("U");
    }else{
        s = QString("%1 | <br> %2").arg(signalNames[i]).arg(0);
    }
    axisY->append(s, 2*max_range + 2*offset + 0.5);
```

```
20      offset = offset + 2*max_range;
21  }
22  foreach (QAbstractSeries* s, m_chart->series())
23      m_chart->setAxisY(axisY, s);
```

**Listing 4.9:** Chart labels setup.

The Y-axis is created as a *QCategoryAxis*: this Qt class allows to define a chart axis as one composed of multiple categories, each one labeled individually. Therefore, it is the necessary choice in this case where multiple signals are drawn on the same chart. All the labels are placed at the center of the category. After setting the font for the labels, the vector containing all the signal names is iterated over to initialize the labels and append them to the chart. If the signal is invalid, it is initialized as "U", otherwise it is set to zero. The effective initial value will be updated in a function inside the *ChartView* class that will be explained later. The correct position of the labels is obtained by accumulating multiples of the maximum range, previously used to offset the signal values. All the obtained line series are then assigned to the Y-axis.

The final step of function *configurePlot* consists in setting up the organization of the Simulation Subwindow (Listing 4.10) by initializing the required values for both the horizontal and vertical scrollbar, as well as the Y-axis range.

```
1  m_horizontalScrollBar->setPageStep(100*m_table_step);
2  m_verticalScrollBar->setPageStep(4*max_range);
3  m_horizontalScrollBar->setRange(0, 0); // zoom full-->no range
4  total_range = 4*max_range*m_nSignals;
5  if(m_nSignals <= NMAX){
6      visible_signals = m_nSignals;
7  }else{
8      visible_signals = NMAX;
9  }
10 max_signals = visible_signals;
11 zoomed_range = total_range;
12 m_verticalScrollBar->setRange(0, 0); // zoom full-->no range
13 m_chart->axisY()->setRange(0, total_range);
```

**Listing 4.10:** Setup of scrollbars and visible range.

The main quantities that define the behavior of a Qt scrollbar are the range, which is the total range the scrollbar can move across, and the page step, which is the amount scrolled each time the bar is dragged; for instance, the horizontal page step is initially set up to move by 100 table steps. These values will later be modified accordingly, depending on the visible range as a consequence of zooming in. The total range of the vertical axis is set to the product between the range of a signal and the total number of signals; some additional quantities that will be used later are also initialized. A maximum number of signals can be shown in the window at once, $NMAX$, equal to 15; this avoids an excessive crowding of the plot.

The elements of the chart have been setup and the Waveform Viewer is now ready to be visualized. Control is returned to the calling Window class: the *show*() function of Simulation Subwindow is accessed to finalize the setup of the Chart View, including the starting values of the labels and fitting the viewer inside the window.

### 4.2.1 Chart View

As mentioned earlier, the *ChartView* class contains the chart that has just been setup, and it could initialize additional widgets, if present, which is the case here. As the constructor in Listing 4.11 entails, there are a few elements which are overlayed to the chart representation.

```
m_parentTab=static_cast<SimulationSubwindow*>(parent);
m_zoomLevel=1;
setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
m_chart=chart;
m_mouseClicked=false;
first_access = true;
m_VLineLabel= new QGraphicsSimpleTextItem(m_chart);
m_axesXLabel= new QGraphicsSimpleTextItem(m_chart);
m_VLineLabel->setBrush(QBrush(Qt::white));
QPen pen(Qt::yellow);
pen.setWidth(1);
m_verticalLine= this->scene()->addLine(0,0,0,verticalLineLength,pen);
m_labelRect=this->scene()->addRect(0,0,labelRectWidth,labelRectHeight,pen,QBrush(
    Qt::NoBrush));
scene()->addItem(m_chart);
m_chart->setBackgroundBrush(Qt::black);
setRenderHint(QPainter::Antialiasing);
this->ensureVisible(m_chart,0,0);
m_VLineLabel->setZValue(ZValue);
m_verticalLine->setZValue(ZValue);
m_labelRect->setZValue(ZValue);
m_axesXLabel->setZValue(ZValue);
```

**Listing 4.11:** *ChartView* constructor.

The additional visual features that are added to the Chart View are a vertical line, which is moved by clicking or dragging on the chart, and a label connected to the line which is updated with its current time position intercepted on the X-axis. Every time this vertical line is moved, not only the attached label is updated, but also all the signal labels are modified to depict the value of each signal in the time instant intercepted by the line. The constructor also initializes the pen used to draw the vertical line, which is yellow, and sets up its initial position and the label's position using defined constants.

Once the *show*() function of the Simulation Subwindow is called, the first call to the *updateLine* function is made, in order to initialize the line's position and to

assign to the signal labels the starting values at time 0 *ns*. Listing 4.12 takes care of the vertical line position and its attached label's value and position.

```
void ChartView::updateLine(QPoint event){
    QPointF intercept=chart()->mapToValue(this->mapToScene(event));
    qreal xval=intercept.x();
    QList<QAbstractAxis *> axesList;
    axesList = m_chart->axes(Qt::Horizontal);
    QValueAxis * axisX = static_cast<QValueAxis *> (axesList[0]);
    if(xval<axisX->min())
        xval=axisX->min();
    else if(xval>axisX->max())
        xval=axisX->max();
    intercept=m_chart->mapToPosition(QPointF(xval,intercept.y()));
    if(first_access == true){
        first_access = false;
        if(m_parentTab->getNSignals() > NMAX){
            m_parentTab->verticalZoom(0);
        }
        m_verticalLine->hide();
        m_VLineLabel->hide();
        m_labelRect->hide();
    } else{
        m_verticalLine->show();
        m_verticalLine->setX(intercept.x());
        m_VLineLabel->show();
        m_labelRect->show();
    }
    m_VLineLabel->setText(QString("%1").arg(xval,7,'f',2));
    m_VLineLabel->setPos(intercept.x()-VLineLabelXPos,m_chart->size().height()-
    VLineLabelYPos);
    m_labelRect->setPos(intercept.x()-labelRectXPos,m_chart->size().height()-
    labelRectYPos);
    QVector<double> values= m_parentTab->getInterceptData(xval);
    m_parentTab->updateLabels(values);
}
```

**Listing 4.12:** Updating the vertical line and its label.

By acquiring the currently intercepted value on the chart, the value on the X-axis can be determined. The intercept is mapped to the position on the chart, and that point is used to setup the position of the vertical line and its label. A boolean value named $first\_access$ is used to determine whether this is the first access to *updateLine*, meaning that the program is in the initialization phase, or the chart is being updated while it's running. If the former is true, the line and its label are not shown, as the starting window simply depicts the full view of the chart, and the signal labels show the value at 0 *ns*; a preliminary zoom is also performed in case there are in total more than 15 signals, in order to show no more than the maximum amount in a single window. In case it is not the first access to the function, the vertical line and the label become visible, and their position is modified accordingly.

A *SimulationSubwindow* class function must be triggered so as to obtain the Y-axis values of all the signals in the time position currently intercepted by the vertical line; these values are stored in a vector. Listing 4.13 depicts $getInterceptData()$,

the function required to retrieve all the signal values at a certain time instant.

```
QVector<double> SimulationSubwindow::getInterceptData(qreal xval){
    QVector<double> interceptedData;
    int index = 0;
    xval = xval * m_timeScale;
    xval = (QString::number(xval, 'e', 3)).toDouble();
    for(int i = 1; (i < table_elements_offset.size()-1) && (index == 0); i++){
        if((xval >= table_elements_offset[i][0].toDouble()) && (xval <
    table_elements_offset[i+1][0].toDouble())){
            index = i;
        } else if(xval == table_elements_offset[i+1][0].toDouble()){
            index = i+1;
        }
    }
    for (int i = 1; i < table_elements[0].size(); i++){
        interceptedData.append(table_elements[index][i].toDouble());
    }
    return interceptedData;
}
```

**Listing 4.13:** The *getInterceptData* function.

The loop is organized in such a way to read from the table the value closest to the effective position of the vertical line: if the current position is between two consecutive values $i$ and $i+1$, the index $i$ has been found, otherwise it is set to the next element (if it is equal to it); this is done especially to ensure that the last possible value effectively corresponds to the last element in the table. The vector to be returned is then filled with the table values at that time instant.

After all the values have been retrieved, the signal labels on the Y-axis must be refreshed by calling a function named *updateLabels* (Listing 4.14). The function is defined in the *SimulationSubwindow* class, not in *ChartView*, so as to keep the management of the axis only in one class.

```
void SimulationSubwindow::updateLabels(QVector<double> values){
    auto axes = m_chart->axes(Qt::Vertical);
    if(axes.size() == 1){
        QCategoryAxis *axisY=static_cast<QCategoryAxis *>(axes[0]);
        int i=0;
        foreach(QString oldLabel, axisY->categoriesLabels()){
            QString name =oldLabel.split(' ').first();
            QString s;
            if(name == "invalid_output"){
                s = QString("%1 | <br> %2").arg(name).arg("U");
            }else{
                s = QString("%1 | <br> %2").arg(name).arg(values.at(i), 10, 'f',
    3, ' ');
            }
            axisY->replaceLabel(oldLabel,s);
            i++;
        }
    }
}
```

**Listing 4.14:** Updating the signal labels.

By iterating over each label, the value can be updated either as "U", if the signal is an *invalid_output*, or taken from the vector at the correct position. The old

label is then replaced with the new string.

The *ChartView* class is also capable of managing different features that the user can access, such as zooming functionalities, clicking and dragging the aforementioned vertical line, and more. The most prominent feature is the zoom, which has been implemented separately for horizontal and vertical zoom. In this way, the user can choose on which axis to modify the magnification, for a more thorough analysis of the signals. To access this feature, one of the standard methods of *ChartView*, called *wheelEvent* (Listing 4.15), has been overridden. This method is triggered whenever the user performs an action using the scroll wheel, to which a custom action can be associated.

```
void ChartView::wheelEvent(QWheelEvent *event){
    if(event->angleDelta().y()>0){
        if(event->modifiers() == Qt::ControlModifier){
            m_parentTab->verticalZoom(0);
        }else{
            chart()->zoom(1.1);
            updateLine();
        }
    }
    else if(event->angleDelta().y()<0){
        if(event->modifiers() == Qt::ControlModifier){
            m_parentTab->verticalZoom(1);
        }else{
            chart()->zoom(0.9);
            updateLine();
        }
    }
}
```

**Listing 4.15:** Creation of shortcuts related to the mouse scroll wheel.

A modifier of the wheel scroll event is also captured, corresponding to a key being pressed while the wheel event is occurring. In this way, the function can distinguish between a normal scroll and a $CTRL + scroll$ action: in the first case, a zoom on the horizontal axis is performed, and the vertical line is updated; in the latter, a custom function that performs a vertical zoom is called.

Another standard function that has been overridden is the one that can intercept when keys are pressed on the keyboard, turning them into shortcuts for particular actions. Function *keyPressEvent* shown in Listing 4.16 allows the user to zoom in and out horizontally using the plus and minus keys, to reset the horizontal zoom using key "F", and to scroll horizontally and vertically using the arrow keys.

```
void ChartView::keyPressEvent(QKeyEvent *event){
    switch (event->key()) {
    case Qt::Key_Plus: {
        chart()->zoom(1.1);
        updateLine();
        break;
    }
    case Qt::Key_Minus:
        chart()->zoom(0.9);
```

```
10          updateLine();
11          break;
12      case Qt::Key_F:
13          chart()->zoomReset();
14          break;
15      case Qt::Key_Left:
16          chart()->scroll(-10, 0);
17          break;
18      case Qt::Key_Right:
19          chart()->scroll(10, 0);
20          break;
21      case Qt::Key_Up:
22          if((m_parentTab->m_verticalScrollBar->value() != m_parentTab->
        m_verticalScrollBar->minimum()) && (m_parentTab->m_verticalScrollBar->
        isEnabled())){
23              m_parentTab->m_verticalScrollBar->setValue(m_parentTab->
        m_verticalScrollBar->value() - m_parentTab->getMaxRange());
24          }
25          break;
26      case Qt::Key_Down:
27          if((m_parentTab->m_verticalScrollBar->value() != m_parentTab->
        m_verticalScrollBar->maximum()) && (m_parentTab->m_verticalScrollBar->
        isEnabled())){
28              m_parentTab->m_verticalScrollBar->setValue(m_parentTab->
        m_verticalScrollBar->value() + m_parentTab->getMaxRange());
29          }
30          break;
31      default:
32        QGraphicsView::keyPressEvent(event);
33        break;
34      }
35  }
```

**Listing 4.16:** Keyboard shortcuts.

The horizontal zoom is implemented using an already existing Qt function that can zoom in or out depending on the specified factor. Instead, the vertical one is defined inside *SimulationSubwindow* and described in the next section. In either case, once a zoom-in has been performed, the functionalities of the scrollbars also become operative.

### 4.2.2 Zoom and Scroll Functionalities

As already mentioned, the horizontal zoom is performed using an already existing Qt function. Instead, the custom vertical zoom is a function (Listing 4.17) that modifies the number of signals that are visible on screen at once, without making any changes to the visible simulation time. Therefore, it effectively enlarges the signals so they can be inspected from up close.

```
1  void SimulationSubwindow::verticalZoom(int direction){
2      zooming_vertical = true;
3      m_verticalScrollBar->setDisabled(false);
4      if(direction == 0){ // Zoom in vertically
5          if(visible_signals > 1){
6              visible_signals--;
7              zoomed_range = 4*max_range*visible_signals;
```

```
8             m_chart->axisY()->setRange(-m_verticalScrollBar->value(), -
    m_verticalScrollBar->value() + zoomed_range);
9         }
10    } else{ // Zoom out vertically
11        if((visible_signals < max_signals-1)){
12            visible_signals++;
13            zoomed_range = 4*max_range*visible_signals;
14            m_chart->axisY()->setRange(-m_verticalScrollBar->value(), -
    m_verticalScrollBar->value() + zoomed_range);
15        }else if (m_nSignals <= NMAX){
16            m_verticalScrollBar->setValue(0);
17            m_verticalScrollBar->setDisabled(true);
18        }
19    }
20 }
```

**Listing 4.17:** The function implementing the vertical zoom.

The *verticalZoom*() function can be accessed by passing an integer value that determines the direction of the zoom: 0 for zoom-in, 1 for zoom-out. Once that is determined, the number of visible signals, initialized to the total number of signals or to the previously mentioned NMAX, is modified accordingly at each access in a range between 1 and $max\_signals - 1$. Therefore, the program can zoom-in until only one signal is visible, and zoom-out until all of them are simultaneously available on screen. This number is used to compute the new visible range of the Y-axis. In particular, once the user has reached the maximum number of signals, the program disables the scrollbar.

Whichever zoom is performed, the two scrollbars become available so that the user can navigate through the chart from top to bottom and from left to right. When a movement of the scrollbar is detected, a signal is automatically asserted to activate the corresponding function capable of handling the change in the visible range; there are two separate methods, one for each scrollbar (Listing 4.18).

```
1  void SimulationSubwindow::horzScrollBarChanged(int value){
2      m_chart->axisX()->setRange(value, (value + m_horizontalScrollBar->pageStep()))
       ;
3      m_chartView->updateLine();
4  }
5  void SimulationSubwindow::vertScrollBarChanged(int value){
6      m_chart->axisY()->setRange(-value, -value + zoomed_range + m_verticalScrollBar
       ->pageStep());
7      m_chartView->setBackgroundBrush(Qt::black);
8  }
```

**Listing 4.18:** Methods handling the scrollbars.

These two slot functions modify the range of the corresponding axis, taking into account the new value of the scrollbar, the current zoomed range (when handling the vertical scrollbar) and the page step of the scrollbar. In addition to that, zooming or scrolling on the horizontal axis also updates the vertical line.

These are not the only necessary functions when the visible range is modified,

either due to a zoom or to a scrolling operation. There are two additional functions, called *xRangeChanged* and *yRangeChanged* (Listing 4.19), which are accessed when the aforementioned events occur. Their purpose is to define the new axis ranges and the scrollbar parameters, namely the range, the page step, the single step (corresponding to the amount scrolled when pressing the Page Up and Page Down keys) and the value (linked to the current scrollbar position).

```
void SimulationSubwindow::xRangeChanged(qreal min, qreal max){
    min = (QString::number(min, 'e', 3)).toDouble();
    max = (QString::number(max, 'e', 3)).toDouble();
    qreal range = max - min;
    if (min < m_tmin){
      m_chart->axisX()->setRange(m_tmin, max);
      min = m_tmin;
    }
    if (max > m_tmax){
      m_chart->axisX()->setRange(min, m_tmax);
    }
    m_horizontalScrollBar->setPageStep(range);
    m_horizontalScrollBar->setRange(0, ceil(m_tmax - m_horizontalScrollBar->
    pageStep()));
    m_horizontalScrollBar->setValue(min);
}
void SimulationSubwindow::yRangeChanged(qreal min, qreal max)
{
    if(zooming_vertical == false){
        vertScrollBarChanged(m_verticalScrollBar->value());
    } else{
        qreal range = max - min;
        if (min < 0){
            m_chart->axisY()->setRange(0, max);
            min = 0;
        }
        if (max > total_range){
            m_chart->axisY()->setRange(min, total_range);
        }
        m_verticalScrollBar->setPageStep(2*range/m_nSignals);
        m_verticalScrollBar->setSingleStep((m_verticalScrollBar->pageStep())-
    max_range);
        m_verticalScrollBar->setRange(-total_range + zoomed_range, 0);
        zooming_vertical = false;
        vertScrollBarChanged(m_verticalScrollBar->value());
    }
}
```

**Listing 4.19:** The two functions that manage the visible ranges.

## 4.3 Final Results

After all the described changes and the new implementations inside FCNS Viewer, the Waveform Viewer functionality is complete and fully operational. Figure 4.1 depicts the full view of the plot when an example iNML table is opened.

After zooming horizontally, the visible time axis is shrinked as shown in Figure 4.2. The labels reflect the signal values intercepted in the vertical line position.

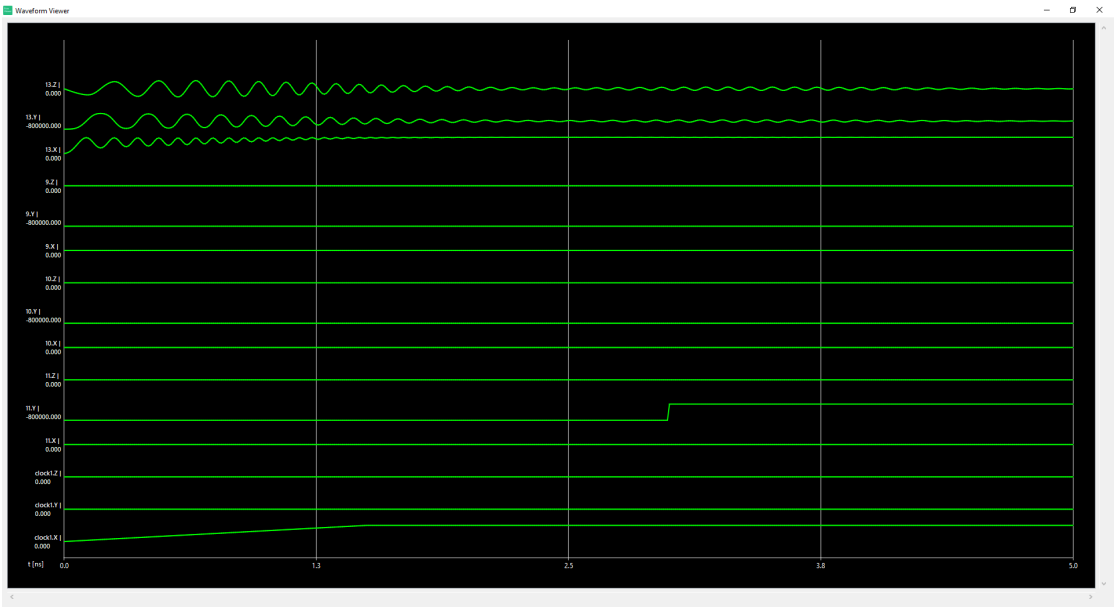



**Figure 4.1:** Full Waveform Viewer window.

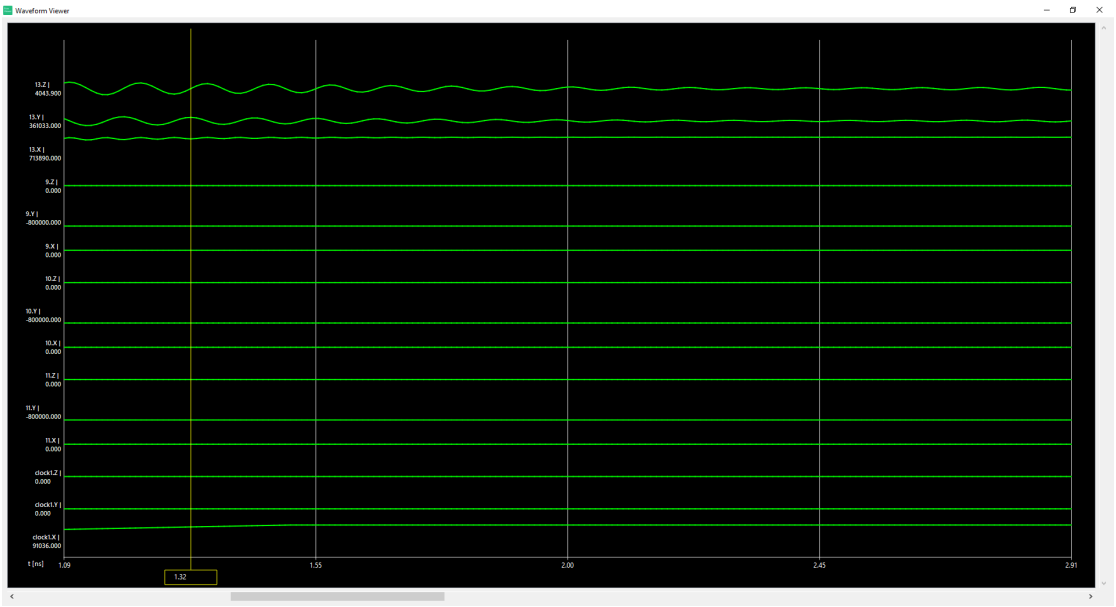



**Figure 4.2:** Horizontal zoom.

In order to visualize one or more signals more closely, a vertical zoom can be performed; in particular, Figure 4.3 shows the three output magnetizations in the same iNML circuit previously depicted. Also, as previously mentioned, there is a

possibility that the output of a MolQCA simulation is invalid, due to the placement of the output pin and molecule being in one of the undetermined cases. When this occurs, the output is shown as a constant red line, labeled as "U" (Figure 4.4).

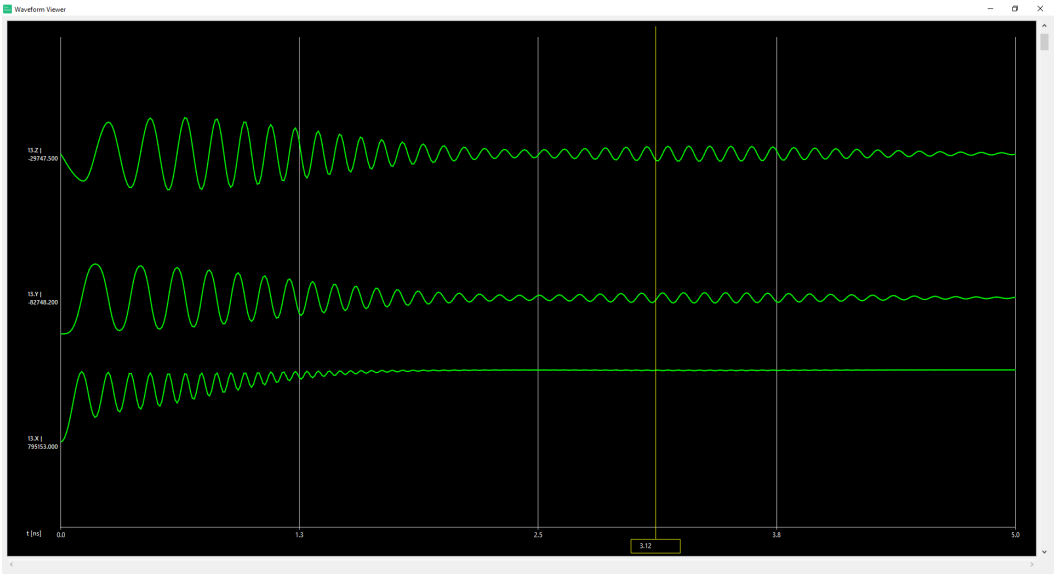



**Figure 4.3:** Vertical zoom on the output pin's magnetization components.

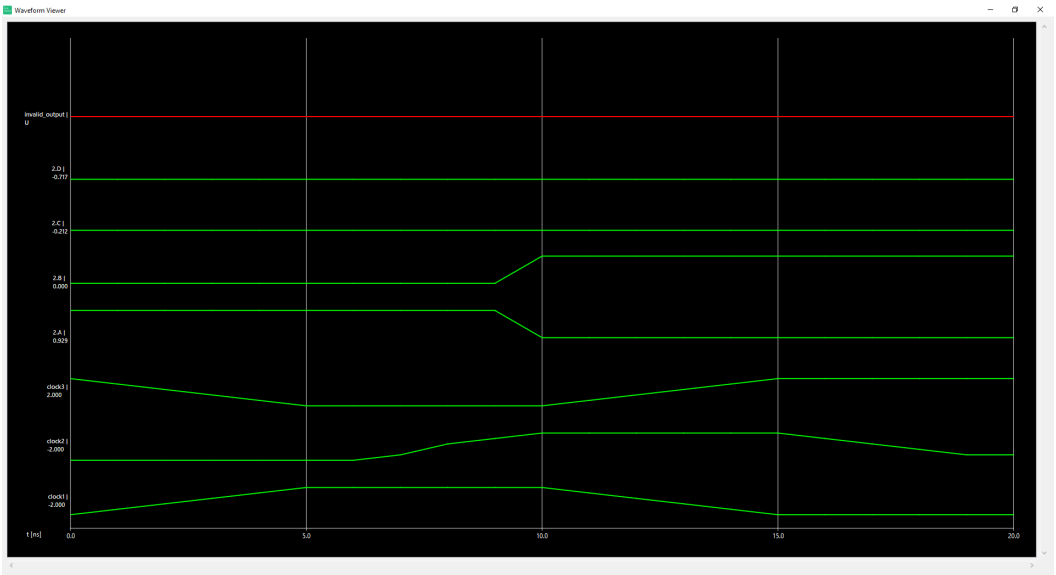



**Figure 4.4:** MolQCA waveforms with an invalid output.

# Chapter 5

# Conclusion and Future Works

The work performed and presented in this Thesis has introduced the capability to perform a full design and simulation flow also for Molecular QCA circuits. This expands the current support for FCN technologies, allowing the research to perform experimental analysis and work on both iNML and MolQCA, to study alternative or complementary implementations to traditional CMOS circuits.

Thanks to the Waveform Viewer feature added to FCNS Viewer, it is not only possible to visualize all the elements of a circuit in 3D, analyzing the evolution of the system behavior in time, but also to verify the values of the clocks and the signals by representing them as waveforms. The new software implementations further expand the ToPoliNano Framework, moving it even closer in terms of feature completeness to the commercial tools available for the CMOS technology.

Future developments could now focus on expanding even further the range of supported Field-Coupled Nanocomputing technologies, by achieving full support for the pNML implementation of Magnetic QCA. Furthermore, it will be possible to study and analyze an evolution toward hybrid systems that would be capable of reaping the benefits of both QCA and CMOS technologies, to obtain the best of both worlds and achieve the progress of new implementations that research is currently working on.

Further expansions of the Waveform Viewer can also be studied, leading to an increment of the available features in order to improve the flexibility of the software, such as modifying signal positions in the window, hiding and showing signals as needed, and more.

# Bibliography


[1] Gordon E. Moore. «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.» In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860 (cit. on pp. 1, 2).

[2] Fabrizio Riente, Giovanna Turvani, Marco Vacca, Massimo Ruo Roch, Maurizio Zamboni, and Mariagrazia Graziano. «ToPoliNano: A CAD Tool for Nano Magnetic Logic». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.7 (2017), pp. 1061–1074. DOI: 10.1109/TCAD.2017.2650983 (cit. on pp. 2, 5, 10, 12, 13).

[3] T. Cole and J.C. Lusth. «Quantum-dot cellular automata». In: *Progress in Quantum Electronics* 25.4 (2001), pp. 165–189. ISSN: 0079-6727. DOI: https://doi.org/10.1016/S0079-6727(01)00007-6 (cit. on pp. 2, 3).

[4] C.S. Lent, P.D. Tougaw, and W. Porod. «Quantum cellular automata: the physics of computing with arrays of quantum dot molecules». In: *Proceedings Workshop on Physics and Computation. PhysComp '94*. 1994, pp. 5–13. DOI: 10.1109/PHYCMP.1994.363705 (cit. on p. 3).

[5] Umberto Garlando, Marcel Walter, Robert Wille, Fabrizio Riente, Frank Sill Torres, and Rolf Drechsler. «ToPoliNano and fiction: Design Tools for Field-coupled Nanocomputing». In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, pp. 408–415. DOI: 10.1109/DSD51259.2020.00071 (cit. on pp. 3, 4, 10, 12, 14).

[6] M T Niemier et al. «Nanomagnet logic: progress toward system-level integration». In: *Journal of Physics: Condensed Matter* 23.49 (Nov. 2011), p. 493202. DOI: 10.1088/0953-8984/23/49/493202 (cit. on p. 6).

[7] Mariagrazia Graziano, Marco Vacca, Alessandro Chiolerio, and Maurizio Zamboni. «An NCL-HDL Snake-Clock-Based Magnetic QCA Architecture». In: *IEEE Transactions on Nanotechnology* 10.5 (2011), pp. 1141–1149. DOI: 10.1109/TNANO.2011.2118229 (cit. on p. 6).

[8] Valentina Arima et al. «Toward quantum-dot cellular automata units: thiolated-carbazole linked bisferrocenes». In: *Nanoscale* 4 (3 2012), pp. 813–823. DOI: 10.1039/C1NR10988J (cit. on p. 8).

[9] Ruiyu Wang, Azzurra Pulimeno, Massimo Ruo Roch, Giovanna Turvani, Gianluca Piccinini, and Mariagrazia Graziano. «Effect of a Clock System on Bis-Ferrocene Molecular QCA». In: *IEEE Transactions on Nanotechnology* 15.4 (2016), pp. 574–582. DOI: 10.1109/TNANO.2016.2555931 (cit. on pp. 8, 9).

[10] Marcel Walter, Robert Wille, Frank Sill Torres, Daniel Große, and Rolf Drechsler. *fiction: An Open Source Framework for the Design of Field-coupled Nanocomputing Circuits*. 2019. DOI: 10.48550/ARXIV.1905.02477 (cit. on p. 14).

[11] Umberto Garlando, Fabrizio Riente, Deborah Vergallo, Mariagrazia Graziano, and Maurizio Zamboni. «ToPoliNano & MagCAD: A Complete Framework for Design and Simulation of Digital Circuits Based on Emerging Technologies». In: *2018 15th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. 2018, pp. 153–156. DOI: 10.1109/SMACD.2018.8434919 (cit. on p. 16).