



**Politecnico
di Torino**

Politecnico di Torino

Master of Science Degree in
Electronic Engineering
2020 / 2021

Simulation of Hybrid Circuits based on Beyond-CMOS Technologies

Academic Supervisors

Prof. Maurizio Zamboni
Ph.D. Fabrizio Riente
Ph.D. Umberto Garlando

Candidate

Alessandro Rocco Scisca

Abstract

Because of the ever-so-complex challenges posed by Silicon in continuing the trend of constant evolution that has characterized electronics for the past few decades, in the recent years researchers have been increasingly interested on developing new and promising technologies that can overcome the current CMOS scaling limits and possibly offer unique features.

These technologies are referred to as *Beyond CMOS* technologies as they employ innovative materials and techniques to construct digital circuits. On account of the scattering in popularity and the development stage of the many candidates of this field, the world of *simulations* is heavily fractured: the more popular entries already feature custom simulators, some simply offer models compatible to other existing simulators, others only come with mathematical models.

The objective of this thesis is to develop new tools to build a generic *Hybrid Simulator*, a simulator not only able to support arbitrary technologies given the right models, but also to handle circuits hosting more of these technologies that coexist in the same design. The presented work was developed as an extension of

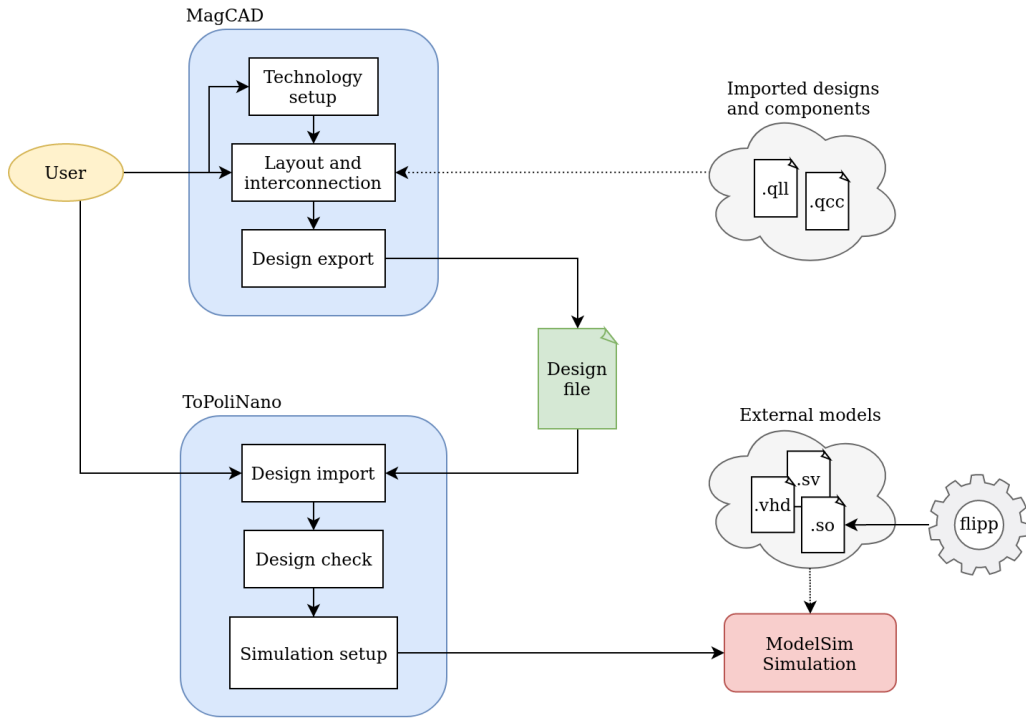


Figure 1: Overview of the devised workflow starting from the design of a component in MagCAD to its simulation in ToPoliNano. This summarizes the most important steps and the entities they interact with

the **ToPoliNano** framework with the objectives of generality, customizability and ease of integration.

The first necessary step was to extend **MagCAD** to allow the user to design hybrid circuits. Accordingly to the program structure I developed the Hybrid Plugin, a plugin providing the interface and all the additional tools needed to place and interconnect components designed in MagCAD featuring arbitrary technologies. On top of this, the new plugin also supports the insertion of custom VHDL or Verilog netlists in the design to represent classical CMOS digital components or testing facilities.

In order to run the simulations managed by **ToPoliNano**, the choice fell on offloading the work to ModelSim, an established and feature-rich simulator. This is because not only does ModelSim have extensive and thoroughly tested support for VHDL, Verilog, VerilogA and much more, but also because of its interesting *Foreign Language Interface* (FLI). This feature offers a C API through which the user can interact with the simulation and insert his own custom models developed in C.

To simplify the interaction with ModelSim and to adapt it with the modern standards, a small C++ library called **flipp** was also developed. It adapts the most relevant features from the FLI to C++ and adds some useful wrappers and helpers and it can be used when designing custom models.

Finally, I developed a module to be inserted into ToPoliNano offering a set of useful classes to build custom C++ models based on flipp plus the necessary infrastructure to manage and interact with the simulation.

The result of this work is a rich, fully customizable and easily extensible system which enables the user not only to define or import models with varied descriptions (VHDL, Verilog, VerilogA, C++), but also to freely design and simulate hybrid circuits where more of these models are interconnected.

Contents

Figures	IV
1 Introduction	1
1.1 Beyond CMOS	1
1.2 Simulations	2
1.3 ToPoliNano Framework	3
2 MagCAD	5
2.1 Introduction	5
2.2 Program Structure	7
2.3 Hybrid Plugin	10
2.4 Hybrid Items	12
2.5 Parser Example: VHDL Parser	17
2.6 Wiring	23
2.7 Wiring Tool	28
2.8 Wiring Elements	32
2.9 Connectors	35
2.10 Undo Commands	38
2.11 Saving	43
2.12 Loading	47
3 Flipp	50
3.1 ModelSim FLI	50
3.2 FLI Example: an Inverter in C	51
3.3 Flipp	57
3.4 Flipp Example: an Inverter in C++	60
3.5 Model-Behavior Approach	62
4 Simulation	67
4.1 Introduction	67

4.2	The Hybrid Circuit	68
4.3	iNML Wire	69
4.4	MTJ Converter	72
4.5	File Reader	76
4.6	File Writer	79
4.7	Testbench	81
4.8	Results	82
5	Conclusion	85
5.1	Conclusion	85
5.2	Future Works	86
	Bibliography	87

Figures

1	Tool Workflow	i
1.1	Comparison of Intel 4004 to Apple M1 Max	1
1.2	ToPoliNano framework	4
2.1	Standard view in MagCAD	5
2.2	Drawing Settings window	6
2.3	Component Insertion	7
2.4	MagCAD structure	8
2.5	The “Option Widget” for a VHDL Netlist item	11
2.6	No Path Error	11
2.7	Hybrid Items icons from the Item Picker	13
2.8	Inheritance Graph for Hybrid Wrapper Items and Parsers	15
2.9	Placed Items	17
2.10	VHDL Parser State Machine	19
2.11	Example of Parsing Error	21
2.12	A Greeting from Wires	23
2.13	Icon for the Wiring Tool	23
2.14	Selected Nodes	24
2.15	Path Selection	25
2.16	Methods of Deleting Nodes	26
2.17	Context Menu from Wiring Tool	27
2.18	Effect of Simplify Geometry	27
2.19	Qt Event System	29
2.20	Wiring Classes Hierarchy	31
2.21	Point Connectors	36
3.1	Waveform of the Inverter in C	57
3.2	Waveform of the Inverter in C++	63
3.3	Waveform of the Inverter with the Model-Behavior approach	65

4.1	Scheme of the Simulation	68
4.2	iNML Signal Propagation	70
4.3	iNML Item Selection	70
4.4	Placing iNML Items	71
4.5	Pin Settings	71
4.6	Final iNML Design	72
4.7	Component exoprt dialog	72
4.8	MTJ Structure	73
4.9	Activation of the MTJ	73
4.10	Simulation Waveforms	83
4.11	Output of the Python checker	84

Listings

2.1	HybridTechnology class declaration	10
2.2	Names of the items provided by the Hybrid Plugin	10
2.3	Example of a FactoryButton constructor	11
2.4	Creation of an item through its factory button	12
2.5	HybridItemWrapperBase class declaration	13
2.6	HybridItemWrapperBase draw() method	14
2.7	The compile() method	15
2.8	Abstract Parser class	15
2.9	VHDLEntityParser class	17
2.10	Structure of the VHDL parser loop	18
2.11	Example of the actions in a parser state	20
2.12	Collection of ParsedData	20
2.13	Usage of a Parser (the missing part from <u>Listing 2.7</u>)	21
2.14	Example of a VHDL file with an error	21
2.15	Simplifying geometry	27
2.16	Enable and disable handlers for QcaSceneTool	28
2.17	QcaSceneTool event handlers	28
2.18	Example of a DrawingScene event handler	29
2.19	Default implementation of a QcaSceneTool event fallback	30
2.20	Full implementation of WiringTool's mouse press event fallback handler	30
2.21	Common loop throughout the Wiring Tool	31
2.22	Main methods of the Segment class	32
2.23	Transform of a Segment	33
2.24	Bounding rect of a Segment	33
2.25	Constructor and main attribute of a Node	33
2.26	Retrieving the connections of a Node	34
2.27	A wire connecting two nodes	34
2.28	Pathfinding in a Wire	34
2.29	Base Connector class	35
2.30	PointConnector class	36

2.31	Listing the connection points of an Hybrid Wrapper item	36
2.32	Listing the connection points of an Hybrid Wrapper item	37
2.33	Naive implementation of a CreateNodeCommand	40
2.34	Improvement on the naive implementation of a CreateNodeCommand	40
2.35	Class for automatic deletion of objects	41
2.36	Final implementation of a simplified create node command	41
2.37	Contents of the save file from <u>Figure 2.3</u>	43
2.38	Exportable class	44
2.39	Type definitions from DrawingSaver	44
2.40	Saving of a scene	45
2.41	Wire's exportItem() method	45
2.42	Node's exportItem() method	46
2.43	Contents of the save file from <u>Figure 2.3</u>	46
2.44	Interception of Hybrid items during loading	47
2.45	The DrawingReader class	47
2.46	Inserting loaded wires	48
3.1	A foreign architecture	50
3.2	Optional arguments in a foreign declaration	51
3.3	Makefile	51
3.4	Declaration of the inverter_init() function	52
3.5	Printing the region name	52
3.6	ModelSim transcript	53
3.7	Allocating the Inverter	54
3.8	C enum mirroring the std_logic type in VHDL	55
3.9	Allocating the Inverter	55
3.10	Testbench for the Inverter in C	56
3.11	FLI Simulation commands	56
3.12	Invoking a .do script	56
3.13	Snippet of the base Handle class in flipp	57
3.14	Region Handle	58
3.15	Example of a wrapping function to create a region	59
3.16	Example of printing a Handle with flipp	60
3.17	Output for <u>Listing 3.16</u>	60
3.18	Implementation of an Inverter in C++	60
3.19	Flipp 's type-functions	62
3.20	Allocation of a Model	63
3.21	Allocation of a Model	64
3.22	run() method for the InverterAction	64

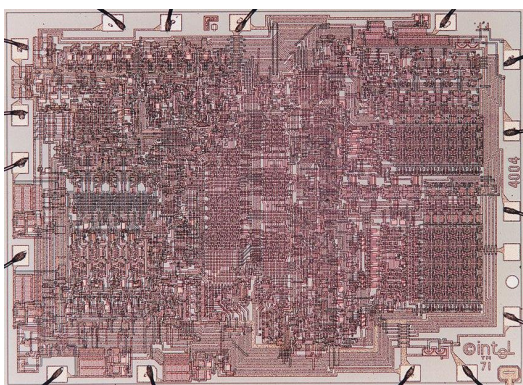
3.23	Some “quality of life” methods	66
4.1	Wire3 entity	72
4.2	D2M Wrapper	74
4.3	MTJ instance	74
4.4	D2M analog section	75
4.5	D2M output control	75
4.6	FReader VHDL description	76
4.7	FReader’s struct in C	77
4.8	FReader’s init function	77
4.9	FReader’s callback	78
4.10	Makefile for the FLI components	79
4.11	FWriter VHDL description	79
4.12	FWriter’s struct in C	79
4.13	FWriter’s init function	80
4.14	FWriter’s callback	80
4.15	Testbench	81
4.16	Simulation commands	82
4.17	Launching the simulation	83
4.18	Python script to check the outputs	83

Chapter 1

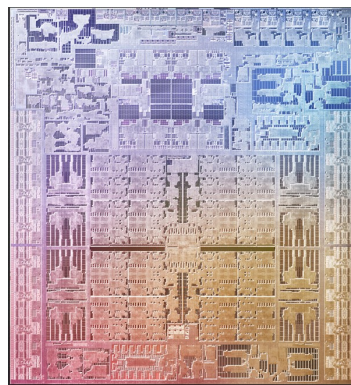
Introduction

1.1 Beyond CMOS

Silicon has been the very essence of electronics since the diffusion of the first transistor based ICs in commerce in the 1970s. Throughout the years, continuous efforts and advancements in fabrication processes have been perfecting our devices: compare the Intel 4004 chip, historically regarded as the first commercial microprocessor, counting 2300, $10\mu\text{m}$ transistors to the latest Apple chip counting 60 billion, 5 nm transistors. These monumental improvements did not come without challenge: as the scale of the transistors gets smaller, parasitic effects, losses and heat dissipation become harder and harder to control. The seemingly unstoppable trend that shaped modern society through technological advancements may be close to a historical turning point: what happens when the challenges become greater than the benefits they promise?



(a) Layout of Intel 4004 chip



(b) Layout of the Apple M1 Max chip

Figure 1.1: Comparison (not to scale) of the layouts of the first Intel chip and one of Apple's latest releases

This very question has inspired the birth of many *Beyond CMOS* technologies,

technologies based on innovative techniques, materials and physical phenomena that promise a continuation to the trend of constant improvement in electronics. Some Beyond-CMOS technologies have already seen the light and have real world applications: for instance, MTJ devices are currently gaining strong traction in Neural Networks [8], MRAMs and logic-in-memory architectures [10]; not to mention the increasing fame of Quantum Computing and the interest sparked in multiple industries such as telecommunications [2].

Even though research is increasingly active in these fields, it is unlikely that an all-solving technology will suddenly replace silicon: each technology has its own benefits and is best suited for some tasks, especially at this primordial stage. Moreover, silicon has not only been the de-facto standard for the last 50 years, but it is also extremely versatile and can be used in digital devices as well as in analog circuits and even in optoelectronics. Based on this, a more likely scenario for the future is either the affirmation or at least a transitory period of *Hybrid Circuits*: circuits composed of multiple sub-parts featuring and interconnecting multiple technologies.

1.2 Simulations

The extremely high costs of fabrication make simulations a necessity in the development of modern day electronics. Simulations allow engineers not only to check the correct functionality of their design at every step of the development process, but also to predict important parameters such as power consumption, area, speed, emissions and EMC, fault tolerance and much more before ever producing a single chip.

In generic terms, circuit simulators aim to represent the state of their parts over time. At the most basic level, this translates into updating and tracking voltages and currents throughout the circuit. The simulation algorithms can vary a lot as simulators try to optimize as much as possible for their specific application, but the one always present concept is the internal representation of time. Based on time management, it is possible to identify two main families of simulators: analog and digital simulators.

- **Analog Simulators** simulate continuous quantities whose details are infinitely fine in the real world. The evolution of these quantities is determined by the solution of a system controlled by the characteristic equations of the analog components, a process that is often computationally expensive. On top of this, computers can only handle finite and discrete quantities, so real world behaviors must be approximated in discrete time steps and the job of a simulator is also to decide how to step in time. A basic simulator may choose to advance the

simulation time with fixed time steps which are small enough to approximate the behavior of the simulated circuit. A more common and optimal technique is to advance simulation time based on the variations of the tracked quantities, dynamically adjusting the effort and increasing resolution when there are a lot of variations while taking bigger leaps through times in stable conditions that do not need as much resolution.

- **Digital Simulators** handle discrete quantities, implying that changes through time are instantaneous: in any time instant, the tracked quantities are constant and changes can be assumed to happen instantaneously. Moreover, there are no equations to solve but defined high level behaviors to simulate. Thanks to this, digital simulators are typically faster than analog simulators. For the same reasons, they are *event-based*: they operate with an event queue from which the next event is popped and handled, possibly spawning other events that are pushed into the queue. In this scenario, time is no longer a controlling factor as much as another quantity being tracked. It is important to underline how despite this structural change, the concept of time resolution is still needed to maintain the causality of certain events and it is useful in various cases such as the representation of *glitches*.

Naturally, this distinction is purely conceptual and real simulators often work in-between the two cases: for instance, simulating power consumption needs analog-like simulations, and a digital simulator unable to do so is somewhat incomplete. It follows that a Hybrid Simulator should also be based upon a hybrid approach, both for a matter of necessity - some beyond CMOS technologies *require* analog simulations - and, because given the novelty of these technologies, it is often desirable to simulate their behavior in levels of detail unachievable by purely digital simulations.

1.3 ToPoliNano Framework

ToPoliNano is an EDA tool developed by the researchers and thesis students of the VLSI group in Politecnico di Torino. It offers two pieces of software: MagCAD and ToPoliNano.

MagCAD MagCAD [6] is a Computer Assisted Design tool intended for the graphical design of digital circuits. Choosing from a range of different technologies, the user is able to pick-and-place components, form interconnections and customize their properties. Designs can be saved, imported and exported and it is also possible

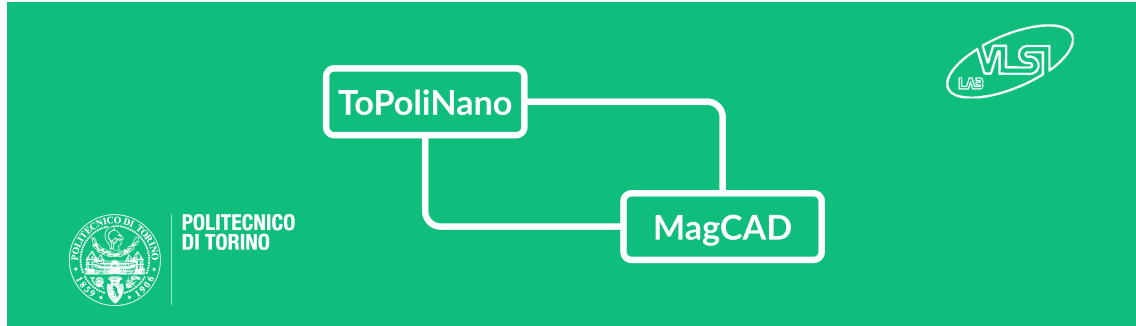


Figure 1.2: ToPoliNano framework

to develop custom libraries of components to be re-used in multiple designs. Depending on the technology, it is also possible to convert layouts into VHDL netlists to be used in conjunction with other VHDL files in any digital simulator.

ToPoliNano ToPoliNano [7] is a tool able to inspect and simulate designs. It takes the output files from MagCAD and, together with other files that the user imports (such as testbenches or other VHDL components), it runs the simulation and reports the results. It also offers helpful tools such as the automatic generation of testbench files which the user can customize by controlling the input stimuli without having to worry about the rest of the VHDL structure.

Chapter 2

MagCAD

2.1 Introduction

MagCAD offers a customizable design environment where it is possible to construct circuits and develop their layout based on a range of different technologies. [Figure 2.1](#) shows an example view of a circuit being interconnected with the Hybrid

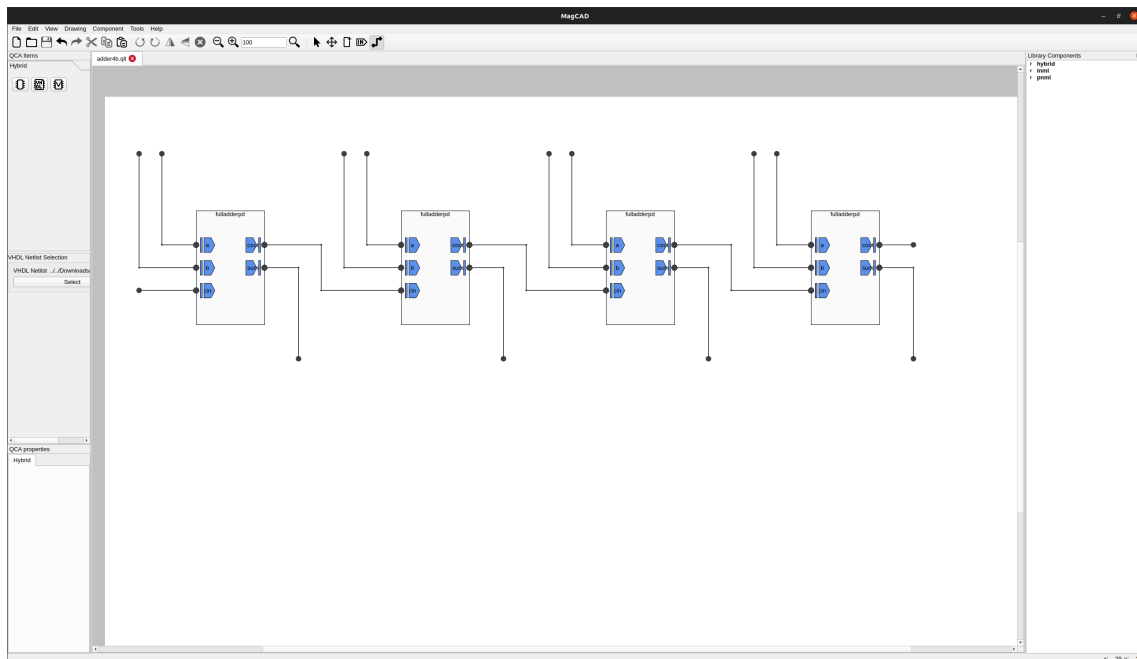


Figure 2.1: Standard view in MagCAD

Plugin that will be introduced later in this chapter.

When launching the program, the Drawing Settings window of [Figure 2.2](#) lets the user choose the technology they want to design their circuit with. It lists all the available technologies and allows the customization of some technological parameters, some of which can also be locally overridden later on from the Item Picker.

Once the settings are filled, the user is met with the window of [Figure 2.1](#) and can

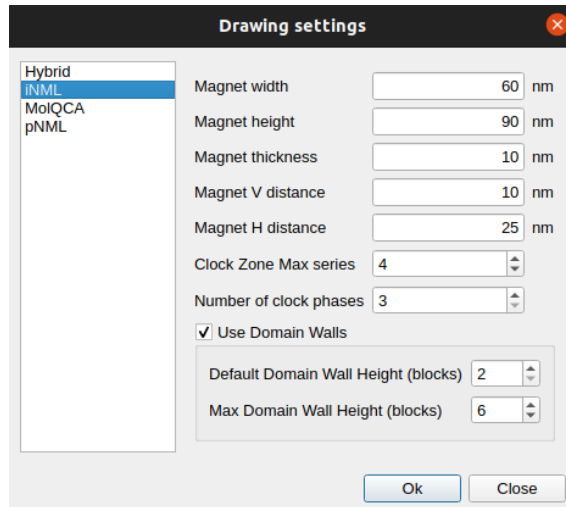


Figure 2.2: Drawing Settings window

at this point proceed with his own design.

It is possible to observe that the main window is composed of four sections:

- **Menus and Toolbars:** on the top of the window, they offer an interface to the standard functionalities such as saving, exporting and loading designs and all the tools to interact with the scene;
- **Drawing Window:** in the middle of the window, this is the region where items and components are placed and interconnected;
- **Item Picker:** on the left of the window, this bar presents all the standard items available for the currently picked technology plus the context-specific, per-item editable properties;
- **Library:** on the right of the window, lists all the saved components to be re-used across multiple designs.

In alternative to creating a new design from scratch, the user can also choose to load an already existing design and edit it. In this case, the Drawing Settings are automatically imported from the saved file as well as all the whole content from the original design, making it unnecessary to manually setup the technology and simplifying the process of sharing and updating projects.

On top of saving a design, it is also possible to export it as a component and to store it in the Component Library, meaning that the circuit can be instantiated as a sub-unit in larger structures. Only components from the currently selected technology can be inserted in a design. In order to place a component, the user can **Right-Click** on its name in the Library and choose to:

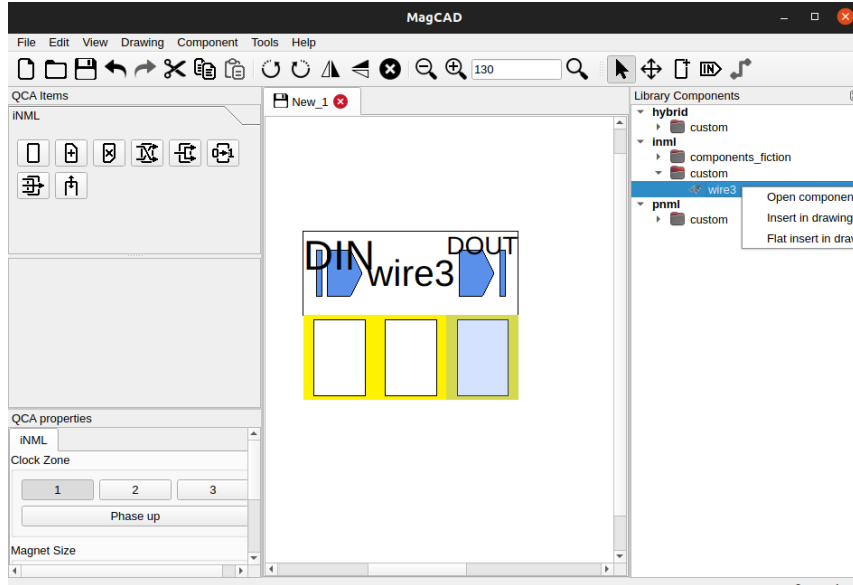


Figure 2.3: Component insertion

1. **Open component** to view the component in a read-only window;
2. **Insert in drawing** to insert the component as a *black box* in the design;
3. **Flat insert in drawing** to insert the exposed contents of the component as if they were copy-pasted. This allows smaller tweaks in larger components and it is also particularly helpful in layout-centered technologies where a whole black-box may take up too much space.

Figure 2.3 shows a view of the insertion of a component, where both an insert (top) and a flat-insert (bottom) operation of the same circuit were carried out. In order to let the insides of a component interact with the outside world, the user can place named *pins* in their designs, representing the input / output ports of the component. When importing such a component, pins will appear on the sides of its black box as visible in Figure 2.3 and will be exported to VHDL (if the technology supports it); on the other hand, flat-inserting the same component will not insert its pins as they are unnecessary and may lead to errors.

2.2 Program Structure

MagCAD is developed in C++ using the Qt framework as the base for the GUI and as a reference library supplying useful containers, functions and compiler definitions. It is currently compiled using `qmake` even though a transition towards `CMake` is in act due to the approaching end of support for `qmake` for newer versions of Qt.

Its core is divided into three parts:

- **MagCAD**: the main executable itself. It initializes the `QApplication` (Qt's fundamental entity that represents an executable program) and the main window with all of its sub-parts, implying it is the manager of all the program-level tasks such as managing its settings, opening and reading save files and populating the Components Library;
- **qcalib**: central library gathering all the concepts that are technology and design related. For example, this library defines classes such as `QcaTechnology` and `QcaItem` that are the bases for almost everything in a drawing scene;
- **qcagraphicslib**: graphical library responsible to handle the interactions with the scene, including the placement, movement and interconnection of the items.

A particularly effective design choice employed by MagCAD is the extensive use of *plugins*. Plugins are optional extensions that mainly represent a technology, its

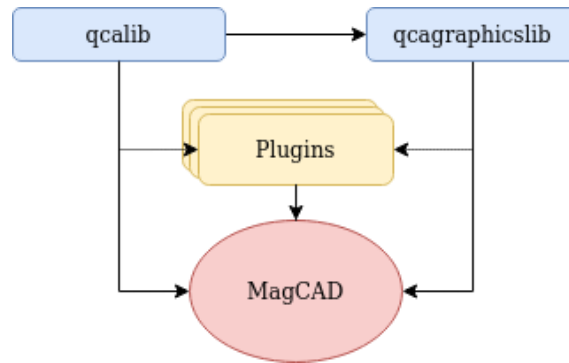


Figure 2.4: MagCAD structure

items and its functions in MagCAD. They are compiled separately from the main executable into *shared libraries* that need to be placed in a specific directory. During its bootstrap, MagCAD will look into that directory and will load all the plugins it finds. The Drawing Settings window from [Figure 2.2](#) is showing the results of that very operation: the entries on the left are none other than the names of the technologies that MagCAD found in its plugin directory when launched.

The most important elements introduced from a plugin are:

- **Technology**: the top level entity in a plugin, it offers information about the plugin itself and about the technology it represents. On top of listing relevant information such as a list of the building blocks offered by the technology, it also allocates and initializes other sub-parts of the plugin itself;
- **Settings**: settings collect the parameters mainly related to the drawing, such as the grid size, and they offer some related methods useful to the drawing classes;

- **Items:** they are the basic building blocks of the design and are the elements that are picked from the Item Picker.

section 2.3 describes the implementation of the Hybrid Plugin, where a more in-depth analysis of the different parts of a plugin are explored in clearer details.

2.3 Hybrid Plugin

The Hybrid Plugin is an extension that allows the design of Hybrid circuits containing sub-elements from mixed technologies. As introduced in [section 2.2](#), the implementation of a new plugin for MagCAD starts from the inheritance of some of the classes offered by `qcalib`.

The interface of a plugin is given by its *Technology* class. It provides some general information about the technology itself such as its name and plugin version together other methods needed during the design phase.

```

1  class HybridTechnology : public QObject, public QcaTechnologyInterface {
2      Q_OBJECT
3      // Macros used to define Qt plugins
4  #if QT_VERSION >= 0x050000
5      Q_PLUGIN_METADATA(IID "it.polito.ToPoliNano.graphics.QcaTechnologyInterface"
6                       FILE "hybrid_plugin.json")
7  #endif
8      Q_INTERFACES(QcaTechnologyInterface)
9  public:
10     static const QString tech_name;
11     HybridTechnology();
12
13     QString name() const override;
14     QString version() const override;
15
16     QStringList providedElements() override;
17     VhdlController *getVhdlController() override;
18     QcaItem *createItem(const QString &name, QcaTechSettings *settings = nullptr)
19         override;
20     QList<QcaToolButton *> createTools(QcaTechSettings *settings) const override;
21     QcaTechSettings *createTechSettings() override;
22 };

```

Listing 2.1: HybridTechnology class declaration

The `providedElements()` method returns the list of names of the items that this technology implements. In this case:

```

1  QStringList HybridTechnology::providedElements() {
2      QStringList list;
3      list << "Component"
4            << "VHDL Netlist"
5            << "Verilog Netlist";
6      return list;
7  }

```

Listing 2.2: Names of the items provided by the Hybrid Plugin

These are the items that are available to select from the Item Picker to place them in the design window.

The `createTools()` method creates the buttons that can be seen in the Item Picker from [Figure 2.1](#) and in [Figure 2.7](#). Each button corresponds to an item and pressing it will put the scene in “insert item” mode so that the next clicks in the

design space will place the component. Buttons derive from a base class called `QcaToolButton`, but this specific type of buttons in fact derives from an extension of that base class called `QcaFactoryButton` that specializes a generic button for the creation of items.

```

1 HybridComponentFactoryButton::HybridComponentFactoryButton(QcaTechSettings *
    settings, QWidget *parent):
2     QcaFactoryButton(settings, parent),
3     m_settings(settings),
4     m_component_properties_widget(nullptr) {
5     // Set button
6     setIcon(QIcon(":/images/component.png"));
7     setIconSize(QSize(24, 24));
8     setCheckable(true);
9     setToolTip("Component");
10    m_component_properties_widget = new HybridComponentPropertiesWidget(settings);
11 }

```

Listing 2.3: Example of a FactoryButton constructor

Buttons also have some other responsibilities. The behavior of a given button may need some parametrization: as it will be more thoroughly explained in [section 2.4](#), this is the case for all Hybrid items since they act as wrappers for components or netlists and it is necessary to have the user select the file that the item is supposed to wrap. Clicking on these factory buttons opens a small widget below the Item Picker called “Option Widget” that can be customized to enter the required parameters. For example, a VHDL Netlist item will open the widget shown in [Figure 2.5](#). This

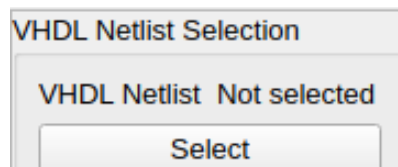


Figure 2.5: The “Option Widget” for a VHDL Netlist item

is because a VHDL netlist is associated to a file stored in the filesystem that needs to be specified before being able to insert the item, since it is necessary to parse the file in order to understand the interface of the item. Trying to insert an item before a VHDL Netlist is selected results in the error of [Figure 2.6](#). In fact, all

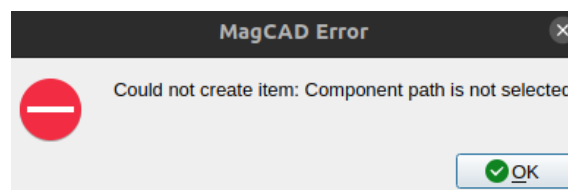


Figure 2.6: Error shown when trying to insert an Hybrid item before selecting its associated file

Hybrid items share this Option Widget because they all share the same behavior, the only difference being the type of file that can be selected and of course the text in the widget, but the basic idea is the same. [Listing 2.4](#) shows the creation of a VHDL Netlist item with the checks to ensure that the path has been selected. The exceptions thrown in this function are caught to populate the error window in [Figure 2.6](#).

```

1 QcaItem *HybridVHDLNetlistFactoryButton::createQca() {
2     // Check if component path is selected
3     HybridTechnology *htech = (HybridTechnology*) technology();
4     HybridVHDLNetlistPropertiesWidget *pwidget = (HybridVHDLNetlistPropertiesWidget
5         *) m_vhdlnetlist_properties_widget;
6     if (!pwidget->isPathSet()) {
7         throw QCAItemError("Component path is not selected");
8     }
9     // Create item and set path
10    HybridItemVHDLNetlist *item = (HybridItemVHDLNetlist*) htech->createItem("VHDL
11        Netlist", m_settings);
12    if (item == nullptr) {
13        throw QCAException("Could not create item");
14    }
15    item->setItemFilePath(pwidget->getPath());
16    item->compile();
17    return item;
18 }

```

Listing 2.4: Creation of an item through its factory button

The `getVhdlController()` method that appears in [Listing 2.1](#) is used to return the technology’s VHDL Controller, which is a separate class needed to generate VHDL files for the technologies that support it. The Hybrid technology does not currently support general VHDL generation, so it simply returns a `nullptr` that will be recognized from the other functions trying to interact with it.

2.4 Hybrid Items

Items are the central entities of a design since they are the elementary building blocks offered by a given technology. Another role of a plugin is to implement classes that inherit from `qcalib`’s `QcaItem` that define the items that can be placed in a design by managing their internal attributes, their appearance, their size and all of the necessary parameters. The Hybrid Plugin has three basic items called “Component”, “VHDL Netlist” and “Verilog Netlist”.

The item called “Component” is not to be confused with the Library Components: even though their role is very similar, the item is in fact a wrapper for a library component and this is needed to generalize the library component which is tightly bound to its internals. Layout centered technologies such as iNML and pNML



Figure 2.7: Hybrid Items icons from the Item Picker

need their library components to be correctly sized in order for the design to make sense, so library components have to adapt their shape based on the placement of their inner items. Since MagCAD was originally built for this type of design, the generic library component classes are structured to follow this behavior. The constraint on sizes does not apply to the Hybrid Plugin and unnecessarily placing large components simply clutters the drawing, so a component item was chosen as a solution since it can be fully customized without damaging the other parts of the program. Similarly to the Component item, the VHDL Netlist and Verilog Netlist items act as wrappers for their respective netlists and are needed to both implicitly include CMOS circuits in the design as well as helpful testbench utilities.

It is already clear how all the described items are in fact similar in behavior: they wrap some content described in a file, whether it is a saved library component or a VHDL or Verilog netlist, and they need to adapt their appearance (namely, the input/output pins) based on the contents of the file. For this reason there is a common base class from which they all descend.

```

1  class HybridItemWrapperBase : public HybridItemBase {
2  public:
3      HybridItemWrapperBase(HybridTechnology *technology, const QString &item_name);
4      ~HybridItemWrapperBase() override;
5
6      void draw(QPainter *painter, const QStyleOptionGraphicsItem *option, QRectF
          cell) override;
7
8      QString getItemFilePath() const;
9      void setItemFilePath(const QString &path);
10
11     virtual void compile() = 0;
12
13     const QString &getItemName() const;
14
15     VhdlItem *copyAsVhdl() override;
16
17     QList<QPointF> connectionPoints() const override;
18
19 protected:
20     static QString getFileName(const QString &path);
21     QSize computeCellSpan();
22     void updatePinPositions();
23
24 protected:
25     QString m_item_name;

```

```

26     QList <QcaPin*> m_pins;
27 };

```

Listing 2.5: HybridItemWrapperBase class declaration

The `draw()` method is an alias for Qt's `paint()` method and it defines the operations needed to graphically draw the item once it is placed in the scene. Notice that a `QcaItem` is not an object that can be directly placed in Qt's `QGraphicsScene` (the drawing window) because that requires to inherit from the `QGraphicsItem` class, but `QcaItem` does not. The `qcagraphicslib` uses another class called `QcaGraphicsWrapper` which, as the name suggests, wraps an item to place it in the scene and calls its `draw()` method when it needs to be rendered. The reason for this is to keep the graphical behavior separated from the functional behavior, if not for `QcaItem::draw()` that is the only instance where an item runs graphical routines.

```

1  void HybridItemWrapperBase::draw(QPainter *painter, const QStyleOptionGraphicsItem
    *option, QRectF cell) {
2      ...
3      // Make drawing
4      painter->drawRect(cell);
5      // Paint pins
6      for (QcaPin *pin : m_pins) {
7          QRectF pin_cell(pin->pos().x() * grid_size.width(),
8                          pin->pos().y() * grid_size.height(),
9                          grid_size.width(),
10                         grid_size.height());
11          pin->draw(painter, option, pin_cell);
12          painter->drawText(pin_cell, Qt::AlignCenter, pin->name());
13      }
14      // Write labels
15      QFont font = painter->font();
16      font.setPixelSize((int) cell.width() / m_item_name.size());
17      painter->setFont(font);
18      painter->drawText(cell, Qt::AlignTop | Qt::AlignHCenter, m_item_name);
19      ...
20 }

```

Listing 2.6: HybridItemWrapperBase `draw()` method

[Listing 2.6](#) shows an extract of the `draw()` method that renders a simple rectangle to represent the item, then draws its pins on top of it and finally writes the item name. It can be helpful to remind that this method is defined in a class from which all Hybrid items inherit, meaning this is used by all of them. The results of this operations are visible in [Figure 2.1](#) and [Figure 2.9](#).

The most important method defined in this Item class is the pure abstract `virtual void compile()` and it represents the action of opening the associated file, reading its contents and extracting the useful information. Notice, for instance, how the `draw()` method iterates through an item's `m_pins`, a vector which needs to be populated according to the content of the file: that is precisely what `compile()`

does. Notice also that a call to it already appeared in [Listing 2.4](#), since an item is compiled immediately before placement.

```

1 void HybridItemVHDLNetlist::compile() {
2     // Get name
3     QDir base = QDir::home();
4     base.cd("MagCADFiles/lib");
5     QString path = base.filePath(getItemFilePath());
6     if (path.isEmpty() || !QFileInfo::exists(path))
7         throw QCAException("Cannot compile item: path is invalid");
8     m_item_name = getFileBaseName(path);
9     // Parse file
10    QFile file(path);
11    file.open(QIODevice::ReadOnly);
12    VHDLEntityParser parser;
13    if (parser.parse(file) != Parser::ParseResult::Success) {
14        ... [Report problem with a message box] ...
15    }
16    // Process parsed data
17    VHDLEntityParser::ParsedData parsed_data = parser.getParsedData();
18    for (const VHDLEntityParser::PinData &pin_data: parsed_data.pins) {
19        m_pins.append(new QcaPin(technology(), pin_data.name, QcaPin::Direction(
20            pin_data.direction)));
21    }
22    // Update item shape
23    QSize span = computeCellSpan();
24    cellRect.w = span.width();
25    cellRect.h = span.height();
26    ...
27 }

```

Listing 2.7: The compile() method

Hybrid wrapper items collaborate with a set of other classes inheriting from an abstract base called `Parser` as shown in the diagram of [Figure 2.8](#) that can be used from the overrides of the `compile()` method.

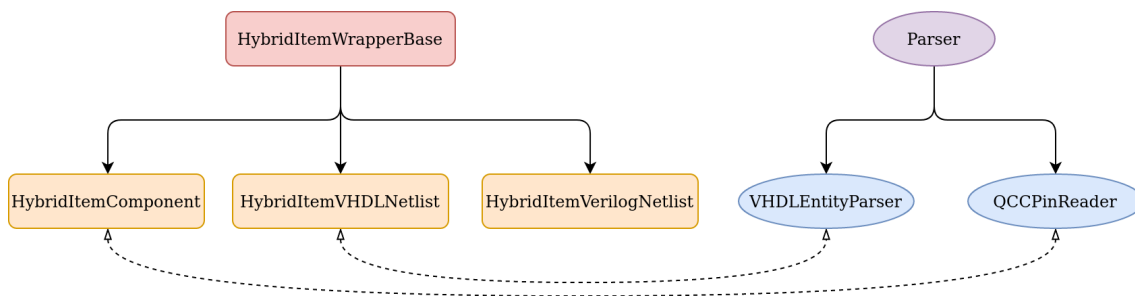


Figure 2.8: Inheritance graph for Hybrid wrapper items and parsers and their relations

```

1 class Parser {
2 public:
3     enum class ParseResult {
4         Success,
5         Fail

```

```

6     };
7
8     class ParsingError;
9     using Token = QString;
10
11 public:
12     Parser();
13     ParseResult parse(QFile &qfile);
14     Parser &operator=(const Parser &parser);
15
16     const QList<ParsingError> &getErrors() const;
17
18     static QList<Token> tokenize(const QString &text, const QRegExp &regex =
        QRegExp("\\b|\\s+"));
19
20 protected:
21     virtual ParseResult runParse(QFile &qfile) = 0;
22     void reportError(int id, const QString &message, std::size_t line = 0, bool
        critical = false);
23
24 protected:
25     QString m_fname;
26     QList<ParsingError> m_errors;
27 };

```

Listing 2.8: Abstract Parser class

The typical objective of a parser is to determine the number of pins and their direction, but for the sake of generality this base class only defines the actual parsing methods, and it is up to the deriving classes to implement their data structures to collect and return their custom results. Since this kind of parsers are supposed to stay simple - a fully fledged compilation at the design stage would heavily slow down the process -, all of its methods are kept simple as well. For instance, a Token does not contain any metadata and directly corresponds to its text. According to the same philosophy, the `tokenize()` method that can be used to split an input line into a list of tokens is a one-liner that splits the input text with a simple regex. More complex parsers will require a proper Token and tokenizer infrastructure.

More in-depth information about the Parser class is presented in [section 2.5](#) that shows how the VHDL parser reads data from a VHDL listing. Just as a final result example, [Figure 2.9](#) shows an instance of a successfully compiled component item called “wire3” (the same that was placed as a library component in [Figure 2.3](#)), built with iNML technology, together with a VHDL item called “fulladderpd”, described in VHDL, placed in a design.

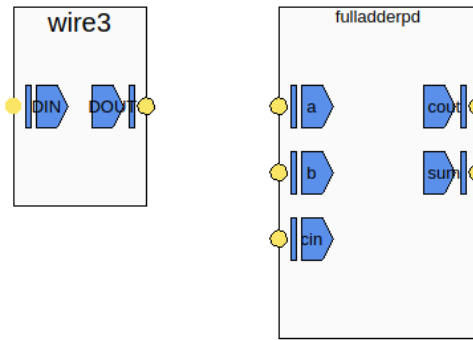


Figure 2.9: Items placed in the scene

2.5 Parser Example: VHDL Parser

Let us take the VHDL parser as the main example to analyze, knowing that the other parsers are organized quite similarly with case-specific adaptations. Given that the target of this elaboration is not to generate a complete VHDL model nor to check the correctness of the netlist, but simply to infer the shape of the item to insert in the scene based on its number of inputs and outputs, it is sufficient to only analyze the `entity` declaration. Notice that this is the reason why this procedure is being referred to as “parsing” and not “compiling” (even though the central method is called `compile()`). For simplicity, generics and data types other than `std_logic`, `std_logic_vector` and `real` are not currently supported.

```

1  class VHDLEntityParser : public Parser {
2  public:
3      struct PinData {
4          QString name;
5          int direction;
6      };
7
8      struct ParsedData {
9          QString name;
10         QList<PinData> pins;
11     };
12
13 public:
14     VHDLEntityParser();
15     Parser::ParseResult runParse(QFile &qfile) override;
16
17     void reset();
18
19     const ParsedData &getParsedData() const;
20     static bool isIdentifierValid(const QString &identifier);
21
22 private:
23     Token nextToken();
24     QString readNextLine();
25     ...

```

26 };

Listing 2.9: VHDLParser class

The VHDL parser reads the file word by word and feeds the inputs to a basic state machine. The results of the parsing step are stored in a private member of the custom type `ParsedData`, which can be accessed with `getParsedData()` as already shown in [Listing 2.7](#). `ParsedData` stores the name of the analyzed entity plus a list of `PinData`, where each entry collects information about the parsed pins, specifically their name and direction.

In order not to over-engineer this relatively small task, the tokens emitted while scanning the file do not have any metadata as it is sufficient to only look at their text. The parser is implemented like a state machine whose states and transitions are shown in [Figure 2.10](#). It is possible to observe that the behavior of this parser is quite linear as the only branching happens when elaborating types; all other instances show how a transition is either resolved into a unique and specific state if the expected entry was met or results in an error otherwise. The model entity declaration upon which the parser is based is the following, where spacing and capitalization do not matter since VHDL is case insensitive:

```
entity <entity name> is
    [generic (...)]
    port (
        <signal>[, <signal>[...]] : <direction> {<basic type> | <ranged type>(<start> <direction> <end>)};
        [other signals]
    );
end entity [<entity name>]
```

What the diagram in [Figure 2.10](#) does not show is what happens inside of the states. In order to also give an idea of how this is implemented, the general structure of the parsing function looks like this:

```
1 ParseState state = WAITING_FOR_ENTITY_DECLARATION;
2 while (!m_stream.atEnd()) {
3     Token token = nextToken();
4     switch (state) {
5         case WAITING_FOR_ENTITY_DECLARATION:
6             ...
7         case READING_ENTITY_NAME:
8             ...
9             ...
10        default:
11            reportError(1, "Unexpected internal state", m_line_n);
12            return ParseResult::Fail;
13    }
14 }
```

Listing 2.10: Structure of the VHDL parser loop

Where an example of a single case is

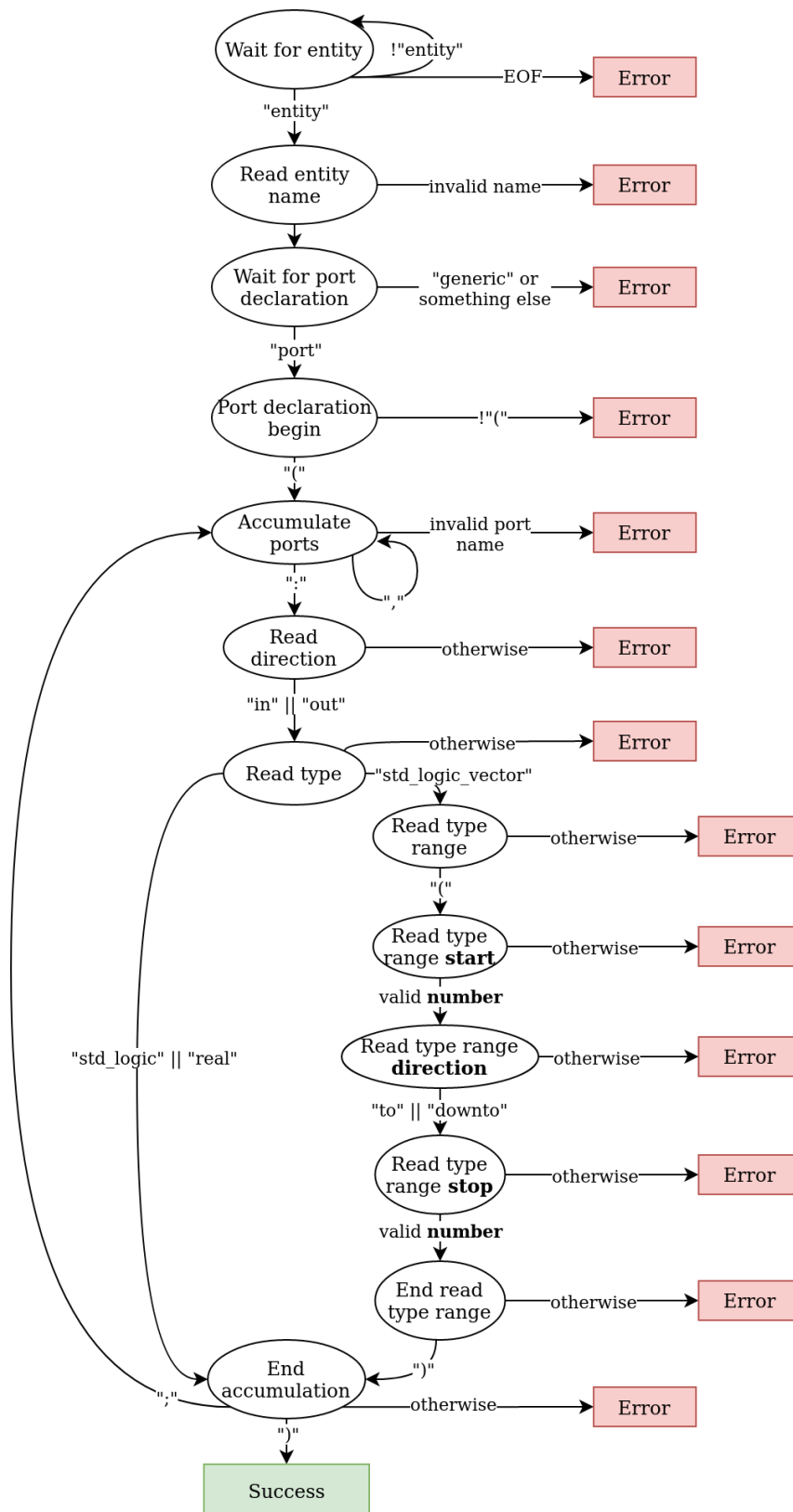


Figure 2.10: VHDL parser state machine

```

1 case ACCUMULATING_PORTS:
2     if (cinsensitive_text == ",") {
3         state = ACCUMULATING_PORTS;
4     } else if (cinsensitive_text == ":") {
5         state = READING_DIRECTION;
6     } else {
7         if (isIdentifierValid(token)) {
8             accumulated_ports << token;
9         } else {
10            reportError(1, QString("Invalid port name: '%1'").arg(token), m_line_n)
11                ;
12            return Parser::ParseResult::Fail;
13        }
14    } break;

```

Listing 2.11: Example of the actions in a parser state

After successfully parsing the type (either directly in “Read type” or “End read type range” if the type is ranged), the parser finally collects the `accumulated_ports` together with their type and dumps this information in the `ParsedData`.

```

1 case READING_TYPE:
2     ports_type = cinsensitive_text;
3     if (ports_type == "std_logic" || ports_type == "real") {
4         // Save ports and start reading the next bunch
5         for (const QString &port_name : accumulated_ports) {
6             PinData data;
7             data.name = port_name;
8             data.direction = direction;
9             m_parsed_data.pins << data;
10        }
11        state = END_ACCUMULATION;
12    } else if (ports_type == "std_logic_vector") {
13        state = READING_TYPE_RANGE;
14    } else {
15        reportError(1, QString("Unknown type '%1'").arg(token), m_line_n);
16        return Parser::ParseResult::Fail;
17    }
18    break;

```

Listing 2.12: Collection of `ParsedData`

Both of these methods also show the reporting of an error. Notice that comments and empty lines are automatically discarded by the `nextToken()` function so that there is no need to handle them by this state machine.

In case of a successful parsing, the item can just be placed in the scene as shown in [Figure 2.9](#). Whenever an unexpected situation occurs, the parser can choose to report the error and quit or to store the error and continue with the operation in case it is possible to recover from it somehow, but this does not happen in this implementation. The parsing function returns a `ParseResult`, an enumeration that is either `Fail` or `Success`. Errors are reported through a class named `ParsingError`. The class contains an optional error ID, a message and information about the error

itself such as the name of the file that generated it as well as the line number. This class has a method called `toErrorString()` that collects said information into a formatted error string. With these classes, the typical parsing process looks like this:

```

1 VHDLEntityParser parser;
2 if (parser.parse(file) != Parser::ParseResult::Success) {
3     QMessageBox mbox;
4     QDir simple_path(path);
5     mbox.setText(QString("Compilation of file '%1' failed with %2 errors.")
6                     .arg(simple_path.canonicalPath())
7                     .arg(parser.getErrors().size()));
8     mbox.setStandardButtons(QMessageBox::Ok);
9     QString compilation_errors;
10    for (const auto &error: parser.getErrors()) {
11        compilation_errors += error.toErrorString() + "\n";
12    }
13    mbox.setDetailedText(compilation_errors);
14    mbox.exec();
15    throw QCAException("Could not parse file");
16 }
17 // Process parsed data
18 VHDLEntityParser::ParsedData parsed_data = parser.getParsedData();

```

Listing 2.13: Usage of a Parser (the missing part from [Listing 2.7](#))

To show an example of an error being generated, let us consider the following VHDL listing containing a declaration that uses an undefined type:

```

1 entity ErrorExample is
2     port (
3         x, y, z: in std_logic;
4         r: out MyNonexistantType;
5     );
6 end entity ErrorExample;

```

Listing 2.14: Example of a VHDL file with an error

When trying to import this entity into a design the compilation fails and opens an error report window as in [Figure 2.11](#). It is important to point out that this parser

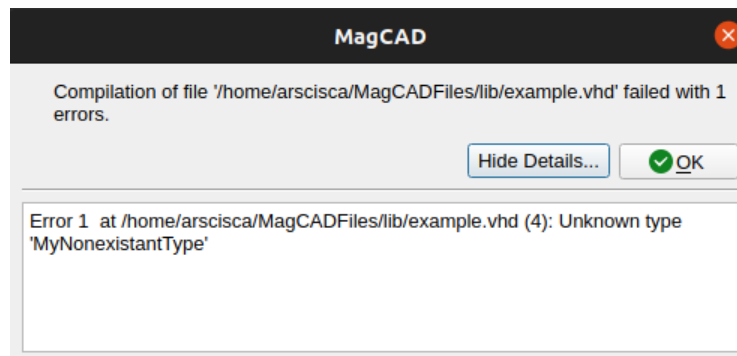


Figure 2.11: Example of parsing error

only considers one file at a time, implying that `MyNonexistentType` may in fact be declared somewhere else but will be considered invalid nonetheless.

This is a design choice dictated by a distinction between the design phase and the simulation phase: implementing such an extensive compilation would often require to compile multiple interdependent VHDL files, possibly including packages and other types of declarations. This would heavily impact performance and MagCAD's usability in larger designs. On top of this, this step is only looking to define input and output ports of an item so most of this processing would be wasted and would also be repeated at simulation time. The choice for this trade-off fell on a restrictive behavior, since an alternative would be the allowance of arbitrary types at the cost of letting more possible errors get to the simulation phase and thus requiring a longer and more twisted back-and-forth process between design fixing and trying to run the simulation.

2.6 Wiring

Unlike the other standard plugins developed for MagCAD up to this point, the Hybrid Plugin does not have a direct connection to the circuit layout, as the positioning of the components in the design does not have any implication at all. While other plugins consider items to be connected when they are adjacent, the Hybrid Plugin cannot follow this paradigm. Its higher level of abstraction requires another form of representation for interconnections, and the classical wires are the best option.

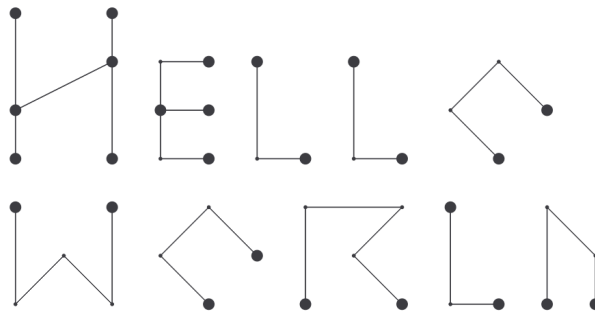


Figure 2.12: A greeting from wires in MagCAD

Wires do not necessarily represent a physical wire or any specific physical entity in general: at this level they only represent an interconnection. Interconnections can be freely created between any set of components regardless of their meaningfulness. For example, it is possible to connect an iNML component to a CMOS circuit even though there is no way that this direct connection can exist in the real world. This is because applying hard-coded rules to connections is in conflict with MagCAD's plugin philosophy as the user may want to make a meaningful connection between two circuits from two separate plugins, but would be forbidden to do so if the plugins do not know about each other's existence. For this reason, it is up to the designer to be careful about the connections they make, just like a programmer cannot rely on the compiler for checking the code functionality.

For the sake of generality, wires have not been implemented as an Hybrid-Plugin-only feature, instead they have been added as a standalone tool (as depicted in [Figure 2.13](#)) to MagCAD itself so that any other technology can choose to use them. Not all the technologies are supposed to interact with wires: in order for a technology



Figure 2.13: Icon for the wiring tool (the currently selected tool)

to be compatible with wiring, its `Settings` class which, as introduced in [section 2.2](#),

also collects generic parameters related to the drawing, must set a custom property called “wiring”. This property is tested from the main window and, accordingly to the currently operating technology, the icon from [Figure 2.13](#) may be grayed out and made unclickable if the “wiring” property is not enabled.

The complexity of this tool brought the concept of a `QcaSceneTool` to life in order to organize the code, its functions and datatypes. The abstract `QcaSceneTool` and its first implementation as a `WiringTool` are described in detail in [section 2.7](#) and the following sections; this section will instead introduce the general usage of the tool.

The main actions desired from this tool are:

- Creating and deleting nodes
- Connecting and disconnecting nodes
- Moving nodes

Some of the concepts and approaches have been inspired by [Blender](#), a famous and beautiful 3D graphics creation tool, and especially from its interactions with vertices in *Edit Mode* that closely resemble what this tool tries to achieve.

Selection Most of the functionalities of the wiring tool revolve around *selection*. A node can be selected by **Clicking** on it: selected nodes appear orange in color. **Clicking** on a node clears the previous selection, but **Clicking** while pressing **Shift** instead selects the new node while maintaining the old selection. **Right-Clicking** on

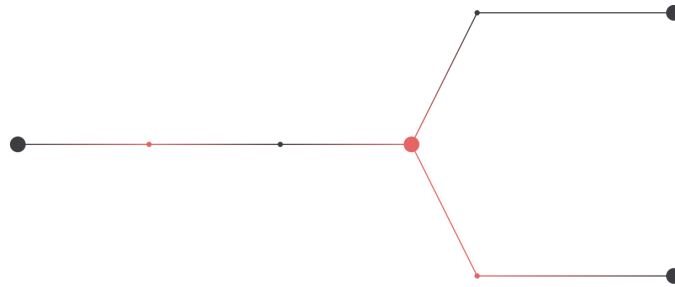


Figure 2.14: Some nodes selected at random using Shift + Click

an empty spot clears the selection. Note that being so central for the usage, changing the selection is an undo/redo-able action as will be further described in [section 2.7](#). **Double-Clicking** on a node selects its whole wire; simply pressing **A** is similar as it selects the whole wire of the last selected node. As from common practice across almost any program, **Ctrl + A** selects everything. Another interesting feature is the path selection: selecting one node and then **Alt + Clicking** another one selects the whole path between the two nodes as in [Figure 2.15](#). Pressing **Shift** in all of the

described actions still has the effect of not clearing any previously existing selection.

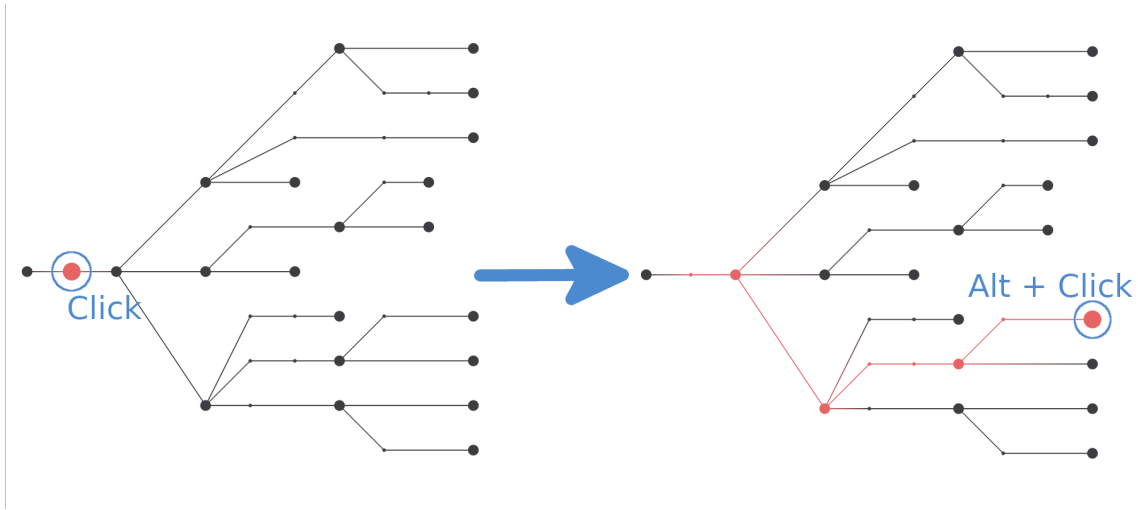


Figure 2.15: Example of path selection

Creating and Deleting Nodes To create a node, the user can simply **Click** on an empty spot: the rule is that no other entity should be in the same grid cell, including items or other wires. The only exceptions are item connectors, also introduced in [section 2.9](#), as they represent the connection point to an item. Creating a node also automatically selects it and the same rules from before apply: simply **Clicking** will select the new node and clear the old selection, pressing **Shift** keeps the old selection. The placed node is automatically connected to the last selected node, if any: this is the reason for automatically selecting the new node as it makes it a lot simpler to draw a wire. If this behavior is not desired, the user should clear the selection before placing a new node by right clicking on an empty spot.

There are two ways of removing nodes: *deleting* and *dissolving*. Selected nodes can be deleted by pressing **Del**, having the effect of removing the nodes and cutting the connections. Pressing **Alt+Del** instead dissolves the nodes, meaning the nodes are removed but a new connection is formed between their immediate neighbors in order to preserve the overall wire. This is only possible with nodes with two connections: trying to dissolve endpoints or branching points has no effect. [Figure 2.16](#) shows a comparison of the two methods.

Connecting and Disconnecting Nodes Nodes can be connected by selecting them and then pressing **C**. The selection-based framework enhances this functionality: it is possible to interconnect multiple nodes at once. If there are more than

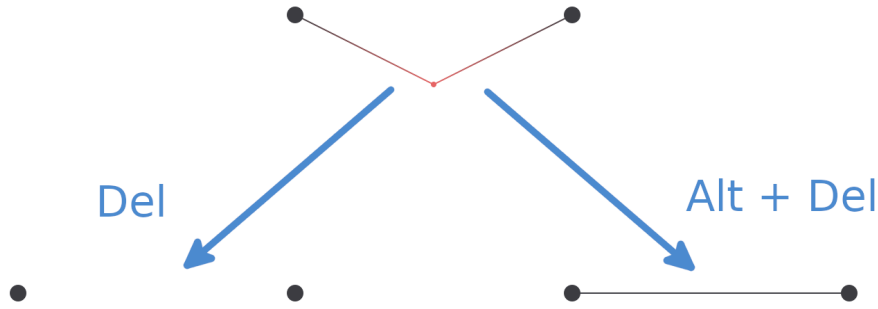


Figure 2.16: Methods of deleting nodes: standard **delete** on the left, **dissolve** on the right

two nodes currently selected, the tool will try to connect them in pairs according to the order of selection: suppose that the selection currently contains nodes $[n_0, n_1, \dots, n_k]$, pressing **C** will *attempt* to form connections $[(n_0, n_1), (n_1, n_2), \dots, (n_{k-1}, n_k)]$. The keyword “attempt” is important because there is a rule to form connections: wires cannot form closed loops - this is why letters like ‘O’ and ‘D’ in Figure 2.12 are not closed. The reason for this is that wired loops have no meaning in the design and are instead prone to introducing errors. For a better user experience, a failed attempt to connecting a single pair of node does not result in the failure of the whole connection operation and is simply skipped.

Disconnecting nodes works slightly differently: pressing **Shift + C** will clear *all* the connections of the selected nodes. This means that even if only two nodes are selected, **Shift + C** not only deletes the arc $n_1 \rightarrow n_2$, but also their other connections.

Moving Nodes Selected nodes can also be moved around by pressing **G**. This key toggles the “grabbing” mode in which selected nodes will follow the mouse movements. To exit from grabbing mode, the user can press **G** again and the nodes will be aligned with the grid and dropped. Just like when creating new nodes, dropping nodes is also restricted to free cells only: if any of the moved nodes collides with an item in the drawing, the movement is undone and nodes return to their original position.

A *context menu* can be opened by pressing **Shift + Right-Click** for an overview of some of the described functions plus some extra actions. For example, the *simplify geometry* command is only available from the context menu and has the effect of *dissolving* redundant nodes. Specifically, nodes are dissolved if they form an angle close to 180° to their neighbors, effectively flattening a wire that was already relatively

Select all	Ctrl+A
Select wire	A
Clear selection	
Connect nodes	C
Disconnect nodes	Shift+C
Delete nodes	Del
Dissolve nodes	Alt+Del
Simplify geometry	

Figure 2.17: Context menu from the wiring tool

flat. Branching nodes are simply ignored in this process. For reasons that are made

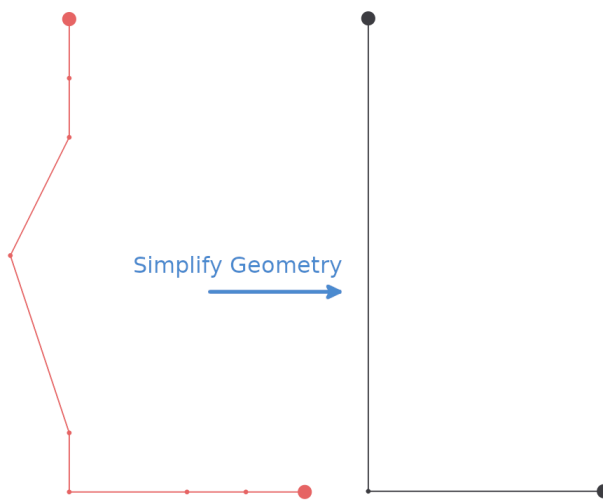


Figure 2.18: Effect of “simplify geometry”

clearer in [section 2.10](#), this happens inside a class called `SimplifyGeometryCommand`, from which an extract shows:

```

1 SimplifyGeometryCommand::SimplifyGeometryCommand(WiringTool &tool, QList<Node *>
   nodes, QUndoCommand *parent) :
2     WiringTool::WTUndoCommand(tool, parent) {
3     // Check for nodes to dissolve
4     for (Node *node : nodes) {
5         if (node->connectionDegree() == 2) {
6             Node *node1 = node->connectedNodes()[0];
7             Node *node2 = node->connectedNodes()[1];
8             if (angleBetween(node1, node, node2) >= SimplifyGeometryCommand::
               dissolve_threshold)
9                 new DissolveNodeCommand(tool, node, this);
10        }
11    }
12 }
13 ...
14 ...
15 ...
16 qreal SimplifyGeometryCommand::angleBetween(const Node *node1, const Node *node2,
   const Node *node3) {

```

```

17     QVector2D direction_21 = QVector2D(node1->pos() - node2->pos()).normalized();
18     QVector2D direction_23 = QVector2D(node3->pos() - node2->pos()).normalized();
19     qreal dot = QVector2D::dotProduct(direction_21, direction_23);
20     qreal angle = qRadiansToDegrees(qAcos(dot));
21     return angle;
22 }

```

Listing 2.15: Simplifying geometry

2.7 Wiring Tool

For a tidier organization and to pave the way for future expansions, the concept of Scene Tool was introduced, first of which is the Wiring Tool, responsible for all the features described in [section 2.6](#). The base class for the scene tools is called `QcaSceneTool` and it defines some pure virtual methods that descendants must implement plus various utilities and helpers. For instance, there are some methods to manage the activation of the tool:

```

1  class QcaSceneTool : public QObject {
2      ...
3  public:
4      ...
5      bool isEnabled() const;
6      void setEnabled(bool enabled = true);
7      void enable();
8      void disable();
9      ...
10
11  protected:
12      virtual void onEnable() = 0;
13      virtual void onDisable() = 0;
14      ...
15  };

```

Listing 2.16: Enable and disable handlers for `QcaSceneTool`

The pure virtual methods `onEnable()` and `onDisable()` are ran whenever the activation state of the tool is changed and allow it to perform some custom operations. For instance the Wiring Tool uses these methods to enable and disable the commands described in [section 2.6](#) so that wires cannot be interacted with when other tools are active.

The most important role of `QcaSceneTool` is the definition of some virtual event handlers as interfaces to the drawing scene.

```

1  class QcaSceneTool : public QObject {
2      ...
3  public:
4      ...
5      virtual void mousePressEventFallback(QGraphicsSceneMouseEvent *event);
6      virtual void mouseReleaseEventFallback(QGraphicsSceneMouseEvent *event);

```



```

7   virtual void mouseMoveEventFallback(QGraphicsSceneMouseEvent *event);
8   virtual void keyPressEventFallback(QKeyEvent *event);
9   virtual void contextMenuEventFallback(QGraphicsSceneContextMenuEvent *event);
10  ...
11 };

```

Listing 2.17: QcaSceneTool event handlers

To understand why the methods are called “fallback”, it is necessary to briefly introduce the Qt event system and how MagCAD’s `DrawingScene` class interacts with it. A `QGraphicsScene` is a Qt class that collects `QGraphicsItems` and `QGraphicsObjects`, the latter being just an extension of `QGraphicsItems` and will thus be treated interchangeably in this discussion. At some point of Qt’s event handling system, events arrive into the `QGraphicsScene` which, by default, forwards them to a specific `QGraphicsItem` that acts as a receiver. For instance, the receiver of a mouse press event is the item below the mouse. The receiver may decide to either process or ignore the event: in the former case the event is considered accepted and the procedure is completed; in the latter the event is forwarded to the item’s parent (if any). If an event is ignored from the whole parenting tree of the original item, it goes back to the scene. Figure 2.19 shows a schematic of this system.

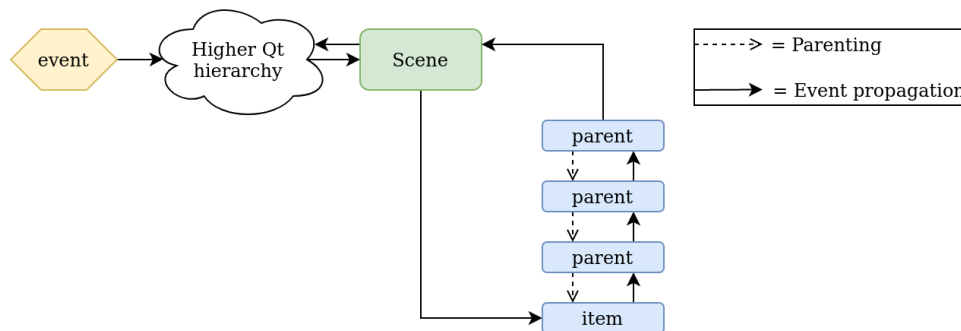


Figure 2.19: Overview of the Qt event system

With this in mind, the reason why the methods are fallbacks is because they are called only if no item has accepted the event. MagCAD’s `DrawingScene` is simply an extension of a `QGraphicsScene` that reimplements some of the event handlers described before. For example, the mouse press event handler:

```

1 void DrawingScene::mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent) {
2     // Check if any tool wants to handle this event
3     if (m_active_tool != nullptr) {
4         QGraphicsScene::mousePressEvent(mouseEvent);
5         if (!mouseEvent->isAccepted())
6             m_active_tool->mousePressEventFallback(mouseEvent);
7         return;
8     }
9     // Legacy actions not handled by a tool
10    ...

```

```
11 }
```

Listing 2.18: Example of a DrawingScene event handler

The call in line 4 is the one that forwards the event to the relevant item in the scene, so the fallback is called only if the item and its parents refuse the event or if there was no item to receive the event to begin with (e.g. a click on an empty spot).

By default, QcaSceneTool’s fallbacks just ignore the events. For instance:

```
1 void QcaSceneTool::mousePressEventFallback(QGraphicsSceneMouseEvent *event) {
2     event->ignore();
3 }
```

Listing 2.19: Default implementation of a QcaSceneTool event fallback

The derived classes such as the Wiring Tool can override them. Continuing with the case of the mouse press event, upon receiving it the Wiring Tool attempts to create a new node at the event location if the right conditions are met. The right conditions require that the grid cell in which the event is located must be inside the drawing region, free and no other actions like node moving must be ongoing.

```
1 void WiringTool::mousePressEventFallback(QGraphicsSceneMouseEvent *event) {
2     Qt::MouseButton mouse_button = event->button();
3     if (mouse_button == Qt::LeftButton) {
4         // Check if user is not doing other stuff
5         if (isGrabbing()) {
6             event->ignore();
7             return;
8         }
9         // Check if event position is valid
10        if (event->scenePos().x() <= 0 || event->scenePos().y() <= 0) {
11            event->ignore();
12            return;
13        }
14        // Check if any other item is in the clicked cell
15        if (!isCellFree(event->scenePos())) {
16            event->ignore();
17            return;
18        }
19        QUndoCommand *parent = new QUndoCommand;
20        CreateNodeCommand *create_command = new CreateNodeCommand(*this,
21            snapToCellCenter(event->scenePos()), parent);
22        // Implicitly select new node
23        bool clear_selection = !WiringTool::isKeyboardModifierPressed(Qt::
24            ShiftModifier);
25        ChangeSelectionCommand::selectNodes(*this, {create_command->node()},
26            clear_selection, parent);
27        if (lastSelectedNode())
28            new ConnectNodesCommand(*this, create_command->node(), lastSelectedNode
29                (), parent);
30        undoStack().push(parent);
31    } else if (mouse_button == Qt::RightButton) {
32        if (!isGrabbing() && !isKeyboardModifierPressed(Qt::ShiftModifier))
33            clearSelection();
34    }
35 }
```

31 }

Listing 2.20: Full implementation of WiringTool’s mouse press event fallback handler

Lines 4 to 18 implement the mentioned checks; lines 19 to 26 finally create the new node. The usage of `QUndoCommands` is required for the standard undo / redo functionalities in Qt and are discussed more in depth in [section 2.10](#), at this point it is sufficient to mention that they handle the allocation and deallocation of the objects and that they run their function when they are pushed into the undo stack (line 26). They are instantiated with `new` as Qt requires and will be automatically deleted by the stack itself. Lines 22 and 23 show the check for clearing the previous selection if `Shift` is not pressed and the update of the selection itself inserting the newly created node; lines 24 and 25 instead show the automatic connection to any previously selected node that happens when creating a new node.

On top of event handling, tools are also intended to organize the items in the scene and the Wiring Tool collects all the information related to wiring. Terms such as “node” and “wire” have been used loosely up to this point, but they have specific meanings in the code. All the items managed by the Wiring Tool follow the hierarchy

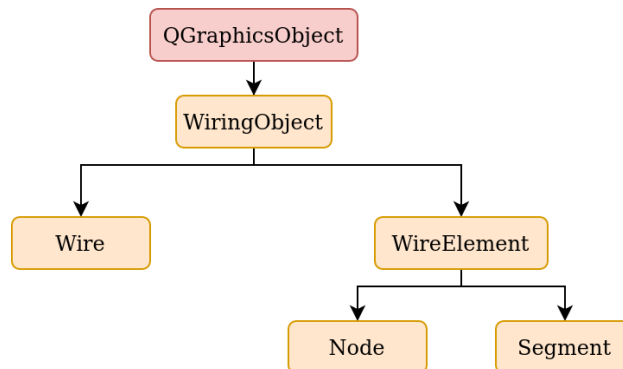


Figure 2.20: Hierarchy of classes used in the Wiring Tool

from [Figure 2.20](#) to group common functionalities. A `Wire` is mostly a container that keeps track of its `Nodes` and offers graph-like algorithms that can be used in various points of the program, for instance the `Wire::pathConnecting()` method returns a path between two nodes passed as arguments used for selecting parts as described in [section 2.6](#). `Segments` are looser entities that only bare graphical information. All of these elements are better described in [section 2.8](#).

By grouping elements this way, it is very simple to interact with the entities used by a tool. A loop such as this one:

```

1 for (Wire *wire : wires()) {
2     for (Node *node : wire->nodes()) {
3         ...
4     }
  
```

5 }

Listing 2.21: Common loop throughout the Wiring Tool

is common throughout the methods of the Wiring Tools and is easily achievable with this implementation.

2.8 Wiring Elements

As introduced in [section 2.7](#), the Wiring Tool mainly works with the classes shown in [Figure 2.20](#).

Segments Segments are purely graphical elements that do not bear any information about the connection graph. They are automatically constructed whenever the connection between two nodes is created and they represent the line that is drawn between them. In order to follow the movements of their nodes, they use Qt’s event and slot system: whenever a node is moved, it emits a signal that the segment receives through its `updatePosition()` slot.

```

1 class WiringTool::Segment : public WiringTool::WireElement {
2     ...
3 public:
4     Segment(Node *node1, Node *node2, WiringTool &tool, Wire *wire = nullptr);
5
6 public slots:
7     void updatePosition();
8     ...
9
10 private:
11     ...
12     /* Inherited from QGraphicsObject */
13     void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *
        widget) override;
14     QRectF boundingRect() const override;
15     ...
16 private:
17     std::array<Node *, 2> m_nodes;
18 };

```

Listing 2.22: Main methods of the Segment class

To simplify this procedure, segments also rely on Qt’s local transform system. Basically, Qt allows graphic items to refer to coordinates through an internal transform of their own which is also inherited from parent graphics objects to child graphics objects¹. In this case, segments have a transform that allows them to consider their

¹Graphics object parenting is not to be confused with class inheritance: in this case, parenting refers to having a graphics object transform be dependent on another one.

nodes as always at $(-1, 0)$ and $(+1, 0)$, greatly simplifying the implementation. This transform is applied in the following method:

```

1 void Segment::transformToNormalizeNodes() {
2     resetTransform();
3     // Set position to midpoint between nodes
4     QPointF midpoint = (m_nodes[0]->pos() + m_nodes[1]->pos()) / 2;
5     setPos(midpoint);
6     // Rotate and scale
7     QLineF line = QLineF(m_nodes[0]->pos(), m_nodes[1]->pos());
8     setRotation(-line.angle());
9     qreal length = line.length();
10    // Don't set the scale to 0!
11    setScale(length / 2 + 1e-16);
12 }

```

Listing 2.23: Transform of a Segment

So that in the end they can implement methods such as Qt's `boundingRect()` (used for collision detection and for updating the scene) in a simplified manner:

```

1 QRectF Segment::boundingRect() const {
2     qreal height = 2 * Node::radius_max / scale();
3     return QRectF(-1, -height / 2, 2, height);
4 }

```

Listing 2.24: Bounding rect of a Segment

Nodes Nodes act both as graphical elements and as interconnection graph elements. They are graphically represented as dots whose size and color may change according to their status. As clearly shown across multiple pictures throughout [section 2.7](#), they become orange when selected and they also become larger when the mouse pointer hovers over them for clarity and better aesthetics. On top of graphical management, they also manage connections.

```

1 class WiringTool::Node : public WiringTool::WireElement {
2     ...
3 public:
4     Node(WiringTool &tool, Wire *wire = nullptr);
5     ...
6 private:
7     QHash<Node*, Segment*> m_connections;
8     ...
9 };

```

Listing 2.25: Constructor and main attribute of a Node

Their most important attribute is `m_connections` which is a hash map that maps any connected node to the segment that connects them. This serves not only to track which nodes are connected to which, but also through which segments; additionally segments may easily be updated to carry some custom information about the connection. As a small example, the list of connected nodes can be obtained with:

```

1 QList<Node *> Node::connectedNodes() const {
2     return m_connections.keys();
3 }

```

Listing 2.26: Retrieving the connections of a Node

When the connection between two nodes is requested, the caller (in this case a Wire) creates a segment and adds the nodes to eachother's map of connections both referring to the same segment.

```

1 Segment *Wire::connect(Node *node1, Node *node2) {
2     // Create a segment and notify the nodes
3     Segment *segment = new Segment(node1, node2, tool(), this);
4     segment->setZValue(Segment::layer);
5     node1->connect(node2, segment);
6     node2->connect(node1, segment);
7     m_segments << segment;
8     return segment;
9 }

```

Listing 2.27: A wire connecting two nodes

Wires Wires are containers that collect nodes and segments and are, for all intents and purposes, a representation of a connected, loopless, undirected simple graph. As such, they implement some graph-related methods such as `pathConnecting()` that returns the path between two nodes. This method is used both for user-based purposes such as path selections like in [Figure 2.15](#) and for internal critical sections such as the enforcement of the connected and loopless attributes.

```

1 QList<Node *> Wire::pathConnecting(Node *node1, Node *node2) const {
2     // If either node does not belong to this wire, no path can exist
3     if (node1->wire() != this || node2->wire() != this)
4         return QList<Node*>();
5     QSet<Node*> visited_nodes;
6     return recursivePathConnecting(node1, node2, visited_nodes);
7 }
8
9 QList<Node *> Wire::recursivePathConnecting(Node *node1, Node *node2, QSet<Node *>
10     &visited) const {
11     QList<Node*> path;
12     Node *current = node1;
13     Node *target = node2;
14     visited << node1;
15     for (Node *node: current->connectedNodes()) {
16         if (visited.contains(node))
17             continue;
18         else if (node == target) {
19             // Found target!
20             path << current << target;
21             return path;
22         } else {
23             // Recursively look for paths starting from this new node
24             QList<Node*> subpath = recursivePathConnecting(node, target, visited);
25             if (!subpath.empty()) {

```

```

25         // Found in recursive call
26         path << current << subpath;
27         return path;
28     }
29 }
30 }
31 return path;
32 }

```

Listing 2.28: Pathfinding in a Wire

2.9 Connectors

An additional small change applied to the `qcagraphicslib` are **Connectors**, the contact point between the wires and items in the scene. Connectors were added with the wires in mind, but they were implemented as an abstract concept that may be re-used according to other needs.

```

1  class Connector : public QGraphicsObject {
2      Q_OBJECT
3  public:
4      explicit Connector(QGraphicsItem *parent = nullptr);
5      virtual ~Connector();
6
7      void attach(QGraphicsItem *item);
8      void detach(QGraphicsItem *item);
9      const QSet<QGraphicsItem*> &attachedItems() const;
10
11 protected:
12     virtual void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
13
14 signals:
15     void clicked(Connector *connector, QGraphicsSceneMouseEvent *event);
16
17 protected:
18     QSet<QGraphicsItem*> m_attached_items;
19 };

```

Listing 2.29: Base Connector class

Even though it is not immediately apparent, the base class is abstract because it does not implement the required virtual methods `paint()` and `boundingRect()` from `QGraphicsObject` because different connectors may have different shapes for various reasons. By default a connector is useful because it emits a `clicked` signal when the mouse is pressed so that some tool (such as the Wiring Tool) can receive that signal and react. The event handling may include a call to the `attach()` method that links a graphics item to the connector for later retrieval. In the case of the Wiring Tool, the event is handled by creating a node on top of the clicked connector and attaching the node to it and it is very similar to [Listing 2.20](#).

The specific connector used by the Wiring Tool is a `PointConnector`, that appears like shown in [Figure 2.21](#):

```

1 class PointConnector : public Connector {
2     Q_OBJECT
3 public:
4     PointConnector(const QPointF &position, const QSizeF &size, QGraphicsItem *
5         parent);
6     ~PointConnector() override;
7
8     void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *
9         widget = nullptr) override;
10    QRectF boundingRect() const override;
11 private:
12    QSizeF m_size;
13 };

```

Listing 2.30: `PointConnector` class

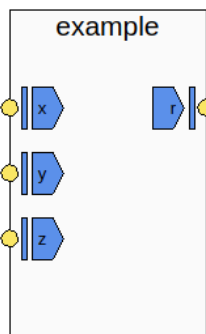


Figure 2.21: `PointConnectors` appear yellow and round in shape

Connectors are generated by a `QcaGraphicsWrapper` based on the indications of its wrapped item, which has a specifically created method called `connectionPoints()` that returns a list of points expressed in local coordinates where connectors should be placed. For instance, the generic Hybrid Wrapper item places its connectors on top of its pins.

```

1 QList<QPointF> HybridItemWrapperBase::connectionPoints() const {
2     QList<QPointF> points;
3     for (QcaPin *pin : m_pins) {
4         // Set the connection point on the tip of the pin
5         QPointF offset;
6         if (pin->direction() == QcaPin::INPUT) {
7             offset = QPointF(0.0, 0.5);
8         } else {
9             offset = QPointF(1.0, 0.5);
10        }
11        points << pin->pos() + offset;
12    }
13    return points;

```


14 }

Listing 2.31: Listing the connection points of an Hybrid Wrapper item

Connectors were also manually added for pins, which are a specific instance of a `QcaItem`, in order to be able to connect them to the items in the scene:

```
1 QList<QPointF> QcaPin::connectionPoints() const {
2     QList<QPointF> list;
3     // Reference point (normalized)
4     QPointF position;
5     if (direction() == INPUT) {
6         position = QPointF(1.0, 0.5);
7     } else {
8         position = QPointF(0.0, 0.5);
9     }
10    QTransform transform;
11    // Orientations must be as asserted below, otherwise this code needs to be
        updated!
12    static_assert(RIGHT == 0 && DOWN == 1 && LEFT == 2 && UP == 3,
13                  "Orientations are assumed to be in counter-clockwise order");
14    qreal rotation = -90.0 * orientation();
15    // Rotate depending on orientation
16    transform.rotate(rotation);
17    position = transform.map(position);
18    list << position;
19    return list;
20 }
```

Listing 2.32: Listing the connection points of an Hybrid Wrapper item

Since connectors only make sense in technologies where wiring is supported, they only appear under that condition.

2.10 Undo Commands

GUI programming poses a lot of challenges as the interactions with the user are greatly varied and edge cases or unpredicted actions are always behind the corner. Among these challenges stands the undo / redo paradigm, greatly increasing the quality of the user experience as much as the implementation complexity.

In Qt, undo functionalities are implemented through a `QUndoStack` and the `QUndoCommands`. An undo command represents an atomic action that can be done and undone repeatedly. To implement this behavior, the `QUndoCommand` class has two virtual methods that subclasses need to override: `undo()` and `redo()`. The method `redo()` is called both when the command has to be executed for the first time - more specifically, it is automatically called when a new command is pushed onto the stack - as well as when it has to be redone after being undone; `undo()` is instead called upon user request, such as when clicking **Ctrl+Z** or selecting this option from the **Edit** menu. The `QUndoStack` simply collects commands. When the user undoes their last action, `undo()` is called on the top element of the stack and the stack pointer is rolled back. At that point, the user can either redo the action or perform a new one. In the former case, the stack pointer advances and `redo()` is called; the latter case instead clears the stack from any undone actions (i.e. actions that are still in the stack, but beyond the current stack pointer because they have been rolled back) and pushes the new one, effectively overwriting the old actions that can no longer be redone. Actions that are removed from the stack, whether it is because new actions overwrite them or because the stack is manually cleared or for any reason in general, are always destroyed.

Undo commands have two additional, very useful properties: parenting and the ability to be merged. Parenting means that a certain command, typically an empty command, can collect a number of child-commands and undoing or redoing the parent means undoing and redoing the children (in order). An example of this has already appeared in [Listing 2.20](#): line 19 allocates the empty `parent` command and the other commands, such as the one in line 20, are allocated by specifying `parent` as their parent. It is then `parent` that is pushed on the stack in line 26. On the other hand, merging commands is useful to unify similar commands: it often makes more sense if some commands of the same type executed in succession are done and undone all at once. This happens, for instance, in a text editor where despite characters are typed one at a time, **Ctrl+Z** does not remove single letters but larger text chunks. As an additional advantage, this also saves space on the undo stack that is typically limited in size and would be easily filled by trivial commands.

The Wiring Tool defines the following commands:

- **CreateNodeCommand**: to create new nodes
- **DeleteNodeCommand**: to delete nodes
- **DissolveNodeCommand**: to dissolve nodes
- **ConnectNodesCommand**: to connect nodes
- **DisconnectNodesCommand**: to remove connections between nodes
- **ChangeSelectionCommand**: to update the selection
- **MoveSelectionCommand**: to move nodes
- **SimplifyGeometryCommand**: to simplify the geometry

Which are generally written in a flexible way so that they can be combined and adapted for more scenarios than their main purpose. There are two main challenges for this paradigm in Qt that can be referred to as “non-finality” and memory management. By non-finality it is meant that undo commands must, by their very definition, only apply changes that can be undone. As a small example, the undo in a text editor cannot simply delete and just lose the recently typed text from the document, because it must keep a local copy of it as well as the location where it was removed from in order to be able to re-insert it. It will only be allowed to lose these pieces of information when the command is outdated such as when the user, after undoing the command, types something else and effectively re-writes history, or when the command becomes too old and is simply removed from the stack, effectively becoming not undo-able anymore.

This introduces the second challenge, memory management, as Qt usually requires the allocation of its objects on the heap and thus it is necessary to delete them at the right time. The problem is that commands cannot blindly delete the things they allocate because sometimes items must survive the deletion of the commands that created them, some other times it is the opposite. To better understand this problem, let us introduce a simplified case of the **CreateNodeCommand**.

The real **CreateNodeCommand** is slightly complicated by the attributes desired by wires in the wiring tool: as already mentioned, wires are assumed to be connected, loopless, undirected simple graphs. The problem lies in the fact that creating a new node without connecting it to anything also implies the creation of a new wire and, at the same time, undoing that same action needs to remove the wire from the scene. So, for this example, let us only consider nodes and ignore wires and connections. In such a scenario, a naive implementation of the command would look like this:

```

1  class SimplifiedCreateNodeCommand : public QUndoCommand {
2  public:
3      SimplifiedCreateNodeCommand(QGraphicsScene *scene,
4                                  const QPointF &position,
5                                  QUndoCommand *parent) :
6          QUndoCommand(parent),
7          m_scene(scene),
8          m_node(nullptr),
9          m_position(position) {
10 }
11
12 void redo() override {
13     m_node = new Node(m_position);
14     m_scene->insertItem(m_node);
15 }
16
17 void undo() override {
18     m_scene->removeItem(m_node);
19     m_node->deleteLater();
20 }
21
22 private:
23     QGraphicsScene *m_scene;
24     Node *m_node;
25     QPointF m_position;
26 }

```

Listing 2.33: Naive implementation of a CreateNodeCommand

There are two problems with the implementation in [Listing 2.33](#):

1. The node's internal state is lost after the first undo.
2. Any external reference to the node is invalidated after every undo / redo.

Both problems may be solved through small changes within the same setup: for instance, the first problem may be faced by storing a local copy of the node before deleting it, but this is more of a workaround than a clean and complete solution. A better idea would be to ensure that the node is always actually the same node. For instance, this would be a better - but yet incomplete - version:

```

1  class SimplifiedCreateNodeCommand : public QUndoCommand {
2  public:
3      SimplifiedCreateNodeCommand(QGraphicsScene *scene,
4                                  const QPointF &position,
5                                  QUndoCommand *parent) :
6          QUndoCommand(parent)
7          m_node(new Node(position)) {
8      }
9
10 ~SimplifiedCreateNodeCommand() {
11     m_node->deleteLater();
12 }
13
14 void redo() override {

```

```

15     m_scene->insertItem(m_node);
16 }
17
18 void undo() override {
19     m_scene->removeItem(m_node);
20 }
21
22 private:
23     QGraphicsScene *m_scene;
24     Node *m_node;
25 }

```

Listing 2.34: Improvement on the naive implementation of a CreateNodeCommand

This is arguably a much cleaner implementation that solves both of the problems mentioned above. The new problem in this case is that the node no longer survives its command that may be deleted without actually intending it. This happens, for instance, when the stack is too full so the oldest command is removed from the stack (and deleted) to make room for the new ones: that will also remove the node from the scene. There are multiple, equally valid solutions to this problem, but the chosen solution has the advantage of automatically handling the deletion. A new class was implemented:

```

1 class Kamikaze {
2 public:
3     Kamikaze();
4
5     void takeOwnership(QObject *object);
6     void restoreOwnership(QObject *object);
7     bool owns(QObject *object) const;
8
9 private:
10     QObject m_deleter;
11     QHash<QObject*, QObject*> m_original_owners;
12 };

```

Listing 2.35: Class for automatic deletion of objects

This class takes advantages of the fact that objects in Qt may have parents that automatically delete them in their destructor. For this reason, by passing an object through `takeOwnership()`, the object's original parent is stored and `m_deleter`, an empty `QObject` created with this only purpose, is set as its new parent. At that point, it is no longer necessary to explicitly delete the object because `m_deleter`'s destructor will destroy it as from Qt's parenting rules. The final implementation for the node creator now becomes:

```

1 class SimplifiedCreateNodeCommand : public QUndoCommand, private Kamikaze {
2 public:
3     SimplifiedCreateNodeCommand(QGraphicsScene *scene,
4                                 const QPointF &position,
5                                 QUndoCommand *parent) :
6         QUndoCommand(parent)

```

```
7         m_node(new Node(position)) {
8     }
9
10    void redo() override {
11        m_scene->insertItem(m_node);
12        if (owns(m_node))
13            releaseOwnership(m_node);
14    }
15
16    void undo() override {
17        m_scene->removeItem(m_node);
18        if (!owns(m_node))
19            takeOwnership(m_node);
20    }
21
22 private:
23     QGraphicsScene *m_scene;
24     Node *m_node;
25 }
```

Listing 2.36: Final implementation of a simplified create node command

With this implementation, memory management is automatically handled by Qt, references are always valid and the solution is not polluted with local copies of the items or their properties.

2.11 Saving

One additional feature to complete the development of the Hybrid Plugin is integrating it in the saving and loading procedures. In general, Hybrid items do not need particular changes as the saving / loading of a component is implemented as a general feature that is inherited from the abstract class `QcaItem`. One addition that was instead necessary is the compilation of the loaded item that must happen before it is inserted in the scene. Wires, instead, require some bigger adaptations.

Design saving is implemented by `qcalib`'s `DrawingSaver` class. It exposes various overloaded methods called `saveQcaLayout()` used to save the design as a layout file (with a `.qll` file format) and a `saveQcaComponent()` to save the design as a component (with a `.qcc` file format). There is a net distinction between the two entities and their generation processes because while a design is “final”, meaning that all of its items have definitive positions, sizes and properties; a component is slightly more abstract: the actual positions of its items are thought of as local coordinates inside of the component that may be arbitrarily placed in the design, in a similar way sizes and properties cannot be hard-coded in a component because they would otherwise clash with the settings of other designs where the component is to be instantiated.

Designs are saved using the `QXmlStreamWriter` class, meaning their contents follow the XML format. The stream writer automatically handles the generation of the necessary tags and fields so that the drawing saver can only worry about *what* to save instead of *how*. [Listing 2.37](#) shows an example of a saved design from which the component “wire3” inserted in [Figure 2.3](#) was extracted.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--File describing the layout of a QCA circuit-->
3 <qcalayout>
4     <technologies>
5         <settings tech="iNML">
6             <property name="Height" value="90"/>
7             <property name="Thickness" value="10"/>
8             ...
9         </settings>
10    </technologies>
11    <components>
12        <item tech="iNML" name="Magnet"/>
13    </components>
14    <layout>
15        <pin tech="iNML" name="DOUT" direction="1" id="1" x="4" y="0" layer="0"/>
16        <pin tech="iNML" name="DIN" direction="0" id="2" x="0" y="0" layer="0"/>
17        <item comp="0" id="3" x="3" y="0" layer="0">
18            <property name="phase" value="0"/>
19        </item>
20        <item comp="0" id="4" x="2" y="0" layer="0">
21            <property name="phase" value="0"/>
```

```
22         </item>
23         <item comp="0" id="5" x="1" y="0" layer="0">
24             <property name="phase" value="0"/>
25         </item>
26     </layout>
27 </qcalayout>
```

Listing 2.37: Contents of the save file from [Figure 2.3](#)

This file shows that the design contains three magnets in line and two pins called DIN and DOUT. It is also possible to observe how a typical save file is structured and what its main sections are:

1. **Technologies:** contains information about the technology in the design, specifically its name and all of its settings;
2. **Components:** lists the name of the used items and the technology they belong to;
3. **Layout:** describes the instancing and positioning of the items in the scene.

The separation between the Components and Layout sections was devised so that the Components section can simply use an indexing pattern in order to avoid repeating the same information about a given component multiple times by having the field `comp` in the `item` tag refer to the entry number (counting from 0) of the item in the Components section.

The new information about wiring is placed in the `layout` section. The choice fell on implementing a new system of lists and hash map as a general solution for the saving of newly implemented objects in the layout instead of having to update the `saveQcaLayout()` method, which would otherwise need to be updated on every new addition to MagCAD. To work with this system, a new interface called **Exportable** can be inherited from in order to expose general saving and loading methods:

```
1 template <class StreamWriter, class StreamReader>
2 class Exportable {
3 public:
4     Exportable() = default;
5     virtual ~Exportable() = default;
6
7     virtual StreamWriter &exportItem(StreamWriter &writer) const = 0;
8     virtual StreamReader &importItem(StreamReader &reader) = 0;
9 };
```

Listing 2.38: Exportable class

The drawing saver now also uses some additional type definitions:

```
1 class DrawingSaver {
2     ...
3 public:
```



```

4     using LayoutExportable = Exportable<QXmlStreamWriter, QXmlStreamReader>;
5     using LayoutCollection = QList<LayoutExportable*>;
6     using LayoutSections = QHash<QString, LayoutCollection>;
7     ...
8 };

```

Listing 2.39: Type definitions from `DrawingSaver`

Thanks to these new types, additions to MagCAD such as wiring can simply define a custom entry in the `LayoutSections` map that the program passes to `saveQcaLayout()` when saving, preventing the loss of control of the number of parameters of that method. On top of this, this approach abstracts the data types used in the design from those needed to save it as circular dependencies could arise: it was the case in an early development stage of the Wiring Tool, implemented in the `qcagraphicslib` which, in turn, uses the `qcalib` that also defines the `DrawingSaver` class - to put it more clearly, if the `DrawingSaver` directly used the Wiring Elements from [section 2.8](#), `qcalib` and `qcagraphicslib` would be cross-dependent. When exporting, the `DrawingSaver` can just iterate through the layout sections, write the section name and call the `exportItem()` method of the passed items.

Combining this approach with the advantages of organizing the code through `QcaSceneTools` makes the saving procedure very simple:

```

1 DrawingSaver::save_state_t DrawingScene::save(QFile *file) {
2     ...
3     DrawingSaver saver;
4     DrawingSaver::LayoutSections sections;
5     if (techSettings->getProperty("wiring").toBool())
6         sections.insert("wiring", m_wiring_tool->collectExportableItems());
7     return saver.saveQcaLayout(globalOptions, this->techSettings, this->qcas(),
8         sections, file);
9 }

```

Listing 2.40: Saving of a scene

Where the `WiringTool::collectExportableItems()` method exploits the fact that Wires also inherit from the `Exportable` class. Their `exportItem()` method uses the same indexing trick of MagCAD's save files to represent nodes:

```

1 QXmlStreamWriter &Wire::exportItem(QXmlStreamWriter &writer) const {
2     writer.writeStartElement("wire");
3     // Assign indices to nodes
4     std::size_t index = 0;
5     QHash<const Node*, std::size_t> node_indices;
6     for (Node *node : m_nodes)
7         node_indices.insert(node, index++);
8     for (Node *node : m_nodes)
9         node->exportItem(writer, node_indices);
10    writer.writeEndElement();
11    return writer;
12 }

```

Listing 2.41: Wire's `exportItem()` method

It is possible to observe how nodes are indexed with a first sweep and then exported with a second. Indexing is required to simplify the description of connections. A node can then save its index, its position and connections:

```

1 QXmlStreamWriter &Node::exportItem(QXmlStreamWriter &writer, const QHash<const Node
  *, std::size_t> indices) const {
2     writer.writeStartElement("node");
3     writer.writeAttribute("index", QString::number(indices[this]));
4     writer.writeAttribute("x", QString::number(pos().x()));
5     writer.writeAttribute("y", QString::number(pos().y()));
6     // Process connections
7     QString connections;
8     for (WiringTool::Node *connection : connectedNodes())
9         connections += QString::number(indices[connection]) + " ";
10    // Remove last space
11    connections.remove(connections.size() - 1, 1);
12    writer.writeAttribute("connections", connections);
13    writer.writeEndElement();
14    return writer;
15 }
```

Listing 2.42: Node's `exportItem()` method

A full example of a save file for a design containing items and wires in the Hybrid technology is the following:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--File describing the layout of a QCA circuit-->
3 <qcalayout>
4     <technologies>
5         <settings tech="Hybrid">
6             <property name="wiring" value="true"/>
7             <property name="Layoutheight" value="7"/>
8             <property name="Layoutwidth" value="16"/>
9         </settings>
10    </technologies>
11    <components>
12        <item tech="Hybrid" name="Component"/>
13    </components>
14    <layout>
15        <item comp="0" id="1" x="14" y="4" layer="0">
16            <property name="Path" value="inml/custom/wire3.qcc"/>
17        </item>
18        <item comp="0" id="2" x="9" y="4" layer="0">
19            <property name="Path" value="inml/custom/wire3.qcc"/>
20        </item>
21        <item comp="0" id="3" x="5" y="4" layer="0">
22            <property name="Path" value="inml/custom/wire3.qcc"/>
23        </item>
24    <wiring>
25        <wire>
26            <node index="0" x="350" y="275" connections="1"/>
27            <node index="1" x="450" y="275" connections="0"/>
28        </wire>
29        <wire>
30            <node index="0" x="550" y="275" connections="1"/>
31            <node index="1" x="700" y="275" connections="0"/>
```

```

32         </wire>
33     </wiring>
34 </layout>
35 </qcalayout>

```

Listing 2.43: Contents of the save file from [Figure 2.3](#)

2.12 Loading

The loading procedure instead requires a bit more attention. It is executed by the `DrawingReader` class which, unlike the `DrawingSaver`, is not part of the `qcalib` since it is a class of the top level MagCAD itself.

Also in this case, most of the operations are automatically executed thanks to the inheritance patterns in MagCAD. The only difference with other items is that Hybrid items need to have their `compile()` method called before they can be inserted in the design. For this reason, they are intercepted while loading in order to compile the content of the file they wrap:

```

1 DrawingReader::parse_state_t DrawingReader::parse(QFile *file) {
2     ...
3     auto *hybrid_item = static_cast<HybridItemWrapperBase*>(currentItem);
4     if (hybrid_item) {
5         try {
6             hybrid_item->compile();
7         } catch (const QCAException &exception) {
8             QMessageBox mbox(
9                 QMessageBox::Critical,
10                "MagCAD Error",
11                QString("Failed to parse file '%1': %2").arg(file->fileName(),
12                                                            exception.what()),
13                QMessageBox::Ok);
14            mbox.exec();
15            return PARSE_FAIL;
16        }
17    }
18    ...
19 }

```

Listing 2.44: Interception of Hybrid items during loading

For the same reasons encountered when saving, Wiring data cannot use the Wire Elements from [section 2.8](#) because that would create a circular dependency between `qcagraphicslib` and MagCAD, so the same trick of using hash maps with intermediate data was employed.

```

1 class DrawingReader {
2 public:
3     ...
4     using WiringNodeProperties = QHash<QString, QString>;
5     using WiringNodeCollection = QList<WiringNodeProperties>;
6     using WiringData = QList<WiringNodeCollection>;

```

```

7
8     DrawingReader(const QList<QcaTechnologyInterface *> & technologies,
9                  ComponentsWidget *componentWidget);
10
11     parse_state_t parse(QFile *file);
12
13     DrawingSettings *parsedSettings() { return m_settings; }
14
15     QList<QcaItem *> parsedItems() const { return m_itemList; }
16     QList<QcaPin *> parsedPins() const { return m_pins; }
17     const WiringData &parsedWiringData() const { return m_wiring_data; }
18 ...
19 };

```

Listing 2.45: The DrawingReader class

The data coming from the wiring section of a save file is stored in the `WiringData` map that explicitly contains the parameters as written in the file itself, so that the Wiring Tool can handle this data on its own. Basically, the `WiringNodeProperties` are a partially digested structure containing the properties from XML file and they correspond to the properties of a single node, the `WiringNodeCollection` groups properties from multiple nodes and will form a wire, finally `WiringData` collects a list of to-be wires.

The structure of the `parse()` method is somewhat similar to the one used in [section 2.5](#), as it is based on a simple state machine. In order to add support for wiring, a new state was added where all the wiring section is parsed and accumulated in a local member called `m_wiring_data` whose reference can be accessed with `parsedWiringData()`.

Once the parsing procedure is successfully completed, it is possible to call `WiringTool::insertWiringElements()` that accepts a reference to a variable of type `WiringData`. It internally processes the data and generates the wires for the design. This method is based upon the undo commands from [section 2.10](#) to re-use the already existing infrastructure. Once again, the same problem from the saving procedure arises: connections are stored as indices, but nodes must be created before being indexed. For this reason, this method creates the nodes first, then maps their index to them, and finally creates the connections.

```

1 void WiringTool::insertWiringElements(const DrawingReader::WiringData &wiring_data)
2 {
3     ...
4     // Create nodes and wires as QUndoCommands
5     QUndoCommand *parent = new QUndoCommand;
6     ChangeSelectionCommand::clearSelection(*this, parent);
7     for (const DrawingReader::WiringNodeCollection &node_collection : wiring_data)
8     {
9         QHash<QString, Node*> node_index_map;
10        // Initialize nodes
11        for (const DrawingReader::WiringNodeProperties &node_properties :
12             node_collection) {

```

```

10     qreal x = node_properties.value("x").toDouble();
11     qreal y = node_properties.value("y").toDouble();
12     QPointF position(QPointF(x, y));
13     auto *create_node_command = new CreateNodeCommand(*this, position,
14         parent);
15     Node *node = create_node_command->node();
16     // Check if node was created on top of a connector
17     for (QGraphicsItem *item : scene().items(position)) {
18         auto *connector = dynamic_cast<Connector*>(item);
19         if (connector) {
20             node->bindToConnector(connector);
21         }
22     }
23     ChangeSelectionCommand::selectNodes(*this, {node}, true, parent);
24     node_index_map.insert(node_properties.value("index"), node);
25 }
26 // Form connections
27 for (const DrawingReader::WiringNodeProperties &node_properties :
28     node_collection) {
29     QString current_node_index = node_properties.value("index");
30     QStringList connection_index_list = node_properties.value("connections"
31         ).split(" ", Qt::SkipEmptyParts);
32     for (const QString &connected_node_index : connection_index_list) {
33         Node *node1 = node_index_map[current_node_index];
34         Node *node2 = node_index_map[connected_node_index];
35         /* Check if connection hasn't already been formed in the reverse
36            order (a.k.a. current_node was already
37            * met as a connected_node before)
38            */
39         if (!node2->isConnectedTo(node1)) {
40             new ConnectNodesCommand(*this, node1, node2, parent);
41         }
42     }
43 }
44 }
45 ChangeSelectionCommand::clearSelection(*this, parent);
46 ...
47 }

```

Listing 2.46: Inserting loaded wires

Notice also that since there is no explicit information about nodes being placed on connectors in the save file, that information is inferred based on the position of the node in lines 16 - 21: this method goes through any items present in the scene below the node - that, by the node creation rules, should only be connectors -, checks if they are actually connectors and then records the association.

Chapter 3

Flipp

3.1 ModelSim FLI

ModelSim is an established digital circuit simulator widely used around the world as a tool for the verification of digital designs. Among countless advanced functionalities, ModelSim has a feature that is particularly interesting to the purpose of this thesis: its Foreign Language Interface (FLI). This feature allows users to define special modules whose behaviors are implemented in a foreign language, namely C.

The FLI opens a new world for the description of hardware, its representative purpose being verification: it is common practice to compare the behavior of VHDL modules to reference models, also called “golden models”, which are pieces of software written in C, C++, Python or whatever the designer prefers in order to test for the correctness of the design. In a simple implementation, this can be done by supplying the same inputs to a VHDL simulation and to an execution of the golden model, to then dump the outputs of both in two files to finally compare them. With ModelSim FLI it is unbelievably easy to compare a VHDL model to a C program in a testbench when the testbench itself is written in C.

When describing a VHDL component, it is possible to set the special **foreign** attribute to an architecture to enable this functionality. The attribute specifies two things: the name of a C function and the name of a compiled shared library that contains that function. This already reveals how the FLI works: ModelSim comes with a special C / C++ header in every installation that acts as an API that enables C or C++ programs to interact with the FLI. These programs must be compiled into a shared library that will be loaded by ModelSim when, while compiling, it meets a **foreign** definition. Throughout this chapter, the main example will be of an inverter for simplicity. With that as a reference, this is how a foreign architecture is declared in VHDL:

```
1 library ieee;
```

```

2 use ieee.std_logic_1164.all;
3
4 entity Inverter is
5     port (
6         x: in  std_logic;
7         q: out std_logic
8     );
9 end entity Inverter;
10
11 architecture ForeignArch of Inverter is
12     attribute foreign of ForeignArch : architecture is "inverter_init inverter.so";
13 begin
14 end architecture ForeignArch;

```

Listing 3.1: A foreign architecture

Where line 12 is where the attribute is declared and `inverter.so` is a compiled shared library in which the function `inverter_init()` is implemented. It is also possible to set an optional textual argument to be passed to the function separated by a semicolon from the rest of the attribute declaration, as in:

```

1 attribute foreign of ForeignArch : architecture is "inverter_init inverter.so; An
   optional textual argument :);

```

Listing 3.2: Optional arguments in a foreign declaration

To understand how ModelSim FLI works, it is best to do it step by step through an example in [section 3.2](#).

3.2 FLI Example: an Inverter in C

This section uses [Listing 3.1](#), an example of a foreign declaration for a simple inverter, as the reference VHDL code which is assumed to be saved in a file named `inverter.vhd`. The only other step before getting into the C code is to setup a Makefile to simplify the development of this module:

```

1 CFLAGS := -Werror -Wall -Wpedantic -std=c11 -fPIC
2 LDFLAGS := -export-dynamic -shared -Bsymbolic
3
4 MTI_PATH = /eda/mentor/2020-21/RHELx86/QUESTA-CORE-PRIME_2020.4/questasim/include
5
6 inverter.so: inverter.o
7     $(LD) ${LDFLAGS} -o $@ $^
8
9 inverter.o: inverter.c
10    $(CC) -c ${CFLAGS} -I${MTI_PATH} -o $@ $^
11
12 clean:
13    $(RM) *.o *.so

```

Listing 3.3: Makefile

Line 4 defines the path to the so called “MTI” headers that expose the needed types and functions. These headers are located in an `include/` directory in ModelSim installation path.

It is now time to move on to the C implementation. First, the creation of `inverter.c` and its `inverter_init()` function:

```

1  #include "mti.h"
2
3  int inverter_init(mtiRegionIdT region,
4                  char *arg,
5                  mtiInterfaceListT *generics,
6                  mtiInterfaceListT *ports) {
7      // Inverter initialization
8      return 0;
9  }
```

Listing 3.4: Declaration of the `inverter_init()` function

This function is automatically called by ModelSim after the elaboration phase, that is a phase between the compilation of the design and the start of the simulation. It is very important to be aware that this phase may be passed through multiple times, for instance when the simulation is restarted. ModelSim does not reset upon restarting the simulation, so any reset of the custom module state must be performed by hand. The only way to catch this scenario is through specific MTI functions that reveal whether the `inverter_init()` function is being called for the first time or not. To not over-complicate this example, this is ignored and it is assumed that the simulation is never restarted.

The parameters of this function are:

1. **region**: an identifier for the region that represents the VHDL entity in the simulation;
2. **arg**: the optional argument that can be passed as in [Listing 3.2](#);
3. **generics**: a null terminated list containing information about the entity generics from [Listing 3.1](#);
4. **ports**: a null terminated list containing information about the ports from [Listing 3.1](#).

The description of the `region` parameter reveals the core organization of MTI: all the references to simulation objects are managed through their IDs and can be interacted with through specific functions. For instance, it is possible to print the name of the region with:

```

1  int inverter_init(mtiRegionIdT region,
2                  char *arg,
3                  mtiInterfaceListT *generics,
```



```

4         mtiInterfaceListT *ports) {
5     mti_PrintFormatted("Region name: %s\n", mti_GetRegionName(region));
6     return 0;
7 }

```

Listing 3.5: Printing the region name

No special steps are required at this point: the file can be compiled and simulated like any other VHDL file. It is the simulator that internally recognizes the need of loading an external library and goes ahead to find it. After compiling the VHDL file with `vcom inverter.vhd` and preparing the simulation with `vsim work.inverter`¹, ModelSim outputs:

```

1 vcom inverter.vhd
2 # QuestaSim-64 vcom 2020.4 Compiler 2020.10 Oct 13 2020
3 # Start time: 11:49:35 on Nov 21,2021
4 # vcom -reportprogress 300 inverter.vhd
5 # -- Loading package STANDARD
6 # -- Loading package TEXTIO
7 # -- Loading package std_logic_1164
8 # -- Compiling entity Inverter
9 # -- Compiling architecture ForeignArch of Inverter
10 # End time: 11:49:35 on Nov 21,2021, Elapsed time: 0:00:00
11 # Errors: 0, Warnings: 0
12 vsim work.inverter
13 # vsim work.inverter
14 # Start time: 11:49:39 on Nov 21,2021
15 # Loading std.standard
16 # Loading std.textio(body)
17 # Loading ieee.std_logic_1164(body)
18 # Loading work.inverter(foreignarch)#1
19 # Loading ./inverter.so
20 # Region name: inverter

```

Listing 3.6: ModelSim transcript

Where the last line shows the expected output.

In order to fully implement the desired model, it is necessary to introduce the main actors in the FLI and how to interact with them. In a very close resemblance to a pure VHDL description, the MTI is based upon the following concepts:

- **Region:** groups elements such as signals, processes and even other sub-regions and roughly corresponds to a VHDL entity;
- **Signal:** a direct handle to a VHDL signal in the simulation;
- **Variable:** a direct handle to a VHDL variable in the simulation. The variable may be a generic, a constant or a process variable;

¹The foreign model does not need to be a top level design: all the foreign models are initialized at this point regardless of where they are in the design hierarchy. This was the case simply because there are no other files at this point.

- **Type:** represents the type of a signal or a variable in the simulation;
- **Driver:** element used to write values on a signal;
- **Process:** a direct handle to a VHDL process, this can be the channel through which the C code can implement its model.

In general, the MTI offers a strong yet simple interface with a high degree of control. All of the described elements can both be retrieved from the simulation or manually created and inserted in the design as if they were written in the VHDL description. For instance, the ports of an entity are signals and, in order to be interacted with, must be retrieved from the simulation. On the other hand, C can create and inject a process to implement a custom behavior. On top of this, it is also possible to break the standard setup of a simulation and force values on signals without processes or drivers through specific MTI functions.

Entries created in the initializer function cannot live on the stack as they will be deallocated when the function returns, so let us define a `struct` and allocate it in the heap:

```

1  #include "mti.h"
2
3  typedef struct {
4      // Ports
5      mtiSignalIdT x;
6      mtiSignalIdT q;
7      // Drivers
8      mtiDriverIdT dq;
9      // Process
10     mtiProcessIdT process;
11 } Inverter;
12
13 void inverter_callback(void *arg) {
14     // Callback for the inverter
15 }
16
17 int inverter_init(mtiRegionIdT region,
18                 char *arg,
19                 mtiInterfaceListT *generics,
20                 mtiInterfaceListT *ports) {
21     // Allocate inverter
22     Inverter *inv = (Inverter*) mti_Malloc(sizeof(Inverter));
23     // Initialize ports
24     inv->x = mti_FindPort(ports, "x");
25     inv->q = mti_FindPort(ports, "q");
26     // Create process and set its sensitivity list
27     inv->process = mti_CreateProcess("proc_inverter", inverter_callback, inv);
28     mti_Sensitize(inv->process, inv->x, MTI_EVENT);
29     // Create driver and set its owner
30     inv->dq = mti_CreateDriver(inv->q);
31     mti_SetDriverOwner(inv->dq, inv->process);
32     return 0;

```

33 }

Listing 3.7: Allocating the Inverter

The `Inverter` struct stores the handle IDs for later usage. After the sensitivity list setup on line 28, the registered callback is executed upon every event on signal `x`.

In order to write the callback, it is necessary to understand how values are read from and written to signals depending on the type of the signal. Types have a “kind” that can be:

- **Scalar** for integer values;
- **Real** for real, floating point values;
- **Enum** for enumerations;
- **Array** for arrays;
- **Time** for time.

The original entity from [Listing 3.1](#) declares `x` and `q` as `std_logic` which is an enumeration. The simplest approach to interact with this type is to mirror the same enumeration in C, being careful that the order of declaration is important:

```

1  typedef enum {
2      STD_LOGIC_U,
3      STD_LOGIC_X,
4      STD_LOGIC_0,
5      STD_LOGIC_1,
6      STD_LOGIC_Z,
7      STD_LOGIC_W,
8      STD_LOGIC_L,
9      STD_LOGIC_H,
10     STD_LOGIC_D
11 } StdLogic;
```

Listing 3.8: C enum mirroring the `std_logic` type in VHDL

The final callback can use this enum to read the value of `x` and set `q` accordingly:

```

1  void inverter_callback(void *arg) {
2      Inverter *inv = (Inverter*) arg;
3      // Read input and prepare output value
4      StdLogic x_value, q_value;
5      x_value = mti_GetSignalValue(inv->x);
6      if (x_value == STD_LOGIC_0) {
7          q_value = STD_LOGIC_1;
8      } else if (x_value == STD_LOGIC_1) {
9          q_value = STD_LOGIC_0;
10     } else {
11         q_value = STD_LOGIC_U;
12     }
13     // Schedule signal assignment with 1 delta delay
14     mti_ScheduleDriver(inv->q, (long) q_value, 1, MTI_INERTIAL);
```

15 }

Listing 3.9: Allocating the Inverter

It is now possible to write a simple testbench to visualize the results:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Testbench is
5  end entity Testbench;
6
7  architecture Behavior of Testbench is
8      component Inverter is
9          port (
10             x: in  std_logic;
11             q: out std_logic
12          );
13      end component;
14
15      signal x, q : std_logic;
16  begin
17      comp_inverter: Inverter port map (x, q);
18
19      proc_testbench: process
20      begin
21          x <= '0';
22          wait for 10 ns;
23          x <= '1';
24          wait for 30 ns;
25          x <= '0';
26          wait;
27      end process proc_testbench;
28  end architecture Behavior;

```

Listing 3.10: Testbench for the Inverter in C

Launching a simulation is no different from any standard digital simulation: it is sufficient to launch the usual commands and everything, including the dynamic linking of the custom functions, is handled automatically. In this case:

```

1  vcom inverter.vhd
2  vcom testbench.vhd
3  vsim work.testbench
4
5  add wave *
6
7  run 60 ns

```

Listing 3.11: FLI Simulation commands

Note that these commands, as standard practice, are saved in a `simulate.do` file that may be launched with:

```

1  vsim -do simulate.do

```

Listing 3.12: Invoking a .do script

Figure 3.1 shows the output waveform with the expected behavior, including the small delay between the input and output.

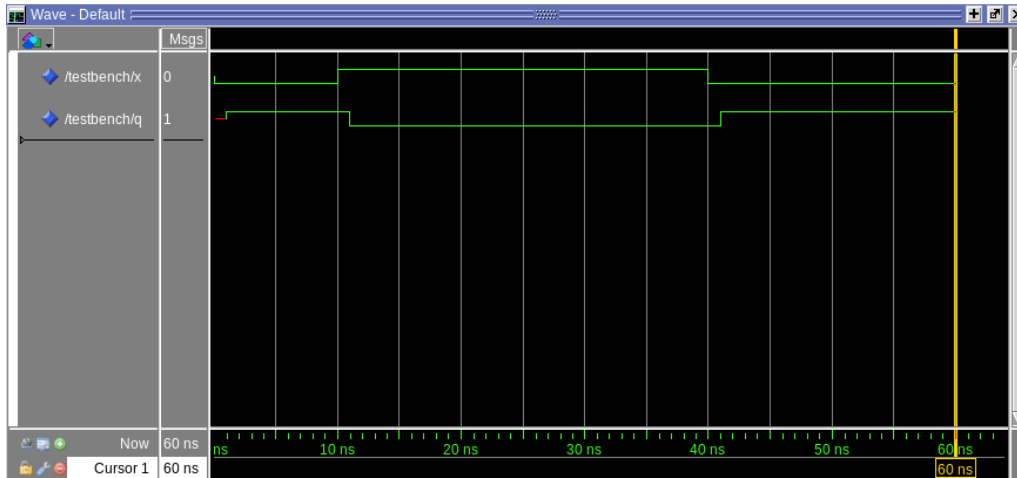


Figure 3.1: Waveform of the Inverter in C

3.3 Flipp

Flipp is a C++ library built on top of the standard C ModelSim FLI developed to offer a higher level access to the functionalities in a more modern approach, easily adaptable to the ToPoliNano framework.

Flipp is completely compatible to the standard FLI in the sense that it does not interfere with the original API in any way: a custom component can use calls from both libraries without problems. This is especially thanks to the fact that ModelSim MTI headers were developed to be compatible with C++ in the first place, so there are no additional compilation or compatibility issues.

Since flipp is, at its core, an external collection of wrappers to the standard FLI calls, ModelSim has no way of knowing about its existence if it is not somehow included in the final shared object indicated in the foreign attribute definition. For this reason, there are two possible approaches to use it: users can either directly compile flipp's sources together with their own functions describing a custom model to produce a single shared object, or they can compile flipp as a separate shared library and link it to their own models. The second approach is more helpful to streamline the final shared object(s) since flipp will be shared among them. For this reason, flipp was in fact developed with that approach in mind.

As discovered in [section 3.2](#), the standard FLI revolves around the concept of IDs that represent the various elements in the simulation and allow interactions with them. In the same way, flipp revolves around its `Handle` classes.

```

1  template<class IDType_>
2  class Handle {
3  public:
4      using IDType = IDType_;
5      ...
6  public:
7      explicit Handle(IDType id) : m_id(id) {};
8      Handle(const Handle<IDType> &other) = default;
9      virtual ~Handle() = default;
10
11      [[nodiscard]]
12      IDType ID() const {
13          return m_id;
14      }
15
16      ...
17
18      virtual void print(std::ostream &ostream, PrintMode mode) const {
19          UNUSED(mode);
20          ostream << "Handle(id=" << ID() << ")";
21      }
22
23      bool operator==(const Handle<IDType> &other) const {
24          return m_id == other.m_id;
25      }
26
27      bool operator!=(const Handle<IDType> &other) const {
28          return m_id != other.m_id;
29      }
30      ...
31  private:
32      IDType m_id;
33  };

```

Listing 3.13: Snippet of the base Handle class in flipp

Handles are wrappers to simulation elements with IDs that offer some convenient methods for the interactions with them. The base class is derived into classes that reflect some of the different ID types in the FLI:

- Region
- Signal
- Type
- Process
- Driver

Other handles (such as variables) are not currently supported. As an example, the `Region` specialization adds the following methods:

```

1  namespace flipp {
2  class Region : public Handle<mtiRegionIdT>{

```

```

3 public:
4     explicit Region(IDType id);
5     Region(const Region &other);
6     ~Region() override;
7
8     static Region create(const std::string &name, const Region &parent);
9     [[nodiscard]] static std::optional<Region> find(const std::string &name);
10
11     [[nodiscard]] std::string name() const;
12     [[nodiscard]] std::string primaryName() const;
13     [[nodiscard]] std::string sourceFileName() const;
14
15     [[nodiscard]] std::optional<Region> parent() const;
16     [[nodiscard]] std::vector<Region> children() const;
17     [[nodiscard]] static std::vector<Region> topLevelRegions();
18
19     [[nodiscard]] std::vector<Signal> signals() const;
20
21     void print(std::ostream &ostream, PrintMode mode) const override;
22     void printTree(std::ostream &ostream, unsigned depth = 8) const;
23
24 private:
25     void printTree(std::ostream &ostream, unsigned depth, std::string &&prefix)
26         const;
27 };

```

Listing 3.14: Region Handle

Notice that it is not a coincidence that this class has no attributes: in fact, no Handles ever have any attributes except for their ID. This is done to prevent data incoherence: Handles are manually created through FLI IDs (as the constructor in [Listing 3.14](#) shows) and while their lifetime is determined by the scope where they are created, the object they handle is not as it lives in the simulation. This means that any information associated to a handle that cannot be written inside the object it handles would have a limited lifetime. On top of this, it is perfectly legal to create multiple handles to the same simulation object, but changes in one would not reflect to others and this would lead to confusion. There could be countless workarounds to this problem, but the most rigorous and generic one is to not use attributes at all and let the user allocate custom data through hash maps or similar structures if needed.

As a representative example of how flipp methods wrap the standard MTI, this is `Region::create()`²:

```

1 Region Region::create(const std::string &name, const Region &parent) {
2     return Region(mti_CreateRegion(parent.ID(), const_cast<char*>(name.c_str())));

```

²The `const_cast()` is needed because the MTI headers lack the `const` declarations in its functions. This practice is a common workaround throughout flipp, but it is used only where necessary and safe.

3 }

Listing 3.15: Example of a wrapping function to create a region

Another interesting feature about flipp is its output management which is brought to the standard of C++ through a global output stream called `sout`, inspired by the classic `cout` but referred to the simulation. Also using this as a first example for flipp, this is how [Listing 3.5](#) would look like with flipp:

```
1 #include "flipp/Handles/Region.h"
2 #include "flipp/I0/I0.h"
3
4 extern "C" int inverter_init(mtiRegionIdT region,
5                             char *arg,
6                             mtiInterfaceListT *generics,
7                             mtiInterfaceListT *ports) {
8     flipp::Region top_region(region);
9     flipp::sout << "Region name: " << top_region << std::endl;
10    return 0;
11 }
```

Listing 3.16: Example of printing a Handle with flipp

Whose output is:

```
1 # Region name: Region("inverter", id=0x7fefe80401d8)
```

Listing 3.17: Output for [Listing 3.16](#)

All of flipp's Handles offer other useful features that allow a back-and-forth interaction with the simulation environment. For instance, it is possible to navigate through the tree of regions in the design through the `topLevelRegions()` and `children()` methods. It can also be interesting to print the full region tree with `printTree()` to explore the design for debugging purposes.

In this case it can also be helpful to develop the same example from before with the new library.

3.4 Flipp Example: an Inverter in C++

In reality, flipp offers multiple approaches to obtain the same result. It is possible to either manually use the Handles described in the previous section, thus obtaining a result which is quite similar to the original C implementation, or to use additional classes that will be introduced in [section 3.5](#).

Starting with a simple implementation, it is possible to follow the same structure from C:

```
1 #include "flipp/I0/I0.h"
2 #include "flipp/Handles/Region.h"
3 #include "flipp/Handles/Signal.h"
4 #include "flipp/Handles/Process.h"
```



```

5  #include "flipp/Handles/Driver.h"
6  #include "flipp/Handles/Value.h"
7  #include "flipp/Handles/Type.h"
8
9  using namespace flipp;
10
11 struct Inverter {
12     Signal x;
13     Signal q;
14     Driver dq;
15     Process process;
16 };
17
18 void inverter_callback(Inverter &inv) {
19     Long x_value = inv.x.value().interpretedData<Type::Kind::Enum>();
20     Long q_value;
21     using namespace flipp::standard_types;
22     if (x_value == StdLogicType::STD_LOGIC_0) {
23         q_value = StdLogicType::STD_LOGIC_1;
24     } else if (x_value == StdLogicType::STD_LOGIC_1) {
25         q_value = StdLogicType::STD_LOGIC_0;
26     } else {
27         q_value = StdLogicType::STD_LOGIC_U;
28     }
29     inv.dq.schedule(q_value, 1);
30 }
31
32 extern "C" int inverter_init(mtiRegionIdT region,
33                             char *arg,
34                             mtiInterfaceListT *generics,
35                             mtiInterfaceListT *ports) {
36     // Allocate inverter
37     Inverter *inv = static_cast<Inverter*>(mti_Malloc(sizeof(Inverter)));
38     // Initialize signals
39     new(&inv->x) Signal(Signal::fromPortsList("x", ports));
40     new(&inv->q) Signal(Signal::fromPortsList("q", ports));
41     // Create process and set its sensitivity list
42     new(&inv->process) Process(Process::create("proc_inverter", [inv]() {
43         inverter_callback(*inv);
44     }));
45     inv->process.sensitize(inv->x);
46     // Create driver and set its owner
47     new(&inv->dq) Driver(Driver::create(inv->q, inv->process));
48     return 0;
49 }

```

Listing 3.18: Implementation of an Inverter in C++

Aside from the C++ adaptations and namespace management, this example tries to keep the structure as close as possible to the reference one. Notice that, first of all, in order to comply with MTT's suggested allocation technique (which does not appear to be required, but was used nonetheless), the slightly obscure *placement new* operator is needed to construct in-place the fields of the struct. Alternatively, it could be possible to overload the `new` operator.

Handles have some convenience static methods used for their initialization, such as `Signal::fromPortsList()` and `Process::create()`. Their methods also closely resemble the functions they wrap, just compare `Process::sensitize()` to the original `mti_Sensitize()`.

The most notable difference is the retrieval of `x`'s value in line 19, whose complexity is arguably higher than the standard MTI method. The reason for this higher complexity is not particularly clear with this example, but could be the cause of mistakes in different scenarios. The problem is that the standard `mti_GetSignalValue()` *always* returns a `long` when called in C, even when the corresponding signal is of type `real`, `time` or `array` in the simulation. When the signal is an `enum` or `scalar`, the value can be used as-is just like it happened in the example. In the other cases, the returned value has to be interpreted as a `void*` that points to a location in memory that contains data of the right type. It is left to the user to distinguish between the two scenarios, to correctly cast to a pointer and to also correctly cast to the correct data type. In order to offer a safer approach, flipp also implements an additional Handler called `Value` which simply wraps these return values. In order to read its data, it is either possible to use `Value::rawData()` that returns the same, unhandled `long` from the MTI call, or to use `Value::interpretedData<>()` which requires the user to insert the type kind in order to internally handle the conversions. According to the C++ standards, flipp defines the following type-functions:

```

1  template <Type::Kind kind> struct ktypes{};
2
3  template <> struct ktypes<Type::Kind::Enum>      { typedef Int    type; };
4  template <> struct ktypes<Type::Kind::Scalar>    { typedef Int    type; };
5  template <> struct ktypes<Type::Kind::Physical>  { typedef Int    type; };
6  template <> struct ktypes<Type::Kind::Real>     { typedef double type; };
7  template <> struct ktypes<Type::Kind::Time>     { typedef Time   type; };

```

Listing 3.19: Flipp's type-functions

By also recycling the previous testbench, it is possible to check that the output waveforms are once again as expected in [Figure 3.2](#). On a side note, observe how easy it is to switch models: no VHDL file was touched for this second example; in practice it is sufficient to simply swap the to-be-loaded shared object in the filesystem for a new one to change the component's behavior.

3.5 Model-Behavior Approach

An alternative approach to the same description problem can be solved with the classes supporting a *Model-Behavior approach*. Notice that by trying to abstract the steps happening inside the `inverter_init()` function from [Listing 3.18](#), it is possible to find a structure that holds in a general case:

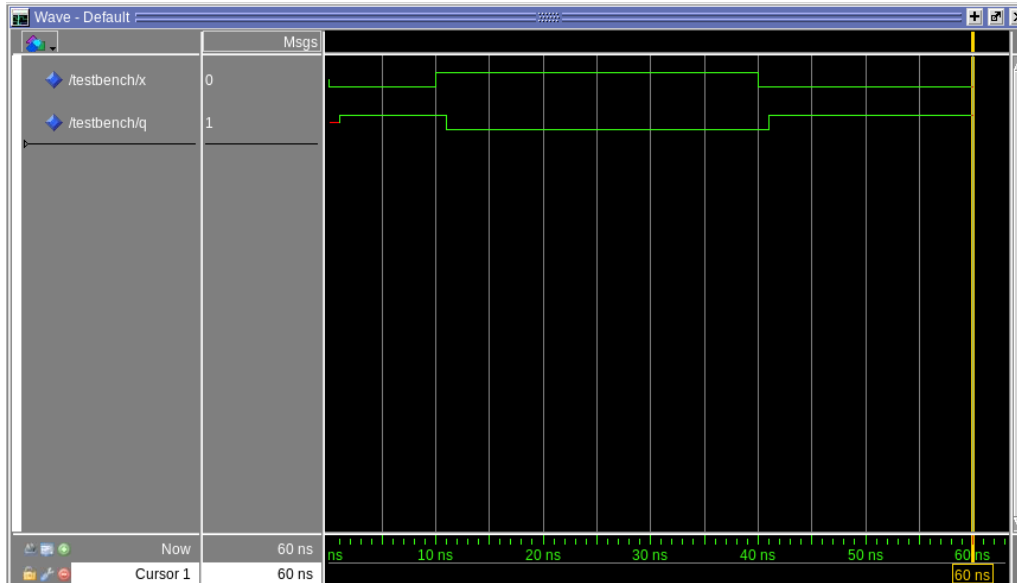


Figure 3.2: Waveform of the Inverter in C++

1. Read ports from the port list
2. Create processes
3. Create drivers

When developing such models, these parts should be handled automatically and the designer should only worry about the model behavior. In this approach, a **Model** represents a circuit with all of its physical elements in the simulation - basically, it is the equivalent of an entity in VHDL -, while a **Behavior** is a collection of **Actions** that interact with the model, with actions being higher level wrappers to processes. This closely resembles the entity-architecture paradigm of hardware description languages.

A model is created by passing it a name, a region and a vector containing all of its ports.

```

1  #include "flipp/IO/IO.h"
2  #include "flipp/Handles/Type.h"
3  #include "flipp/Models/Model.hpp"
4
5  extern "C" int inverter_init(mtiRegionIdT region,
6                               char *arg,
7                               mtiInterfaceListT *generics,
8                               mtiInterfaceListT *ports) {
9
10     Region reg(region);
11     // Create model
12     Model *inv = static_cast<Model*>(mti_Malloc(sizeof(Model)));
13     new(inv) Model(reg.name(),
14                   reg,
15                   {
16                       Signal::fromPortsList("x", ports),
17                   }
18     );
19 }
```

```

16         Signal::fromPortsList("q", ports)
17     });
18 }

```

Listing 3.20: Allocation of a Model

Once the model has been initialized, it is time to set up its behavior. To do this, the user create an arbitrary number of Actions to finally call `generateHandles()` that automatically creates the process and driver handles to manage the action.

```

1  using namespace flipp;
2
3  class InverterAction : public flipp::Action {
4  public:
5      InverterAction (Model &model,
6                      std::set<std::string> triggers,
7                      std::set<std::string> outputs) :
8          Action(model, std::move(triggers), std::move(outputs)) {
9      }
10
11     void run() override {
12         ...
13     }
14 };
15
16 extern "C" int inverter_init(mtiRegionIdT region,
17                             char *arg,
18                             mtiInterfaceListT *generics,
19                             mtiInterfaceListT *ports) {
20     Region reg(region);
21     // Create model
22     Model *inv = static_cast<Model*>(mti_Malloc(sizeof(Model)));
23     new(inv) Model(reg.name(),
24                   reg,
25                   {
26                       Signal::fromPortsList("x", ports),
27                       Signal::fromPortsList("q", ports)
28                   });
29     // Add actions and auto-generate their associated handles
30     inv->behavior().addAction<InverterAction>({"x"}, {"q"});
31     inv->behavior().generateHandles();
32 }

```

Listing 3.21: Allocation of a Model

Each Action is a subclass of the abstract `flipp::Action` that must implement at least its `run()` method, which is equivalent to the callbacks from the previous example. In this case, indeed:

```

1  class InverterAction : public flipp::Action {
2      ...
3      void run() override {
4          Int x_value = std::get<Int>(readInput("x"));
5          Int q_value;
6          using namespace flipp::standard_types;
7          if (x_value == StdLogicType::STD_LOGIC_0) {
8              q_value = StdLogicType::STD_LOGIC_1;

```

```

9      } else if (x_value == StdLogicType::STD_LOGIC_1) {
10          q_value = StdLogicType::STD_LOGIC_0;
11      } else {
12          q_value = StdLogicType::STD_LOGIC_U;
13      }
14      setOutput("q", (Long) q_value, 1);
15  }
16  };

```

Listing 3.22: `run()` method for the `InverterAction`

Since this is a higher level description, it gets rid of the problem of interpreting the data when reading a signal with a wrapper function that returns an `std::variant` in which the signal value is written accordingly to its type. For the rest, the structure of this function is similar to the original callback. Running the simulation once again by using the same scripts, testbench and VHDL files from before produces the expected output shown in [Figure 3.3](#).

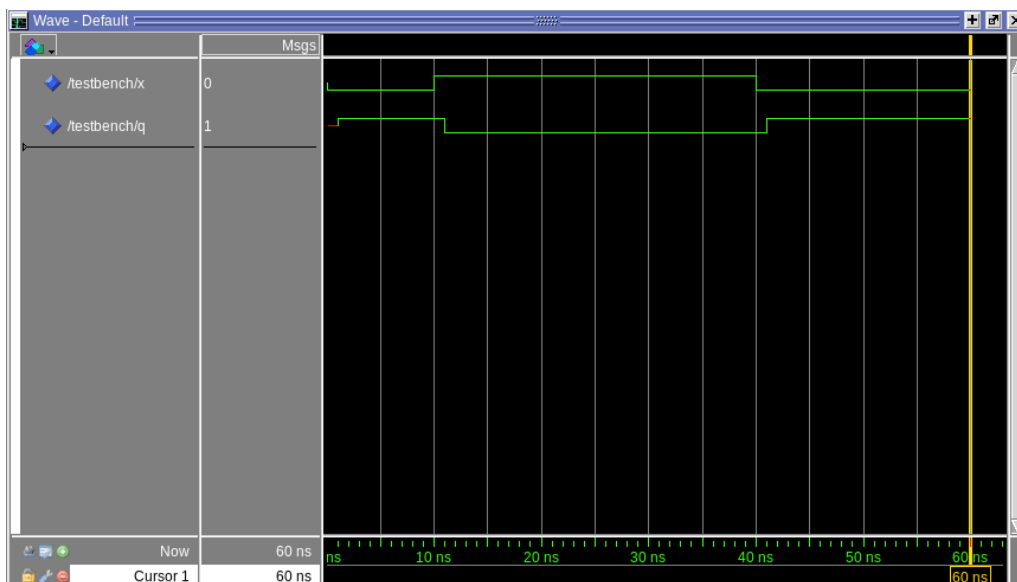


Figure 3.3: Waveform of the Inverter with the Model-Behavior approach

This alternative approach takes away the complexity of manually managing all the handles by packaging and automating all of the cumbersome steps of creating processes, drivers and linking them and the low level interactions required by the standard approach. Of course, this simple scenario produces similar results in complexity and size for both approaches, but this would not be the case for most larger designs.

On top of this, the Model-Behavior approach also offers additional “quality of life” features. For instance, an Action has a `trigger()` method that can be freely called to run it just as if its corresponding process had been awakened from inside the simulation, allowing more freedom in the interactions between actions. As another

example, `Actions` also have the `preRun()` and `postRun()` virtual methods that are empty by default but can be overridden, both called before and after `run()` when the `Action` is triggered.

```
1 class Action {
2 public:
3     ...
4     void trigger();
5     ...
6 private:
7     virtual void preRun();
8     virtual void postRun();
9 }
```

Listing 3.23: Some “quality of life” methods

These are mostly intended for debugging purposes, but they can be customized arbitrarily and may be used to initialize simulation variables or similar purposes.

Chapter 4

Simulation

4.1 Introduction

This chapter presents an example of a complete simulation of a dummy Hybrid Circuit using the system presented by this thesis. Because of the early stage of development of the complete system, it is going to be necessary to take some manual steps in the process, but as the environment evolves, these steps are going to be completely automated and handled by ToPoliNano itself.

A general introduction of the simulated circuit is presented in [section 4.2](#); the following sections go over and explain each component, the technology it is based on and how its description works. This introduction will instead describe the simulation environment and its setup.

All of the simulation files are assumed to be stored under a directory called `simulation/` anywhere in the file system. There are three subdirectories:

- `simulation/src/` contains the source files for “physical” components;
- `simulation/tb/` contains the source files for the top level testbench and all the components that have no real physical meaning;
- `simulation/sim/` is the folder where the simulation is run, so it contains all the files generated by the simulator, the `work/` directory and the custom automation scripts.

The simulation runs on a licensed version¹ Questa ADMS version 2020.4. In general, none of the features described by this thesis require an analog-capable simulator: most of the testing and development was actually done under QuestaSim,

¹This is required to be able to use the FLI.

which does not handle analog simulations. Since this example includes an analog component described in VerilogA - the analog extension of the standard, purely digital Verilog -, this small change was required. In any case, the procedures are always the same and the commands are almost identical in both cases, so this is not a problem.

4.2 The Hybrid Circuit

The circuit presented in this example does not have any real-world counterpart or meaning, but it is a relatively simple instance where multiple components collaborate in a single simulation despite not only being based on different technologies, but also differing in description style and provenience. This is a very important aspect since real simulations will often use parts coming from different sources: in this case, some are automatically generated by MagCAD, others are manually written by the thesis author, others come from research.

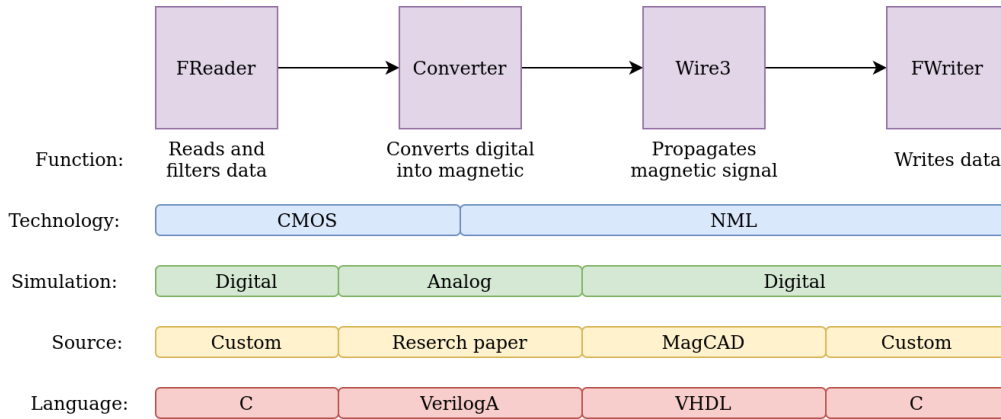


Figure 4.1: Scheme of the Simulation

Figure 4.1 depicts a small scheme where the simulated components, a brief description and their characteristics are listed. Functionally speaking, the simulated circuit will read data from an input file through a component that is assumed to be digital, filter it and convert it into magnetic signals that propagate until they are written in an output file. The components used by this circuit are quite varied and each will be described in a small dedicated section in this chapter. At this point it can be interesting to discuss the multiple levels of diversity shown in Figure 4.1 that this simulation embeds.

Starting from the most physical layer of the simulation, there are two technologies at play: CMOS and NML - more specifically, iNML. The iNML is a technology where signals are no longer voltages controlled by MOSFETs, but they are magnetic fields held in nanoscopic magnets. It is the interaction of adjacent magnets that

determines the propagation and evaluation of signals. A single component, the “Converter”, lives in between the two domains and is used as a bridge between them. It is realized with an MTJ, a device whose magnetization is controlled by an input current and vice versa. Two purely behavioral components, the File Reader and File Writer, respectively read input data from a file to feed it to the circuit and write the received data into another file. Having an abstract description, they technically do not belong to any real technology, but they are used as placeholders for a CMOS circuit generating input data and another iNML circuit digesting the data.

Moving on to the simulation type, it is possible to observe how most components are placed in the digital domain, while the converter lives in the analog domain. That is mostly related to the desired descriptions for the various components: the file readers and writers are mostly testbench utilities, so it makes sense they have a behavioral description. The iNML wire also has a Behavioral description and is thus simulated digitally. The converter, instead, is based on the solution of an analog system and uses related tools to react to its inputs and control its output.

Finally, the sources and languages of these descriptions differ too: the file reader and writer are developed in C through the FLI introduced in [chapter 3](#); the iNML wire was automatically generated by MagCAD in VHDL; the MTJ-based converter description comes from a publication and is described through a Verilog wrapper of a VerilogA netlist.

4.3 iNML Wire

The wire is a component built with iNML and exported from MagCAD. As briefly introduced, iNML technology represents its states through the magnetization of its nano-magnets.

The simplest possible component is a nano-magnet wire, where an input signal is propagated by a domino-like chain effect that inverts the magnetostatic field of every next element in line [14]. In order to obtain the most control over the change of state in these devices - and also out of necessity for integrating them on a chip -, these nano magnets are typically excited through a multi-phase clocking system [12]. This is important to understand because MagCAD follows this principle and its auto-generated VHDL would be unclear otherwise. The objective is to create a short wire composed by three magnets in line, just like the one in [Figure 4.2](#). Observe from the figure how this will invert the input signal in the last magnet: this will be important to keep in mind when observing the simulation results that will manifest this input data inversion, which is otherwise simply propagated linearly

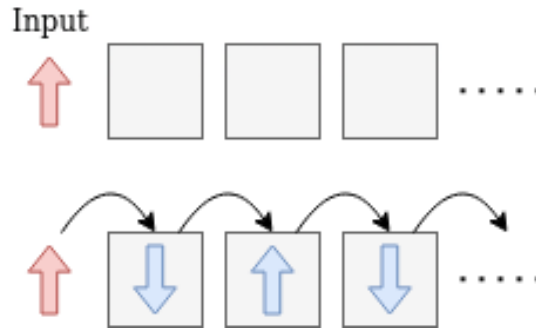


Figure 4.2: Propagation of a signal in iNML technology

input-to-output.

To create this component, MagCAD is opened and “iNML” is selected from the Drawing Settings window already met in [Figure 2.2](#) with the standard settings. From the Item Picker, the “Magnet” element is selected and the option “Use auto-



Figure 4.3: Selection of the iNML item to place

matic phase” is enabled - even though this example will use a single phase - as in [Figure 4.3](#). Three items are then placed in the design space as in [Figure 4.4](#). Since the objective is to generate a VHDL description, input and output pins are placed by clicking on the “Pin” tool and setting their properties ([Figure 4.5](#)) under the Item Picker. The final design is the one shown in [Figure 4.6](#).

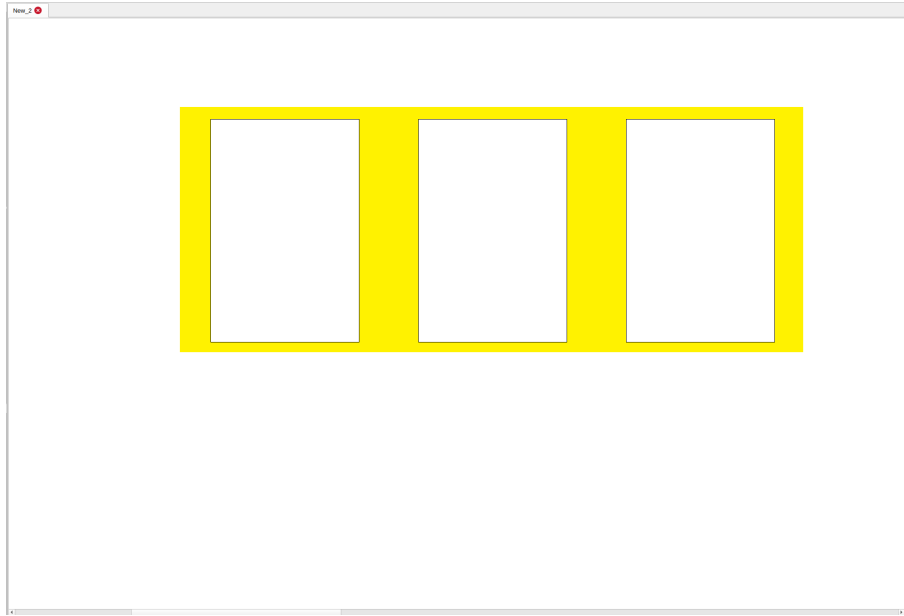


Figure 4.4: Placing iNML magnets

Figure 4.5: Setting the input and output pins

Finally, clicking on **File > Export Component** opens a dialog to save the design, which is going to be saved under the name of “wire3”, and then the Export Dialog from [Figure 4.7](#) where, after making sure that the “Export VHDL” option is checked, the component can be exported. Another window to setup some technological parameters opens up, but this example uses the default values so no additional customization is needed. The generated VHDL files are now saved in the default MagCAD output directory, which in the case of a standard configuration corresponds to the path `~/MagCADFiles/VHDL/wire3/`. This directory contains the VHDL description of the generated component together with the generation log (`wire3.log`) and other VHDL library files that contain the necessary definitions for the simulation. Notice that there are also some auto-generated testbenches that may be used directly in the case this was the top-level design, or simply as an inspiration for the custom testbench in [section 4.7](#). This is where ToPoliNano would read its input files, but in this case these output files are manually copied into the source directory `simulation/src/`.

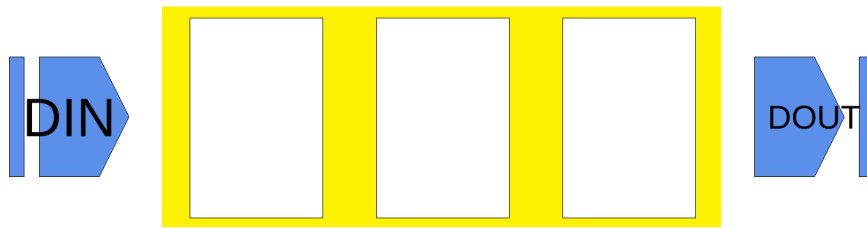


Figure 4.6: Final iNML design

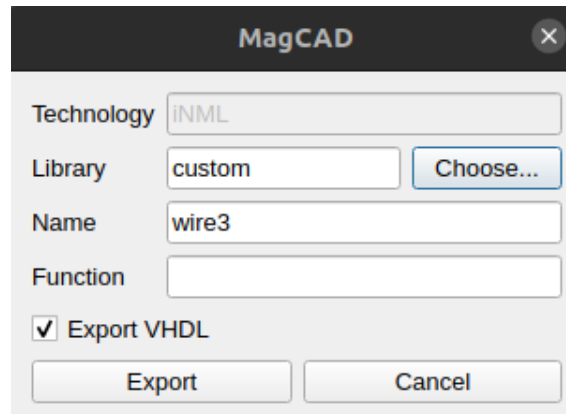


Figure 4.7: Component export dialog

The generated entity looks like this:

```

1 entity iNML_custom_wire3 is
2 port(
3   DOUT: out std_logic;
4   DIN: in std_logic;
5   CLK: in std_logic_vector(0 to 2));
6 end iNML_custom_wire3;
```

Listing 4.1: Wire3 entity

and the additional files upon which this is based that are also copied to the simulation source directory are `definition_inml.vhd` and `library_inml.vhd`.

4.4 MTJ Converter

An MTJ is a device that connects its magnetic properties to the current that flows through it. It is formed by two ferromagnetic layers called *fixed layer* and *free layer* separated by a thin insulating layer as depicted in Figure 4.8. The resistance of the insulating layer, through which current may flow thanks to quantum tunneling, depends on the magnetization of the ferromagnetic layers and determines the high impedance / low impedance state of the device [1]. It is possible to either program the device by sending a current through it, consequently magnetizing its free layer,

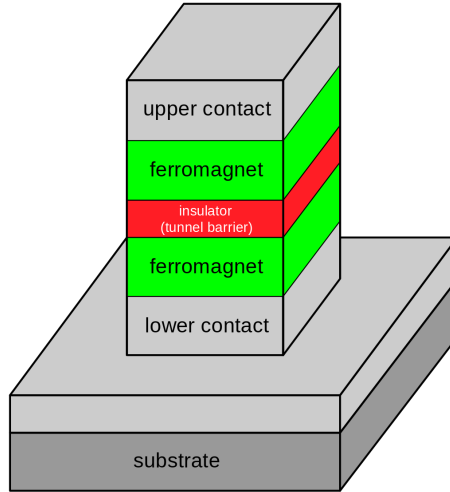


Figure 4.8: Structure of an MTJ, courtesy of [Wikipedia](#)

or to read its magnetic state through the resistance of the insulating layer.

As a small explanatory example, [Figure 4.9](#) shows the activation of an MTJ in an ad-hoc simulation ran with SPICE. A 100 μA current is sent through the device,

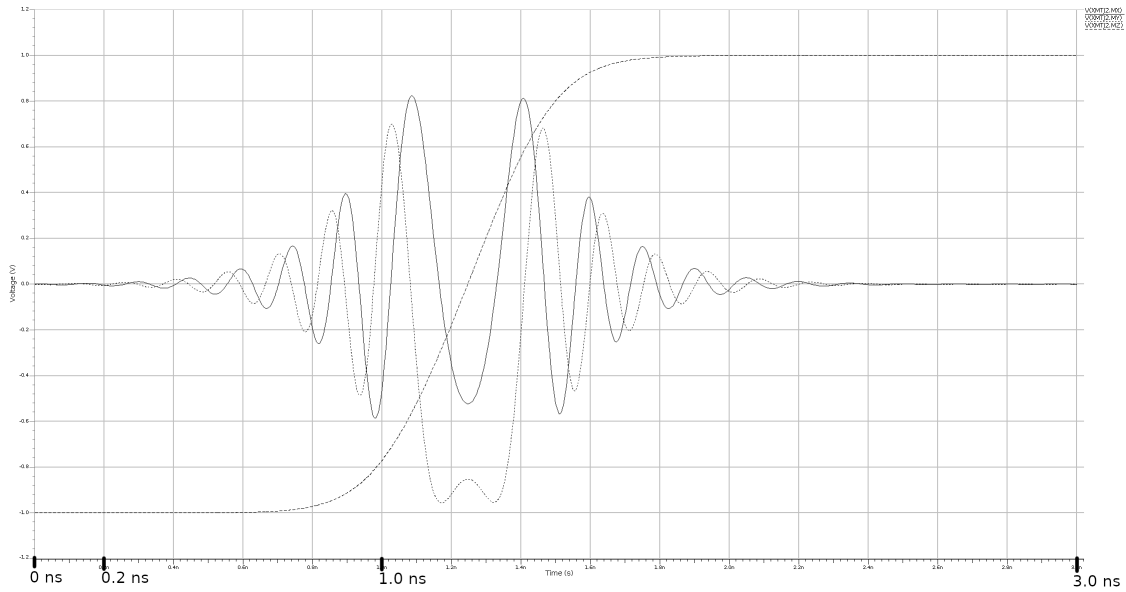


Figure 4.9: Activation of the MTJ

modifying the magnetic field inside of the free layer of which [Figure 4.9](#) shows a normalized version. The most significant component of the field is the Z component, which is the one that determines the resistance of the insulating layer (and thus the state of the MTJ), but it is possible to observe how the field moves through all coordinates. In this case, the current is flipping the direction of the normalized field that goes from -1 to 1. Note that, taking [Figure 4.8](#) as a reference, the Z axis is horizontal, and can be assumed to be going from left to right.

An MTJ is used in the simulation to convert the input digital signal coming from the file reader into a magnetic signal that can propagate through the iNML wire described in [section 4.3](#). The model is written in VerilogA and was developed by the Nanoelectronics Research Laboratory at Purdue University [9]. This model solves the characteristic equations of an MTJ, but since its interface is completely analog and a bit complex, an additional wrapper written in VerilogA but with a purely digital interface was also added.

The wrapper is called D2M (Digital To Magnetic) and its module has two ports: **din**, the input digital signal, and **mout**, the output magnetic signal.

```

1 module D2M (
2     input din,
3     output reg mout
4 );

```

Listing 4.2: D2M Wrapper

Notice that the output magnetic signal is technically of the same type of any digital signal in Verilog, but this is only because the VHDL generated in [section 4.3](#) is a behavioral description of the magnetic wire and uses the 0 / 1 state of a digital signal to represent the orientation of the magnetic field.

After setting up the parameters with the default values from the model from Purdue University, an instance of the MTJ is created and connected to local signals.

```

1 // ... Parameters setup ...
2
3 electrical fl, pl, hx, hy, hz, th, ph, MX, MY, MZ;
4
5 PMAMTJ #(
6     .Ku2      (Ku),
7     .W        (W),
8     .P_L      (P_L),
9     .P_R      (P_R),
10    .L         (L),
11    .Tm        (Tm),
12    .Ms        (Ms),
13    .alpha     (alpha),
14    .Lambda_L  (Lambda_L),
15    .Lambda_R  (Lambda_R)
16 )
17 mtj(fl, pl, hx, hy, hz, th, ph, MX, MY, MZ);

```

Listing 4.3: MTJ instance

A VerilogA description is characterized by an **analog** block where its analog properties are defined. Quantities in the analog section may have different types, but this example only focuses on **electrical** quantities. It is not possible to use an **electrical** by its name only, but two access functions are needed: **V(x)** and **I(x)**, representing the voltage or current of that given net. The analog section only uses its specific **<+>** (contribution) operator, which, as the name suggests, marks a

contribution to the selected quantity. Multiple contributions to the same signal are obviously allowed. There is one additional rule to be considered: variables and signals that live in the analog or digital domain can be read but *cannot* be written in the other domain, where “digital domain” refers to anything that is described outside of an `analog` block.

In its analog section, D2M’s module first sets the initial conditions of the MTJ using the special `initial_step` event as suggested by the developers of its model.

```

1  analog begin
2      @(initial_step) begin
3          V(th) <+ 1.75e-3;
4          V(ph) <+ 0.0;
5      end
6
7      V(pl) <+ 0;
8
9      @(din);
10     I(f1) <+ transition(din == 0, 0.5e-9) * (+‘ISW);
11     I(f1) <+ transition(din == 1, 0.5e-9) * (-‘ISW);
12 end

```

Listing 4.4: D2M analog section

It then fixes the voltage of `p1` (the *pinned layer*) to 0 V and controls the current flowing through `f1` (the *free layer*) based on the input `din`, which is the port that is going to be connected to the file reader in the final design. Lines 9 to 11 are used to translate the digital input into the transitions of an analog quantity: line 9 is an empty event handler that is used to ensure that the analog solver is awoken on events of `din`, which is digital and would otherwise be ignored; lines 10 and 11 use a small trick based on the `transition()` filter, a VerilogA call used to prevent hard discontinuities on the signals to help and stabilize the analog solver, to set the contributions on the current. The `transition()` filter takes as its first argument an expression that, over time, resolves into a piecewise constant function; the second argument is the transition time needed to travel from one value to the next. Using a boolean check like the ones in lines 10 and 11 is valid because they correspond to a piecewise constant functions whose values are either 1 or 0, depending on the value of `din` on every instant. Notice that the two calls use complementary boolean checks, so even though `I(f1)` has two contributions, only one is active at a time. Multiplying by \pm ‘ISW has the effect of setting the contribution to \pm ‘ISW when its expression evaluates to 1. So in this case, the current transitions to +‘ISW when `din` is 0 with a rise time of 0.5 ns, and it goes to -‘ISW in 0.5 ns when `din` is 1. ‘ISW is a macro to set the driving current to 500 μ A.

Finally, outside of the `analog` block, the module checks the values of the magnetic field and sets the output accordingly.

```

1 always @(cross(V(MZ))) begin
2     if (V(MZ) > 0)
3         mout = 1'b1;
4     else
5         mout = 1'b0;
6 end

```

Listing 4.5: D2M output control

This block is triggered when MZ - the normalized representation of the magnetic field of the free layer along the MTJ's Z axis - crosses 0. When the field is positive, the output is set to 1, otherwise it is set to 0.

Both of the descriptions used by this converter are saved in the sources directory under the names `simulation/src/pmamptj.va` and `simulation/src/D2M.va`.

4.5 File Reader

The File Reader is a component responsible to read input data from a text file. The proposed environment under flipp has been discussed and tested in [chapter 3](#), but since it is a very young alternative to an already established and in-house supported feature of ModelSim, this section uses the standard FLI in C for a more solid example that may be more of interest to the reader.

The input file contains one 1 or 0 per line that represent a bit stream. The role of this reader is to read the input bits on every rising edge of the clock and filter the input data with a sliding window before outputting it. The window has a width of 3 and acts as a majority voter, and it is filled with 0s at the beginning.

The first step is to initialize the VHDL file with the foreign architecture declaration as in [section 3.2](#). It is a standard declaration where the foreign attribute specifies the name of the initialization function and the name of the shared library that contains it. The shared library is called `io.so` because the File Writer is compiled there as well.

```

1 entity FReader is
2     port (
3         clk:    in std_logic;
4         dout:   out std_logic
5     );
6 end entity FReader;
7
8 architecture ForeignArch of FReader is
9     attribute foreign of ForeignArch : architecture is "freader_init io.so";
10 begin
11 end architecture ForeignArch;

```

Listing 4.6: FReader VHDL description

Moving on to the C description, the structure containing the component data is the following:

```

1 #define WSIZE 3
2 _Static_assert(WSIZE > 1, "Window size should be at least 1");
3 _Static_assert(WSIZE % 2 == 1, "Window size should be odd");
4
5 typedef struct {
6     // MTI Data
7     mtiSignalIdT clk;
8     mtiSignalIdT dout;
9     mtiDriverIdT ddout;
10    mtiProcessIdT process;
11    // Custom data
12    FILE *f;
13    bool file_ok;
14    int window[WSIZE];
15 } FReader;

```

Listing 4.7: FReader's struct in C

On top of the standard FLI IDs for the `clk` and `dout` signals, `dout`'s driver and its `process`, this time there is some additional data. The reader keeps a pointer to a `FILE` because it has to read one line per update of its callback, together with a flag that signals whether the file is okay to prevent trying to read from it after the file ends. Finally, the `window` is a simple array of `WSIZE` integers.

The initialization function allocates the described struct and initializes both the simulator-specific data and the custom data.

```

1 int freader_init(mtiRegionIdT region,
2                 char *arg,
3                 mtiInterfaceListT *generics,
4                 mtiInterfaceListT *ports) {
5     // Allocate reader
6     FReader *freader = (FReader*) mti_Malloc(sizeof(FReader));
7     if (freader == NULL) {
8         mti_PrintFormatted("Could not allocate FReader\n");
9         return -1;
10    }
11
12    // Initialize simulation quantities
13    freader->clk = mti_FindPort(ports, "clk");
14    freader->dout = mti_FindPort(ports, "dout");
15
16    freader->process = mti_CreateProcess("proc_freader", freader_callback, freader)
17    ;
18    freader->ddout = mti_CreateDriver(freader->dout);
19    mti_SetDriverOwner(freader->ddout, freader->process);
20    mti_Sensitize(freader->process, freader->clk, MTI_EVENT);
21
22    // Initialize custom data
23    freader->f = fopen("data.txt", "r");
24    if (freader->f == NULL){
25        mti_PrintFormatted("Could not open file!\n");
26        return -1;
27    }
28 }

```

```

26     }
27     freader->file_ok = true;
28     for (unsigned i = 0; i < WSIZE; ++i) {
29         freader->window[i] = 0;
30     }
31     return 0;
32 }

```

Listing 4.8: FReader's init function

After initializing all the simulator-specific fields just like in [section 3.2](#), it tries to open the input file (and reports an error in case of failure) and initializes the window to 0.

The callback associated to the process in line 16 is called `freader_callback` and its implementation is the following:

```

1 void freader_callback(void *arg) {
2     FReader *freader = (FReader*) arg;
3     // Check if file is ok
4     if (!freader->file_ok)
5         return;
6     // Check if rising edge of clock
7     if (mti_GetSignalValue(freader->clk) != STD_LOGIC_1)
8         return;
9     // Read data
10    int num;
11    if (fscanf(freader->f, "%d", &num) == EOF) {
12        fclose(freader->f);
13        freader->file_ok = false;
14        return;
15    }
16    for (unsigned i = WSIZE - 1; i > 0; --i) {
17        freader->window[i] = freader->window[i-1];
18    }
19    freader->window[0] = num;
20    // Vote and set output
21    long dout = 0;
22    for (unsigned i = 0; i < WSIZE; ++i)
23        dout += freader->window[i];
24    dout = (dout > WSIZE / 2 ? STD_LOGIC_1 : STD_LOGIC_0);
25    // Schedule driver
26    mti_ScheduleDriver(freader->ddout, dout, 0, MTI_INTERNAL);
27 }

```

Listing 4.9: FReader's callback

Before making useless calls to the FLI, it checks if the file is still open and returns immediately if it is not. The circuit has to work synchronously with the clock: the equivalent in VHDL would have a process sensible to the clock and would check for the rising edge with `if clk'event and clk = '1' then`. The equivalent in this C implementation is the sensitization to events on `clk` and the condition on line 7 to check if the clock is currently set to 1. After that, it inserts the new value in the window by reading it from the file - and returning if the file is over - and shifting it

into the window. Finally, it implements a simple majority voter and schedules the output driver accordingly.

For ease of use, the following Makefile was also written:

```

1 CFLAGS := -std=c11 -Werror -Wall -Wpedantic -fPIC -I/eda/mentor/2020-21/RHELx86/
   QUESTA-CORE-PRIME_2020.4/questasim/include/
2 LDFLAGS := -export-dynamic -shared -Bsymbolic -L.
3
4 io.so: freader.o fwriter.o
5     $(LD) ${LDFLAGS} $^ -o $@
6
7 freader.o: ../tb/freader.c
8     $(CC) ${CFLAGS} -c $^
9
10 fwriter.o: ../tb/fwriter.c
11     $(CC) ${CFLAGS} -c $^
12
13 clean:
14     rm -f *.o io.so

```

Listing 4.10: Makefile for the FLI components

Notice that it also includes the file writer.

4.6 File Writer

The File Writer is very similar to the Reader: it is also a synchronous component, except that it receives data from the simulation and dumps it into an output file.

```

1 entity FWriter is
2     port (
3         clk: in std_logic;
4         din: in std_logic
5     );
6 end entity FWriter;
7
8 architecture ForeignArch of FWriter is
9     attribute foreign of ForeignArch : architecture is "fwriter_init io.so";
10 begin
11 end architecture ForeignArch;

```

Listing 4.11: FWriter VHDL description

The entity is almost identical to the reader's.

```

1 typedef struct {
2     // MTI Data
3     mtiSignalIdT clk;
4     mtiSignalIdT din;
5     mtiProcessIdT process;
6     // Custom data
7     FILE *f;
8 } FWriter;

```

Listing 4.12: FWriter's struct in C

As depicted from the struct, this component is even simpler because it has no need for any custom data outside of the FILE pointer and also has no output ports. For this reason, the initialization function:

```

1  int fwriter_init(mtiRegionIdT region,
2                  char *arg,
3                  mtiInterfaceListT *generics,
4                  mtiInterfaceListT *ports) {
5      // Allocate writer
6      FWriter *fwriter = (FWriter*) mti_Malloc(sizeof(FWriter));
7      if (fwriter == NULL) {
8          mti_PrintFormatted("Could not allocate FWriter\n");
9          return -1;
10     }
11
12     // Initialize simulation quantities
13     fwriter->clk = mti_FindPort(ports, "clk");
14     fwriter->din = mti_FindPort(ports, "din");
15
16     fwriter->process = mti_CreateProcess("proc_fwriter", fwriter_callback, fwriter)
17     ;
18     mti_Sensitize(fwriter->process, fwriter->clk, MTI_EVENT);
19
20     // Initialize custom data
21     fwriter->f = fopen("output.txt", "w");
22     if (fwriter->f == NULL) {
23         mti_PrintFormatted("Could not open file!\n");
24         return -1;
25     }
26     return 0;
27 }
```

Listing 4.13: FWriter's init function

Finally, its callback only needs to synchronize with the clock with the same technique of the reader and dump the received data in the output file.

```

1  void fwriter_callback(void *arg) {
2      FWriter *fwriter = (FWriter*) arg;
3      // Check if rising edge of clock
4      if (mti_GetSignalValue(fwriter->clk) != STD_LOGIC_1)
5          return;
6      // Read data
7      long din = mti_GetSignalValue(fwriter->din);
8      int fdata;
9      if (din == STD_LOGIC_0)
10         fdata = 0;
11     else if (din == STD_LOGIC_1)
12         fdata = 1;
13     else
14         return;
15     fprintf(fwriter->f, "%d\n", fdata);
16 }
```

Listing 4.14: FWriter's callback

Values that are neither a 0 nor a 1 are simply ignored.

Both the file reader and writer's VHDL entities and C programs were saved in the simulation directory under the `tb` subfolder, having their complete names be `simulation/tb/freader.vhd`, `simulation/tb/freader.c`, `simulation/tb/fwriter.vhd` and `simulation/tb/fwriter.c`.

4.7 Testbench

A simple Verilog testbench was developed to interconnect the components and observe the behavior of their internal signals. This is another step that, in a future implementation, can be generated automatically by the ToPoliNano system. It is currently not implemented because a meaningful implementation of this system is more complex than it may seem at a first glance.

For instance, a passage of [section 4.4](#) explained that the interface of the wire component is the same as any VHDL digital description since it is a behavioral model. It is not simple to rule-out what this component can be connected to (and how) and what not: a simple restriction based on some information about the technology and its allowed connections provided by MagCAD is not sufficient and this simulation brings an important example of why. Just like MagCAD can generate behavioral VHDL models, so can other tools and developers, as was the case for the MTJ and its wrapper - the problem being that outside modules do not follow MagCAD standards or have its metadata. On top of this, there are various other possibilities: for instance, users may need to connect a purely custom testbench utility for testing purposes which may be ruled-out by too restrictive rules; complex circuits may contain parts that can logically be connected but they are located deep down in two different hierarchies. For this example, all of these problems were faced with the attention of the developer.

Observing the testbench file:

```
1  `timescale 1ns/100ps
2
3  module testbench();
4
5  reg clk = 0;
6  wire multiclck[2:0];
7
8  wire fdata;
9  wire converter_out;
10 wire wire_out;
11
12 always #15 begin
13     clk <= ~clk;
14 end
15
16 genvar i;
17 for (i = 0; i < 3; i = i+1)
```

```
18     assign multiclck[i] = clk;
19
20 FReader freader(.clk(clk), .dout(fdata));
21 D2M d2m(.din(fdata), .mout(converter_out));
22 iNML_custom_wire3 wire3(.CLK(multiclck), .DIN(converter_out), .DOUT(wire_out));
23 FWriter fwriter(.clk(clk), .din(wire_out));
24
25 endmodule
```

Listing 4.15: Testbench

It is possible to see that the testbench also generates the clock to send to the components. Observe that the `multiclck` signal, which is the one that is going to be sent to the `wire3` component, should in theory have three phases. For simplicity it was assigned to the same clock phase since the component only uses one. The clock has a period of 30 ns. After this, the testbench simply instantiates the components and interconnects them.

This testbench was saved in the simulation folder under `simulation/tb/testbench.v`

4.8 Results

The simulation was run under a licensed version of Questa ADMS version 2020.4. To simplify its development and management, a final folder `simulation/sim/` that will contain the `work/` directory and all the simulator-related files was added together with a `simulate.do` file that contains the commands to run.

```
1 # Set paths
2 set SRC_PATH "../src"
3 set TB_PATH "../tb"
4
5 # Set simulation parameters
6 set DURATION "500 ns"
7
8 # Compile sources
9 valog ${SRC_PATH}/pmamtj.va
10 valog ${SRC_PATH}/D2M.va
11 vcom ${SRC_PATH}/definition_inml.vhd
12 vcom ${SRC_PATH}/library_inml.vhd
13 vcom ${SRC_PATH}/wire3.vhd
14
15 # Compile testbench
16 vcom ${TB_PATH}/freader.vhd
17 vcom ${TB_PATH}/fwriter.vhd
18 vlog ${TB_PATH}/testbench.v
19
20 vasim -cmd simulate.cir work.testbench -t 100ps
21
22 add wave *
23
```

```
24 run ${DURATION}
```

Listing 4.16: Simulation commands

Notice that the `vasim` command is called with the `-cmd` flag that contains the SPICE directives for the analog simulator. In this case, it simply sets the analysis to a transient analysis with a maximum step of 1 ps. The simulation can be finally ran by launching:

```
1 $ vasim -do simulate.do
```

Listing 4.17: Launching the simulation

After running a brief simulation, the output window shown in [Figure 4.10](#) reveals the waveforms. Notice that the output of the wire inverts the signal coming from

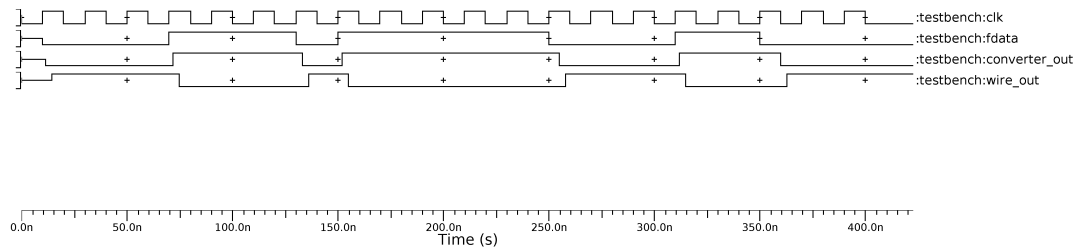


Figure 4.10: Simulation Waveforms

the converter as predicted in [section 4.3](#). The behavior of the circuit appears to be as expected: the values correctly propagate through the components with small yet observable delays, they are correctly transformed by the elements and they are finally written into the output file.

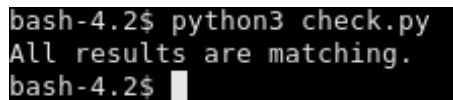
A small Python script internally computes the expected outputs and compares them to the ones generated by this testbench to confirm the correct functionality.

```
1 def gen_reference_outputs(fname: str):
2     WSIZE = 3
3     with open(fname, 'r') as f:
4         window = [0] * WSIZE
5         for line in f:
6             # Skip empty lines
7             line = line.strip()
8             if not line:
9                 continue
10            # Convert value and push it in the window
11            value = int(line)
12            window = [value, *window[:-1]]
13            # Compute output considering the inversion
14            res = 0 if sum(window) > WSIZE // 2 else 1
15            yield res
16
17
18 def read_sim_outputs(fname: str):
19     with open(fname, 'r') as f:
```

```
20         for line in f:
21             yield int(line)
22
23
24 def compare(data, outputs):
25     return all(x == y for x, y in zip(data, outputs))
26
27
28 def main():
29     model, sim = gen_reference_outputs('data.txt'), read_sim_outputs('output.txt')
30     if compare(model, sim) is True:
31         print("All results are matching.")
32     else:
33         print("Results mismatch!")
34
35 if __name__ == '__main__':
36     main()
```

Listing 4.18: Python script to check the outputs

This script uses generators to loop through the two files to finally call the `compare()` function. The simulation output reader simply iterates through the lines and yields the read data; the output generator reads the input data, executes the filtering action and finally yields the results one by one also considering the data inversion that happens in the real circuit. [Figure 4.11](#) shows the console output of this script that confirms the correctness of the results.



```
bash-4.2$ python3 check.py
All results are matching.
bash-4.2$
```

Figure 4.11: Output of the Python checker

Chapter 5

Conclusion

5.1 Conclusion

The result of this work is a new toolset able to handle the simulation of *Hybrid Circuits*, circuit where CMOS and *Beyond-CMOS* technologies collaborate to implement the circuit behavior.

The updates on MagCAD allow the design of Hybrid Circuits thanks to the newly developed Hybrid Plugin, a plugin gathering all the components and features needed to design Hybrid Circuits in which components with heterogeneous descriptions can be placed and interconnected in the same design space. The creation of the Hybrid Plugin also required various updates to MagCAD itself, including the creation of a new design tool called “Wiring Tool”, used to interconnect items in a scene, and MagCAD’s save / load system to add support for the new features.

A new environment based on ModelSim and its Foreign Language Interface (FLI) has been proposed in order to extend the range of possibilities for the description of new models. The standard ModelSim FLI is an extension that allows the inclusion of custom models programmed in C in a classic simulation. In order to simplify the modeling process and to bring it closer to the standards of the ToPoliNano environment, a wrapper C++ library called *flipp* has been developed. On top of adapting some of the standard FLI features from C to C++, it also offers additional simplifications and automatizations of the component description process in the form of its *Model-Behavior* description that allows developers to focus on their models instead of the low level minutia of interacting with the simulation imbued in the standard approach.

5.2 Future Works

Part of the proposed environment has been developed with the ToPoliNano framework in mind, but is still presented under a prototype level since it is not ready to handle user interactions in ToPoliNano's GUI environment. Future developments of these elements will fully integrate it in the framework by also managing the GUI features. It is possible that during the process of integration or the development of new models for the first time, some limitations or additional requirements for the current environment may arise and need to be addressed and adapted.

Bibliography

- [1] Giovanni Finocchio, Massimiliano Di Ventra, Kerem Y Camsari, Karin Everschor-Sitte, Pedram Khalili Amiri, and Zhongming Zeng. «The promise of spintronics for unconventional computing». eng. In: *Journal of magnetism and magnetic materials* 521 (2021). ISSN: 0304-8853 (cit. on p. 72).
- [2] Giovanni Amedeo Cirillo, Edoardo Giusto, Filippo Gandino, and Giovanni Mondo. «Quantum Computing tutorial». ita. In: (2020) (cit. on p. 2).
- [3] U. Garlando, F. Riente, and M. Graziano. «FUNCODE: Effective Device-to-System Analysis of Field Coupled Nanocomputing Circuit Designs». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), pp. 1–1.
- [4] Fabrizio Cairo, Marco Vacca, Giovanna Turvani, Maurizio Zamboni, and Maria-grazia Graziano. «Domain Wall Interconnections for NML». eng. In: *IEEE transactions on very large scale integration (VLSI) systems* 25.11 (2017), pp. 3067–3076. ISSN: 1063-8210.
- [5] Raffaele De Rose, Marco Lanuzza, Felice Crupi, Giulio Siracusano, Riccardo Tomasello, Giovanni Finocchio, Mario Carpentieri, and Massimo Alioto. «A variation-aware simulation framework for hybrid CMOS/spintronic circuits». eng. In: *2017 IEEE International Symposium on Circuits and Systems (IS-CAS)*. IEEE, 2017, pp. 1–4. ISBN: 9781467368537.
- [6] F. Riente, U. Garlando, G. Turvani, M. Vacca, M. R. Roch, and M. Graziano. «MagCAD: A Tool for the Design of 3D Magnetic Circuits». In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 3 (2017), pp. 65–73. DOI: [10.1109/JXCDC.2017.2756981](https://doi.org/10.1109/JXCDC.2017.2756981) (cit. on p. 3).
- [7] F. Riente, G. Turvani, M. Vacca, M. R. Roch, M. Zamboni, and M. Graziano. «ToPoliNano: A CAD Tool for Nano Magnetic Logic». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.7 (July 2017), pp. 1061–1074. ISSN: 0278-0070. DOI: [10.1109/TCAD.2017.2650983](https://doi.org/10.1109/TCAD.2017.2650983) (cit. on p. 4).

- [8] Abhronil Sengupta, Aparajita Banerjee, and Kaushik Roy. «Hybrid Spintronic-CMOS Spiking Neural Network with On-Chip Learning: Devices, Circuits, and Systems». eng. In: *Physical review applied* 6.6 (2016). ISSN: 2331-7019 (cit. on p. 2).
- [9] Xuanyao Fong, Sri Harsha Choday, Panagopoulos Georgios, Charles Augustine, and Kaushik Roy. *Purdue Nanoelectronics Research Laboratory Magnetic Tunnel Junction Model*. Oct. 2014. DOI: [doi:/10.4231/D33R0PV04](https://doi.org/10.4231/D33R0PV04). URL: <https://nanohub.org/publications/16/1> (cit. on p. 74).
- [10] Yue Zhang. «Compact modeling and hybrid circuit design for spintronic devices based on current-induced switching». Theses. Université Paris Sud - Paris XI, July 2014. URL: <https://tel.archives-ouvertes.fr/tel-01058504> (cit. on p. 2).
- [11] Georgios D Panagopoulos, Charles Augustine, and Kaushik Roy. «Physics-Based SPICE-Compatible Compact Model for Simulating Hybrid MTJ/CMOS Circuits». eng. In: *IEEE transactions on electron devices* 60.9 (2013), pp. 2808–2814. ISSN: 0018-9383.
- [12] Mohmmad T. Alam, Jarett DeAngelis, Michael Putney, X. Sharon Hu, Wolfgang Porod, Michael Niemier, and Gary H. Bernstein. «Clocking scheme for nanomagnet QCA». In: *2007 7th IEEE Conference on Nanotechnology (IEEE NANO)*. 2007, pp. 403–408. DOI: [10.1109/NANO.2007.4601219](https://doi.org/10.1109/NANO.2007.4601219) (cit. on p. 69).
- [13] D.K. Reed, S.P. Levitan, J. Boles, J.A. Martinez, and D.M. Chiarulli. «An application of parallel discrete event simulation algorithms to mixed domain system simulation». In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. Vol. 2. 2004, 1356–1357 Vol.2. DOI: [10.1109/DATE.2004.1269085](https://doi.org/10.1109/DATE.2004.1269085).
- [14] R.P. Cowburn. «Digital nanomagnetic logic». In: *61st Device Research Conference. Conference Digest (Cat. No.03TH8663)*. 2003, pp. 111–114. DOI: [10.1109/DRC.2003.1226](https://doi.org/10.1109/DRC.2003.1226) (cit. on p. 69).

Acknowledgements

È mia convinzione che ognuno di noi debba apprezzare il proprio impegno, i sacrifici e gli sforzi che lo portano a raggiungere i traguardi e gioirne ma che, per poterli pienamente apprezzare, bisogna soprattutto riconoscere l'importantissimo ruolo giocato da chi ci circonda. È per questo motivo che desidero dedicare questo breve spazio per ringraziare coloro i quali reputo fondamentali nel raggiungimento di questo obiettivo.

Ringrazio innanzitutto i Dottori Fabrizio Riente e Umberto Garlando, il cui ruolo non è stato solo quello di seguirmi nello sviluppo del lavoro, rivelandosi dei punti di riferimento sempre presenti che mi hanno aiutato nel costruirlo e nell'affrontarne i problemi, ma che nel corso degli anni del loro impegno verso il Politecnico e i suoi studenti hanno permesso a me e ai miei compagni di conoscere ed imparare seguendo nei laboratori, nelle esercitazioni, nei progetti e nei corsi. Il vostro è un ruolo in cui è facile perdere di vista il proprio valore, persi tra le correzioni, le domande e le richieste di decine di studenti; ma spero che il prendere coscienza del fatto che in ognuno di quegli studenti c'è una persona che si sta formando soprattutto grazie a voi e che la muta gratitudine che noi studenti abbiamo nei vostri confronti possano sempre esservi di ispirazione. Allo stesso modo ringrazio il Professore Maurizio Zamboni, la cui passione per il proprio campo e per l'insegnamento lo portano a regalare agli studenti dei momenti indimenticabili in molti dei corsi più emozionanti e formativi del percorso di Elettronica triennale e magistrale, che mi ha permesso di lanciarmi in questo lavoro dal quale ho imparato tantissimo sotto innumerevoli aspetti.

*“Viandante, sono le tue impronte
il cammino, e niente più,
viandante, non c'è cammino,
il cammino si fa andando.
Andando si fa il cammino,
e nel rivolger lo sguardo
ecco il sentiero che mai
si tornerà a rifare.
Viandante, non c'è cammino,
soltanto scie sul mare.”*

Mi disse un giorno un Professore, citando Antonio Machado. Sono tante le cose che mi ha insegnato questo Professore - non soltanto lo studio di funzioni e le equazioni differenziali - e credo che la sua lezione più importante sia proprio questa: che noi siamo le nostre impronte, e che il nostro cammino non è che una scia sul mare. Se è davvero così, Professore Cortese, mi ritengo onorato per avere incrociato la sua scia, e le sono eternamente grato perché proseguirò con il mio cammino solo grazie ai suoi insegnamenti. Mai dimenticherò le sue *disfide*, le sue riflessioni e la sua gentilezza, e spero che un giorno io possa diventare, anche in piccola parte, ciò che lei è.

Sono grato della fatto che le mie radici affondino in un ambiente che per me resterà sempre parte della mia famiglia, il Don Bosco Ranchibile di Palermo, dove tutti i professori ma in particolar modo il Professore Lamia, il Professore Biondi e il Professore Sofia hanno non solo acceso ed alimentato la mia passione per la scienza, ma hanno sempre creduto in me e mi hanno aiutato a crescere e diventare la persona che sono. È durante quella lezione in cui Professore Sofia introdusse i circuiti logici alla classe che mi sorpresi per aver provato un'indescrivibile emozione davanti ad un suo disegno di un circuito digitale, momento che riconosco essere il seme per le scelte di vita che ho compiuto fino ad oggi.

I miei amici, sempre presenti nella gioia e nelle difficoltà, sono quelli a cui devo tutto me stesso. Sono cresciuto insieme a Sofia, Domenico e Antonio, e anche se le nostre strade a volte ci allontanano ed altre ci riavvicinano, siete tutti parte di me e vi ringrazio per avermi regalato tutto quello che va oltre lo studio ed il lavoro: la vita vera e propria. Ci siamo conosciuti bambini, insieme diventiamo adulti e spero di poter invecchiare con voi. Riunirmi con i vecchissimi amici - Mariano e Gabriele - ed incontrare i nuovi - Federica e Martina - colora i miei giorni, riaccende le vecchie passioni come il Pianoforte e mi rende felice. Grazie soprattutto a chi, nonostante sia appena entrato nella mia vita, sembra farne parte da anni.

Infine, se esiste una *casa* oltre le quattro mura della nostra dimora, quella per me è la famiglia, che sarà sempre con me, ovunque io mi trovi nel mondo. La nonna

Antonietta alla quale voglio un bene immenso, i nonni che non sono più tra noi, gli zii, i cugini e soprattutto Mamma e Papà: tutte le persone del cui amore non dubiterò mai, vi ringrazio davvero per tutto. Mi avete visto nascere e siete con me in uno dei giorni più importanti della mia vita, spero che siate orgogliosi di chi sono e di chi lavoro per diventare.

A tutti voi che siete parte della mia vita, Grazie.

“All we have to decide is what to do with the time that is given us.”

- J. R. R. Tolkien