

The Madeo framework

MADEO FET: Compiling for reconfigurable architectures
MADEO BET: Generic backend tools for reconfigurable
architectures

Loïc Lagadec

September 25, 2006

LESTER FRE 2734
ARCHITECTURES ET SYSTEMES
Université de Bretagne Occidentale
<http://as.univ-brest.fr>
mail: loic.lagadec@univ-brest.fr

Contents

I	MADEO BET	17
1	Starting guide	19
1.1	Flow	19
1.2	Installation (obsolete)	21
2	Designing an architecture	25
2.1	Principles of design	25
2.1.1	Method of design	25
2.1.2	Hierarchical structure	26
2.1.3	The atomic elements	27
2.1.4	Accessing and connecting elements	31
2.1.5	Additional parameters	33
2.2	Using <i>Architecture Designer</i>	36
2.2.1	Loading/Saving an architecture	36
2.2.2	Accessing/editing a description	38
2.2.3	Starting with a new description	39
2.2.4	Creating an architecture	39
2.2.5	Included examples	40
2.2.6	Managing dependencies between elements	40
2.3	Generating documentation	41
2.3.1	Principles	41
2.3.2	Using MADEO-GENDOC	41
2.4	VHDL generation	42
2.4.1	Principles	45
2.4.2	Using <i>VHDL Producer</i>	45
3	Programming an architecture	47
3.1	Introduction	47
3.2	Towards new needs	47
3.2.1	Algorithms	47
4	Tools	49
4.1	Using <i>FPGA Editor</i>	49
4.1.1	Application description	50
4.1.2	Selecting BLIF/EDIF files	50
4.1.3	Placing and routing a BLIF/EDIF file	52

4.1.4	Pasting the module	52
4.1.5	Simulating the circuit	56
4.1.6	Using the window commands	56
4.2	Floor planning	57
4.3	Drawing regular circuits	62
4.3.1	Introduction	62
4.3.2	Structural representation	62
4.3.3	Compatibility	63
5	Architectural prospection	65
5.1	Introduction	65
5.2	Tools	65
5.2.1	Starting UIProspection	65
5.2.2	Analyzing an architecture	65
5.2.3	Selecting a BLIF/EDIF file	66
5.2.4	Modifying the value of an element	67
5.2.5	Collecting the results	68
5.2.6	Performing the prospection	69
II	MADEO FET	73
6	introduction	75
6.1	Execution model	76
6.2	Type system	78
7	design flow and optimizations	79
7.1	design flow	79
7.2	Possible Types	81
7.3	Optimizations	81
7.3.1	type inference	81
7.3.2	code factorization	85
7.3.3	dead code removal	85
7.3.4	constant folding	85
7.3.5	no op removal	86
7.3.6	operator fusion	86
7.3.7	Automatic decomposition	86
7.3.8	Operator flattening	87
7.4	Logic generation	87
7.4.1	Blif generation	87
7.4.2	EDIF Generation	88
7.4.3	Mixed netlist	89
7.5	Tips	89
7.5.1	Temporaries	89
7.5.2	ArrayedResult	89
7.5.3	Array	89
7.6	the <i>at</i> : message	90

7.6.1	Comparisons	90
8	Graphical tool	93
8.1	Tool	93
8.1.1	Control panel	93
8.1.2	The ToolBar	95
8.1.3	Code menu	96
8.1.4	Graphic representation	98
8.2	Interfacing low level tools (MADEO-bet)	100
9	Example: Defining a floating point multiplier	101
9.1	Introduction	101
9.2	The implemented algorithm	101
9.3	Implementation	102
9.3.1	The methods	102
9.3.2	The types	103
9.3.3	The resulting graph	104
9.3.4	Towards infinite precision operators	109
9.3.5	Conclusion	110
10	Schematic design	111
10.1	Introduction	111
10.2	Position within the flow	111
10.3	Tool	112
10.3.1	Example	113
10.3.2	Composite nodes and hierarchical visiting	114
10.3.3	Conclusion	115
III	Appendix	117
A	MADEO BET appendix	119
A.1	Some examples of architectures description using the gram- mar	119
A.1.1	First basic example	119
A.1.2	Example 2	121
A.1.3	Example 3	123
A.1.4	Example 4 with custom representation	125
A.1.5	LPPGA	126
A.2	Grammar	130
A.3	Heterogeneous architectures	134
A.4	Inspecting changes over the definition of an architecture . . .	135
A.5	Textual output	136
A.6	Changing the cost function of the placer	138
A.7	Prospection results	139
A.8	Example of prospection	142
A.8.1	History of prospecting result	142

A.8.2	Average of prospecting result	143
B	GenDoc results	145
B.1	LPPGAArray	147
B.1.1	Architecture Design (Level 1)	147
B.2	LPPGACell2	148
B.2.1	Architecture Design (Level 2)	149
B.3	MySwitchLPPGA	151
B.3.1	Architecture Design (Level 3)	151
	Annexes	152
	Annexes 1: Sources LPPGACell2	152
C	Madeo fet appendix	157
C.1	Blif vs Edif description	157
C.1.1	smalltalk code	157
C.1.2	Resulting EDIF	157
C.1.3	Resulting Hierarchical Blif	160
C.2	Resulting Flatten Blif	164
D	Sis scripts	167
E	Inference	169
E.1	CIR_MethodGraph	170
E.2	CIR_Block	171
E.3	CIR_Operator	173

List of examples

1.1	Launching the environment	21
2.1	The ARRAY syntax	26
2.2	The COMPOSITE syntax	27
2.3	The FUNCTION syntax with explicit naming	28
2.4	The FUNCTION syntax with global IOs	28
2.5	The WIRE syntax	29
2.6	The REGISTER syntax	29
2.7	The MULTIPLEXER syntax	30
2.8	The SWITCHBLOCK syntax	30
2.9	The REFERENCE syntax	31
2.10	Naming the elements	31
2.11	Accessing the elements	31
2.12	The LINK syntax	32
2.13	The LINK alternative syntax	32
2.14	The CONNECTION syntax	32
2.15	The alternative CONNECTION syntax	33
2.16	The PRODUCE syntax	33
2.17	The CATEGORY syntax	34
2.18	The REPRESENTATION syntax	34
2.19	The custom parameters syntax	35
2.20	The BASECOST syntax	36
4.1	The floorplanner cost function	59
4.2	Setting the penalty	59
4.3	First example	60
4.4	Second example	60
4.5	A serial adder	64
5.1	Defining MV-variables	68
7.1	Literals	81
7.2	Intervals	81
7.3	Radix based	81
7.4	Unions	81
7.5	GF16	81
7.6	GF128	81
7.7	Swapping the most/less significant four bits for an integer ranging from 0 to 63	88
7.10	when comparing a value to a literal, the literal must be the receiver	90

7.8	Swapping the most/less significand four bits for an integer ranging from 1 to 63	91
7.9	end of the 7.8 example	92
9.1	The entry point. Each float appears as three values : a sign, a significand, and an exponent	102
9.2	The first operation over the exponents	102
9.3	The operations over the signs	102
9.4	The full operation over the significands, ignoring the wished data width	102
9.5	The significand normalization implies to shift and truncate the value before returning it, but to answer as well the shift that was applied in order to carry that shift over the exponent value	103
9.6	First type	103
9.7	Second type	104
9.8	Third type, with its associated values, either fraction or float, ranging from 1 to $2 - \epsilon$ with $\epsilon = 2^{-3}$	104
9.9	The Blif description of the 9.4 table enlightens two characteristics of the node : first, the $R_{out}(op)$ reduction factor is important as the inputs are encoded using 6 bits and the outputs only require 4 bits, and secondly the outputs are called $t39_1..3$ and $t40$, what means the nodes owns two outputs, the first of which requires 3 bits and the other one only one bit.	104
10.1	The equation description of the adder	113
10.2	The two bits adder	114
A.1	An heterogeneous architecture	134
A.2	The new method generated from A.1	134
A.3	Placement information	136
A.4	Routing information	137
A.5	Example of cost function	138
D.1	The 5lut.script doing the technology mapping for 5-Luts	167
E.1	the <i>inferTypes</i> method	170
E.2	The <i>inferOutputsType</i> method	170
E.3	the <i>inferType</i> method	171
E.4	the <i>inferTypes</i> method	171
E.5	the <i>inferOutputsTypes</i> method	172
E.6	The <i>inferOutputsTypes</i> method	173

List of Figures

1.1	MADEO BET flow	20
1.2	Snapshot of the common environment	22
1.3	Snapshot after the installation process has completed	22
1.4	Saving your image	23
1.5	Smalltalk menu	23
1.6	Architecture Designer	23
2.1	Principles of design	26
2.2	View of the functions IOs connections	27
2.3	Global view of the model	28
2.4	Example of action zone of wire	29
2.5	Two examples of switches	30
2.6	Some examples of the default user interface over the basic elements of he architecture	35
2.7	Loading an architecture description (1)	37
2.8	Loading an architecture description (2)	37
2.9	Selecting a description	38
2.10	Choosing a category	38
2.11	Code generated by <i>build</i>	39
2.12	The tree representation of a composite	41
2.13	The GenDoc user interface	42
2.14	The GenDoc parameters	43
2.15	The GenDoc actions	43
2.16	The GenDoc transcript	44
2.17	The GenDoc html output	44
4.1	The interface of <i>FPGA Editor</i>	49
4.2	The application description formalism	50
4.3	Selecting a BLIF/EDIF file from the menu	51
4.4	BLIF/EDIF File Browser	51
4.5	selecting a BLIF/EDIF file	52
4.6	The list of available modules	52
4.7	Pasting the module on the FPGA	53
4.8	After pasting the module on the FPGA	54
4.9	Selecting the zoom factor	54
4.10	Representing either all or only used resources	55

4.11	The simulation of P&R circuits	56
4.12	Floorplanning 1	58
4.13	Floorplanning 2	58
4.14	Floorplanning 3	60
4.15	The result of the example 4.3	61
4.16	SCCompositeNodes	62
5.1	Launching UIProspection	66
5.2	Prospection Designer	66
5.3	Viewer button	67
5.4	Detailed view of the selected element	67
5.5	Result of a single place and route	68
5.6	Automation of prospection	68
5.8	Selecting the type of result	68
5.7	Keyboarding a new value	69
5.9	The UIElements interface	70
5.10	The elements to be prospected	70
5.11	Setting the number of runs for prospection	71
5.12	Latex output	71
6.1	The modules can be either flat or hierarchical; the modules can be composed in order to produce pipelines or can be in- stantiated during architecture synthesis.	75
6.2	State machines can be obtained by methods operating on pri- vate variables having known initial values.	76
6.3	Fan-in from 2 nodes with $Card(fout \times gout) < Card(fin) \times$ $Card(gin)$	77
7.1	MADEO BET and MADEO FET flow	80
7.2	A node links an output to some inputs, and computes the output current value, depending on the inputs' current value in order to build the HL OO LUT	82
7.3	The starting code encapsulating an ArrayedResult. Note that the code is structured automatically by extracting all nodes that belong to the inheritance tree of the values	84
7.4	After the rewriting process, only three nodes remain. Note the yellow color that denotes the hierarchical nodes.	84
7.5	code factorization	86
7.6	The at: semantic	90
8.1	The MADEO-FET launching procedure	93
8.2	User Interface	94
8.3	Control panel	94
8.4	Context menu	95
8.5	Madeo-Fet toolBar	95
8.6	Code menu	96
8.7	CIR menu	97

8.8	BLIF menu	97
8.9	SCNode menu	98
8.10	The graph representation	99
8.11	MADEO FET - MADEO BET interfacing	100
9.1	A small floating Point multiplier	105
9.2	The reduced 9.1 graph	106
9.3	The 9.2 graph with operator fusion	107
9.4	The LUT of the <i>normalizeSignificand</i> : operator	108
9.5	The global LUT	109
9.6	Verification over the circuit	109
10.1	Interfacing MADEO-FET and the schematic editor	111
10.2	The Schematic Editor launching procedure	112
10.3	The schematic editor	112
10.4	The SCNode Chooser	113
10.5	Flattening nodes	115
A.1	Architecture viewer	135
A.2	Analyzing the bounding box	139
A.3	Analyzing the CPU time	139
A.4	Analyzing the number of unrouted signals	140
A.5	Analyzing the routing cost	140
A.6	Analyzing the number of runs (20 max)	140
A.7	Analyzing several criteria	141

Introduction

Reconfigurable architectures offer a promising trade-off between the ASICs' peak performances and the flexibility the software engineers are used to. Their "tunable" nature widens their use scope, allowing to recoup the non recurring costs and makes them financially attractive.

Unfortunately, these architectures often suffer from a poor development environment preventing short time to market, and slowing down hardware evolutions.

From our point of view the two major points that deserve RA are:

- First, the application designing front end does not fully exploit the hardware capabilities; this is mainly due to the VLSI CAD domain from which emerged the RA CAD nearly twenty years ago. A consequence is that applications are commonly described as VHDL programs that make use of C-like typed operators (8,16,32 bits operators) that are present in libraries despite the hardware permits to implement custom operators, with better performances.
- Secondly, the back end tools, that are needed to program such architectures, are strongly linked to their target architecture, exhibit poor reuse and bring a long development process for each new architecture. This prevents the designers from testing their architectures at an early stage, prior to any physical realisation, as these tools are the only solution to get an accurate feed back over a design choice. In addition, linking CAD tools to an architectural model makes architectures comparisons harder, and increases the probability of defective software whereas reuse speeds up the development process, increases productivity and software quality.

The fact is that by keeping secret the technical informations the CAD tools designers need, the FPGAs vendors refuse third parties to bring up their own tools.

The Madeo project is an attempt to answer to these limitations. This project has been on since 95/96 and came after the ArMen project. One of the concluding remarks of the ArMen project was that to be dependent of the CAD tools the FPGAs vendors provide is highly time consuming as no API is preserved from one generation to another. Another conclusion was that programming a complex hardware mixing several computational

models requires a high level unified front end making use of code generation on demand. As a consequence, MADEO makes use of a non-typed functional language when designing application, and prohibits any commercial back-end tools; MADEO is made of two layers, as introduced below:

The top layer called MADEO-FET produces RTL logic or EDIF netlist, from a functional code (with no types) from one hand, and a context from another hand. A context describes the set of different values for every variable. By performing types inference, and several more classical optimizations, the compilation process results in a hierarchical graph of operations each of which owns a minimized logic description.

Implementing this graph cannot be archived based on library based operators. On the contrary, this stage requires to be able to produce custom operators, before composing them. This is done by invoking the back end layer.

The back-end layer called MADEO-BET produces a textual "bitstream" from a hierarchical RTL description (BLIF, EDIF, or an internal format supporting regular circuits) from one hand, and a modelization of an architecture from another hand. The designer benefits from a set of hardware elements (functions, switch, multiplexers, etc...) that can be combined and/or specialized to describe a given architecture. This stage is archived through a compilation process, after the designer has described within a private language its architecture.

A set of generic tools (P&R, floorplanner, regular editor, etc...) manipulates this architecture, with as main benefits, the capability to describe any architecture and to do it fast, to share the tools between all these architectures, and to get feedback over architectures.

It is also possible to automate this architectural variation, and to collect feedback to drive the designer's choices regarding a class of target application to implement.

VHDL descriptions are to be added to offer the designer a way to go up to the physical synthesis of its architecture. This will benefits as well to the tools by returning after-synthesis accurate information to tune the algorithms parameters.

This report introduces the MADEO framework from a user point of view, starting with MADEO-BET (chapters 2 to 5) and considering MADEO-FET as a second part.

The first chapter is a short starting guide through the installation procedure.

Part I

MADEO BET

Chapter 1

Starting guide

Introduction

MADEO-bet[7] [8] [6] is a framework composed of a set of back-end tools for modeling and programming reconfigurable architectures. MADEO-bet comes within the framework of a more general set of tools, targeting the programming and the manipulation of reconfigurable architectures at a very high level of abstraction, using a symbolic and non-typed environment. MADEO-bet provides the generic back-end in this compilation flow.

This chapter does not introduce the upper stages of MADEO which are described in part II, but focuses on practical information regarding the back-end tools. This report is splitted into several sections: the first one describes the installation procedure. The second section introduces the modelization stage while the next section describes its associated tool: the *Architecture Designer*. The next chapter details the way to program a architecture while the following one addresses the tool allowing to program a modeled architecture, including floorplanner and regular circuit editor. The last chapter details how to automate the architectural prospection in front of a fixed circuit.

1.1 Flow

CAD tools for reconfigurable architectures commonly suffer from several lacks. First, except few tools such as VPR [2], they are dedicated to a single architecture, or at most to a family of architectures. It's a fact that taking advantage of a cad tool may prevent some bad decision when designing a new architecture. Designing an architecture meanwhile developing its associated cad tool can lead to serious problem as illustrated by the PROTEUS project [4].

Another point is that a tool which fits to any architecture, allows not simply testing the architecture, but more, allows comparing several architecture. And Last, by reusing most of the code, such a tool decreases the required amount of development time.

Secondly, CAD tools do not exhibit any feature that would allow to draw

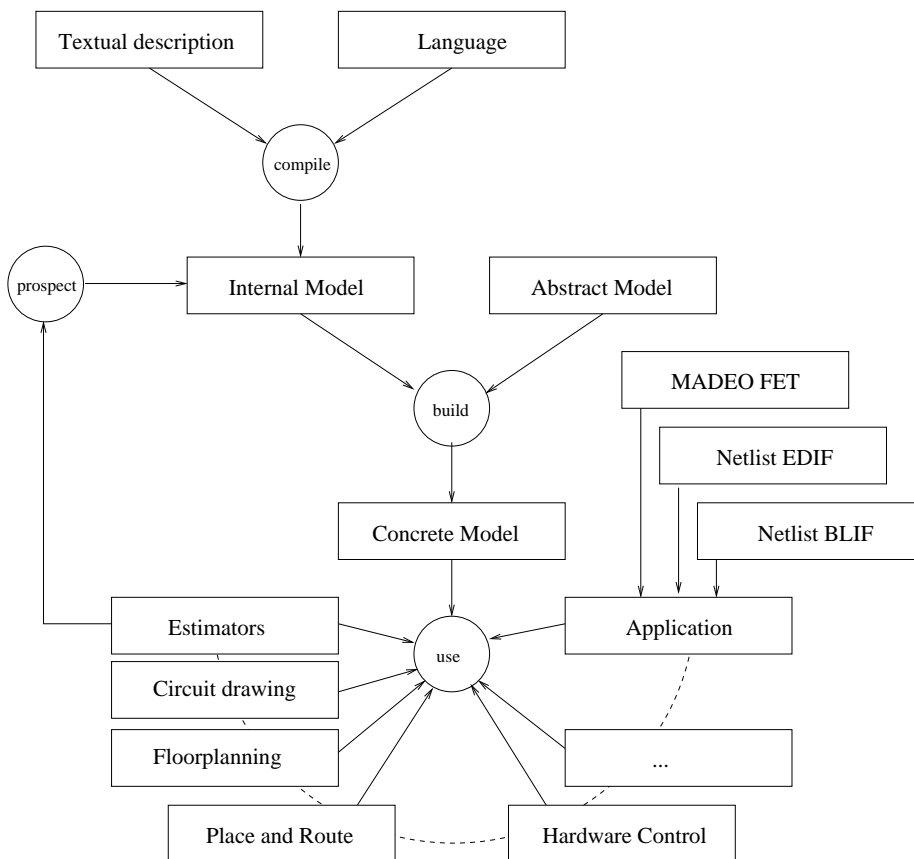


Figure 1.1: MADEO BET flow

regular circuits, based on an existing application structuring. Designers claim the tools must bring new functionalities such as hot debugging by interfacing the real architecture if any, or supporting several abstraction levels that could be mixed or exchanged one to the other, or drawing regular circuits.

MADEO-BET focuses on these points; and provides a generic backend tool that can adapt to any architecture, according this architecture is described within a custom language. In addition, by providing a stable and common back end to a higher level compiler, MADEO BET lets MADEO-FET benefit from accurate information when electing a design policy for an application both depending on the hardware and the application field (see part II).

1.2 Installation (obsolete)

Requirements In this section, we assume that Visualworks [1] has been previously installed¹. As well, we assume that all the files of the MADEO-bet package have been copied into a local user directory.

The MADEO-bet package The MADEO-bet package is composed of the following Visualworks files :

- MADEO-bet.pcl, madeo-bet.pst
- EDIF.pcl, EDIF.pst
- visualxx.im (where xx represents the Visualworks version number).
the 30 version runs on windows, and the 31 version runs on Linux 2.2x kernels
- additional files such as some BLIF/EDIF files

These files are contained in an archive file name imagexx.tgz (where xx represents the Visualworks version number).

The installation The installation process starts with launching the Visualworks environment in a similar manner than the code provided in 1.1.

```
$(VISUALWORKS)/bin/visualnc visualxx.im
```

CODE EX. 1.1: Launching the environment

Be careful of choosing as current directory the one in which the MADEO-bet files are present.

After this is done, the user' screen must be close to the one of figure 1.2. The upside windows is called VisualLauncher. Click then on the **start installation** button to install the package.

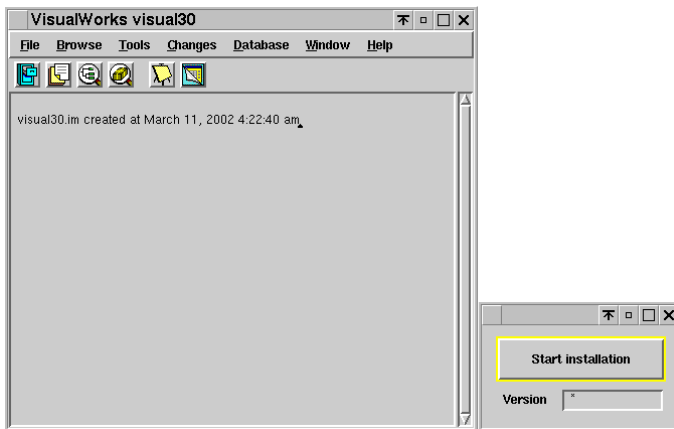


Figure 1.2: Snapshot of the common environment

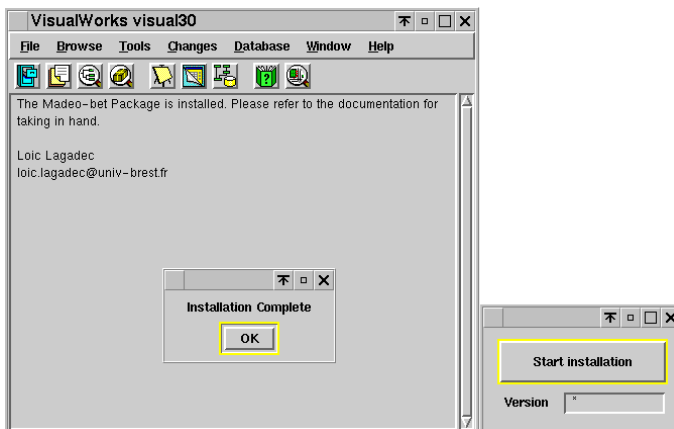


Figure 1.3: Snapshot after the installation process has completed

After the installation completes (figure 1.3), it is recommended to save the resulting image (figure 1.4).

Opening the graphical interface Designing a new architecture relies on the *Architecture Designer* tool (figure 1.6) which is accessible through the last icon (figure 1.5).

The full description of the FPGA takes place in this interface.

¹To install Visualworks, refer to <http://www.cincon.com>

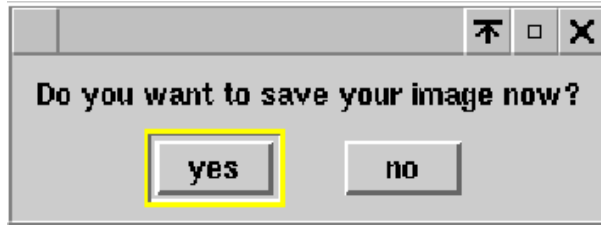


Figure 1.4: Saving your image



Figure 1.5: Smalltalk menu

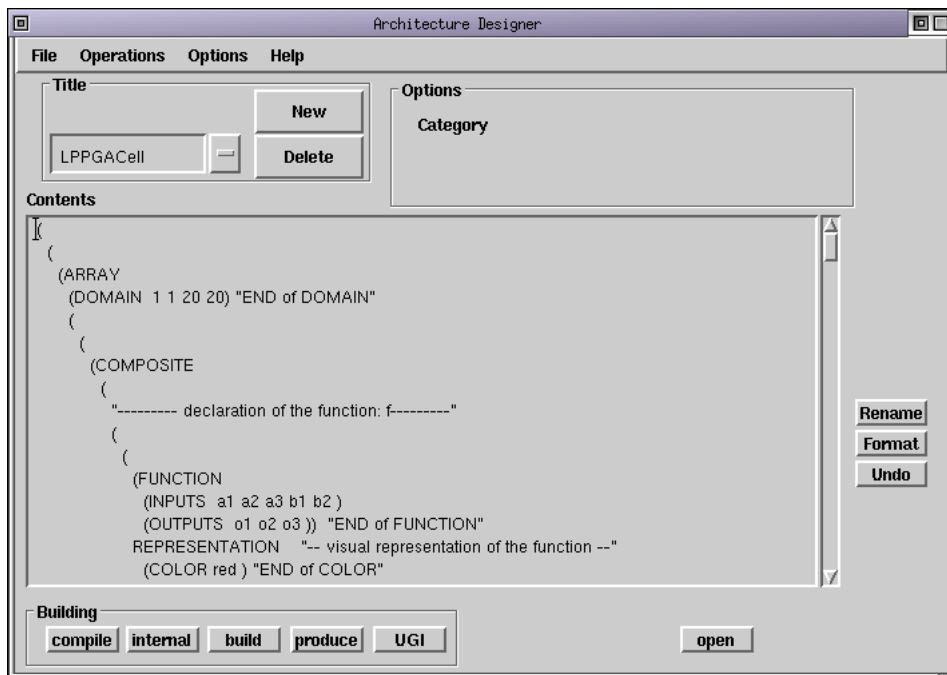


Figure 1.6: Architecture Designer

Chapter 2

Designing an architecture

2.1 Principles of design

Within MADEO-bet, a reconfigurable architecture appears as a collection of 2-D grids of identical tiles. Each of these tiles can be either an atomic elements or a hierarchical composition of elements (figure 2.1).

The hierarchical elements contain either a 2-D grid of sub-elements owning their private position or a set of sub-elements that share the same location (e.g sub-elements within a CLB).

Atomic elements are functions, wires, switches, etc. . .

Example The architecture which serves as an example along this report is LPPGA developed by V. George [5]. The full model of this architecture is provided in appendix A. The FPGA appears as a 2D-grid of cells. These cells are composed of a logical function, a switch, some wires and some connections between these various elements (figure 2.2). However it's possible to define a more complex architecture for the elements of the table. For the LPPGA architecture there is only one level in the hierarchic architecture (figure 2.3).

2.1.1 Method of design

The design of a FPGA with *Architecture Designer* is based on a grammar provided in the appendix B of this document or obtained while going in the menu *Help* $\dot{}$ *BNF description*. From this grammar it's possible to represent the architecture of the FPGA in a detailed way.

Once the architecture description is validated, Smalltalk classes are generated (paragraph 2.1.5) which represent every level of the hierarchy.

Despite these classes provide a sufficient support for the architecture, the developer has the opportunity to modify/optimize the produced code by hand in a classical Visualworks way (section 4.1.6).

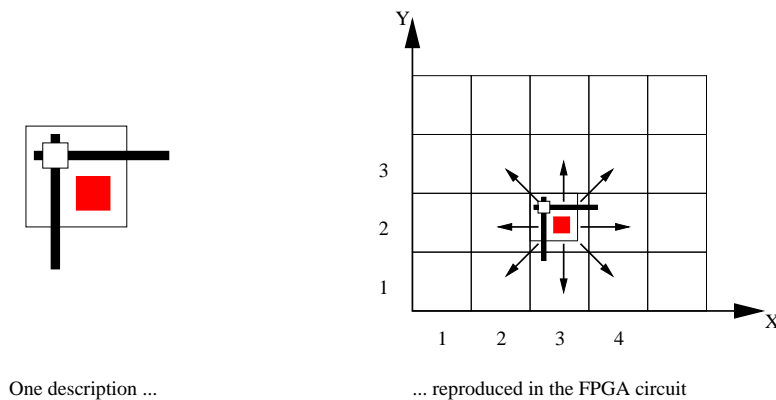


Figure 2.1: Principles of design

2.1.2 Hierarchical structure

2D-Grids

A 2D-grid describes the regular replication of a given pattern. The information needed is the grid surface and its position on one hand, and the element to be replicated on the other hand.

```
(ARRAY
  (DOMAIN 1 1 x y)
  element
)
```

CODE EX. 2.1: The ARRAY syntax

Note that heterogeneous architectures can be built by defining a set of domain-element couples. This is illustrated by the appendix A.3.

The composite elements

A composite element enables to group several objects within a single container in order to favor reuse/replication of elements. The composite elements do not only provide structural description but describe connections between elements as well.

In the example, the composite element corresponds to the FPGA cell. It recovers the list of elements as well as connections. The code representing the creation of a composite is as follows:

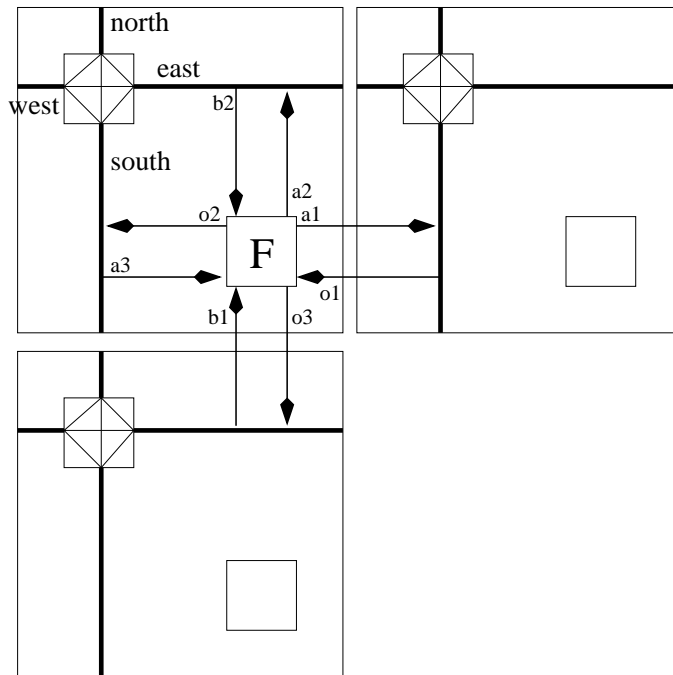


Figure 2.2: View of the functions IOs connections

```

(COMPOSITE
  (
    list_of_elements
  )
  list_of_connections
)

```

CODE EX. 2.2: The COMPOSITE syntax

Note: All the elements contained within a composite are accessible through their symbolic naming, so that all of them must be provided a name (2.1.4).

2.1.3 The atomic elements

The set of available atomic elements is extensible under certain circumstances. This subsection does not detail the object-oriented rules the developer must conform to when creating a new kind of atomic element but introduces the most commonly used elements.

The functions

A function is a basic element which corresponds to the logical function of the cell. It is made up of a list of inputs and outputs and accepts potentially any "processing unit" assuming the IOs are compatible. However, the possible functions can be restricted to a set of symbols on demand.

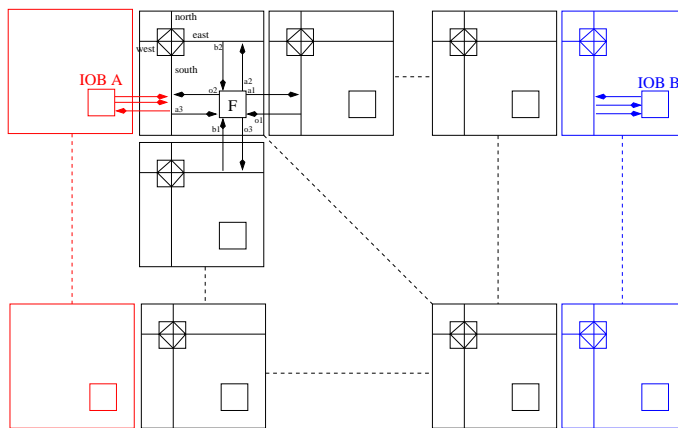


Figure 2.3: Global view of the model. Note that the IOBs are encapsulated within composite elements in order to add lacking resources such as the right most vertical channel.

It is created in two ways, according to the kind of IOs enumeration (see codes 2.3, 2.4).

```
(FUNCTION
  (INPUTS list_of_inputs )
  (OUTPUTS list_of_outputs )
)
```

CODE EX. 2.3: The FUNCTION syntax with explicit naming

```
(FUNCTION
  (INPUTS named input wire )
  (OUTPUTS named output wire )
)
```

CODE EX. 2.4: The FUNCTION syntax with global IOs

The wires

A wire is described by its width. A wire with a unary width is a single wire whereas a wire with a width above one can be either seen as a wire or as a channel owning tracks. This drives the behavior of the router as using single wire leads to global routing and using channels provides a detailed routing.

The wires do not set their range neither their population; these two characteristics are computed regarding the connections (see 2.1.4).

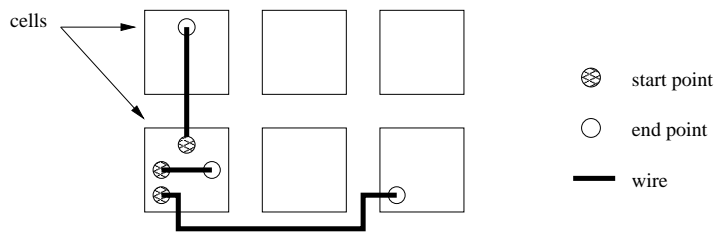


Figure 2.4: Example of action zone of wire

These two examples illustrate the use of a wire versus the use of a channel:

-
1. WIRE (WIRE (WIDTH *size_of_wire*))
 2. SIMPLE CHANNEL (WIRE (WIDTH *size_of_wire*) EXPANDED)
 3. COMPLEX CHANNEL (WIRE (WIDTH *size_of_wire*) EXPANDED OF (*wire* *))
-

CODE EX. 2.5: The WIRE syntax

Note that there is no restriction on sub wires. Sub wires can be simple wires, channel or complex channels; they can be described as a link (see further on); they can be associated an individual representation.

The registers

```
(REGISTER
  (INPUT string_describing_a_wire )
  (OUTPUT string_describing_a_wire )
)
```

CODE EX. 2.6: The REGISTER syntax

The multiplexers

The switches

The switch permits to carry out the interconnection between two wires. It can carry out a regular interconnection or not (see fig. 2.5). In the case

```
(MULTIPLEXER
  (INPUTS string_describing_a_wire * )
  (OUTPUT string_describing_a_wire )
  (WIDTH width)
)
```

CODE EX. 2.7: The MULTIPLEXER syntax

of the LPPGA the switch is regular. The definition of the switch is based on two parts, the first corresponds to the enumeration of the wires which are used by the switch, the second gives all connections carried out between different the wires used in switch. The creation of the switch is made using the code 2.8:

```
(SWITCHBLOCK
  (RESOURCES
    (
      name_of_wire1
      name_of_wire2
      ...
    )
  )
  list_of_connections
)
```

CODE EX. 2.8: The SWITCHBLOCK syntax

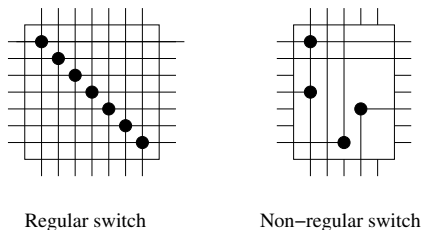


Figure 2.5: Two examples of switches

In a matter of simplicity, a standard switch interconnecting wires of the same width is assumed to be a regular switch (see figure 2.5). In case widths differ and no particular connectivity is specified, the switch will be all-to-all connected. Note that this can bring surprising results when performing prospection (section 5) if the designer is not aware of linking the wires width.

The references

The references permit to call upon objects previously defined in another architecture to re-use them in new architectures. As every complex object is represented through a Smalltalk class, the name of the class acts as a parameter for reference. More information on classes definition is presented in the section **Category** (2.1.5).

```
(REFERENCE name_of_Class_element )
```

CODE EX. 2.9: The REFERENCE syntax

2.1.4 Accessing and connecting elements

Naming the elements

Naming the elements is required when aggregating elements within a composite. Note that the elements of the 2D-Grid are implicitly named by their position within the grid.

```
( unnamed_element NAMED name )
```

CODE EX. 2.10: Naming the elements

Accessing the elements

A hierarchical element accesses to its sub-elements using the *at:* syntax. The parameter is either a symbol or a point defining the position of the sub element.

As all elements know their container, relative addressing is also possible. Sending the *container* message to an element returns its container.

The example 2.11 addresses the element named *f*, contained in the same container that the requester.

```
myElement container at: #f
```

CODE EX. 2.11: Accessing the elements

Using aliases

For convenience some resources appear in several naming space (i.e. several composite elements). This is done using aliases which enable to delude the elements about the resource owning, without replicating shared resources. For example, shared lines often take advantage of owning several names.

```
(
  (LINK '(self relativeAt: position_of_target_cell) name_of_wire')
  NAMED name_of_second_wire )
```

CODE EX. 2.12: The LINK syntax

As the architecture appears as a regular pattern replication, some problems may happen on the borders. As an example, defining a tile made of a left and a up routing channel will produce degenerate right and bottom borders as some resources defined as links won't exist. A solution is to specify an exception to be raised on demand using an alternative syntax (code 2.13).

```
(
  (LINK '(self relativeAt: position_of_target_cell) name_of_wire'
  IFNONE definition_of_the_alternative_element)
  NAMED name_of_second_wire )
```

CODE EX. 2.13: The LINK alternative syntax

Another solution is to define an heterogeneous architecture (made of several domains), what allows to tune the borders to match the regularity requirements.

Connecting elements together

Connections are used to connect all elements of the cell. There are two types of connections: connections related to outputs and connections related to inputs.

```
(CONNECTION
  'object_containing_output1 connect: output_name_1 to: wire'
  'object_containing_output2 connect: output_name_2 to: wire'
  ...
)
```

CODE EX. 2.14: The CONNECTION syntax

```

(CONNECTION
  '(object_containing_input1 at input1) connectTo input2 of ob-
ject_containing_
input2'
  ...
  OR
  '(object_containing_input1 at input1) connectTo (ob-
ject_containing_input2 at input2)'
  ...
)

```

CODE EX. 2.15: The alternative CONNECTION syntax

For the second connections, two equivalent writings are available:

The connections are not limited to the current cell; it is possible to reach any cell of the circuit on demand through absolute/relative element addressing (see 2.1.4).

2.1.5 Additional parameters

Produce

Marking out PRODUCE gives the possibility to create a Smalltalk class which name is passed as a parameter. Note that the name of class must to start with a **capital letter**.

```

(
  (
    ( definition_of_element )
    NAMED name_of_element )
  PRODUCE Name_of_Class )

```

CODE EX. 2.16: The PRODUCE syntax

Category

Smalltalk classes are organized within categories. It's thus significant to group all the classes concerning the FPGA modeled in a single category. Marking out CATEGORY permits to direct the class generated by PRODUCE.

```

(
  (
    (
      ( definition_of_element )
      NAMED name_of_element )
      PRODUCE name_of_Class )
    CATEGORY name_of_Category )

```

CODE EX. 2.17: The CATEGORY syntax

Representation

Simple graphical representations can be generated automatically based on some grammar informations. These informations are provided using the REPRESENTATION syntax element that takes place between the end of the definition of an object and the attribution of the name of this object.

```

(
  ( object_definition )
  REPRESENTATION
    ( option1_of_representation )
    ( option2_of_representation )
)

```

CODE EX. 2.18: The REPRESENTATION syntax

The options of REPRESENTATION are

- (COLOR *color*) to define the color of display of the object.
- (RECTANGLES *x1 y1 x2 y2*) to represent the object in form of a rectangle defined by the coordinates of the corner in top on the left (*x1,y1*) and the corner in bottom on the right (*x2,y2*).
- (CHANNEL *x1 y1 x2 y2 vx vy*) to represent the wires in the cell: (*x1,y1*) is starting point, (*x2,y2*) is ending point of the wire. The vector (*vx, vy*) represents spacing between the wires when option EXPANDED is used in the definition of the wire.
- (TEXT *x y information_to_be_displayed*) allows to display text starting from the position (*x,y*).
- (*name_of_wire* (LINE *x1 y1 x2 y2*)) used by the switch this option gives space fixes to use to display the connected wires.

- (CUSTOM *name_of_the_custom_method*) enables to reuse custom representation whereas the generic representation is rebuilt.

Defining custom UIG for elements

The elements of the architectures are provided default UGI as shown in the 2.6 figure. Refining the *interfaceClass* parameter, associates a new hand made interface. The parameter must provide the name of the class describing the interface.

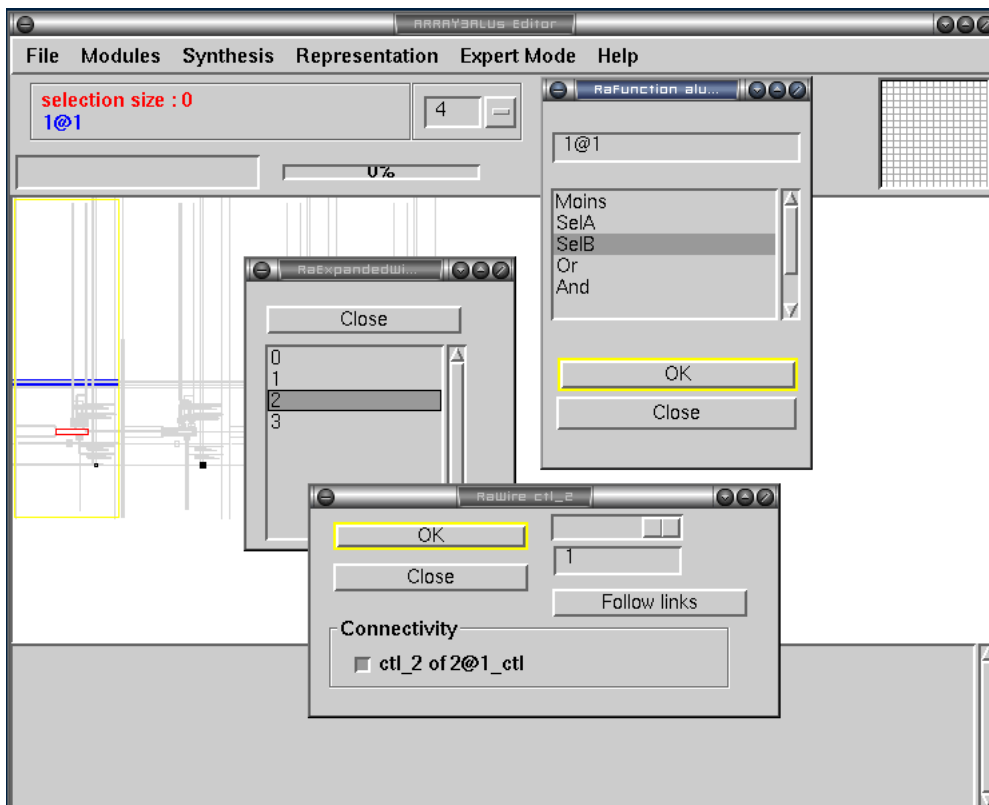


Figure 2.6: Some examples of the default user interface over the basic elements of the architecture

Defining custom parameters

The elements can be added some custom parameters to adapt to special needs. These parameters can be exploited by user-defined algorithms.

```
(SET parameter_name parameter_value)
```

CODE EX. 2.19: The custom parameters syntax

Defining variable parameters

When prospecting the hardware definition space, the developer benefits from variable parameters allowing to perform loops in an automatic way. The produced results are collected. This is described in detail in section 5.

Setting the cost of the resources

It is possible to set the cost of every individual routing resource to influence the router's behavior. The default cost for a routing resource is 1. Adapting the cost of the resources depending on their range enables to favor the use of some resources (long lines, neighbor to neighbor connections, etc. . .).

This is done by setting the *baseCost* parameter.

```
( WIRE ( WIDTH size_of_wire (BASECOST number ) )
```

CODE EX. 2.20: The BASECOST syntax

Note that using this parameter only enables to modify the router's behavior. To change the placer's behavior, changing the cost function is required (see appendix A.6).

2.2 Using *Architecture Designer*

A convenient way to produce and use some architectural descriptions which are grammar compliant is to open the *Architecture Designer* tool. This tool facilitates common operations over textual descriptions such as compiling, saving/loading, producing the model, etc. . . and enables to invoke some other tools such as the user interface or the prospection automation tool.

2.2.1 Loading/Saving an architecture

There are two methods to recover an architecture contained in a file using the menu of *Architecture Designer*.

The file list interface (see fig. 2.7) enables to select a file from which to load a piece of Smalltalk code. This is the standard Smalltalk way to load (file in) code within the environment. The code to be loaded describes an architecture through a set of classes that have been previously generated (section 2.2.3)

The window of figure 2.8 enables to choose a file to load as a textual description of an architecture. Opening this window is accessible through the *File > load* menu.

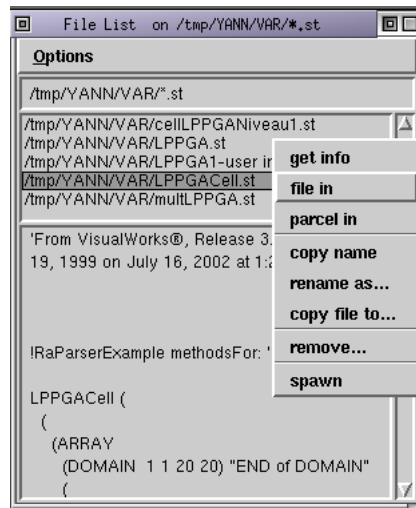


Figure 2.7: Loading an architecture description (1)

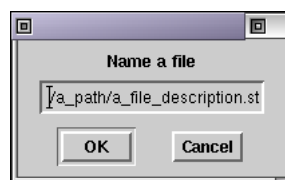


Figure 2.8: Loading an architecture description (2)

The user is asked to provide a path (a_path) and the name of the file of a textual description to be loaded ($a_file_description.st$). The files containing the description of an architecture have as an extension **.st**.

Saving your architecture is a menu operation in *File* $\hat{=}$ *save*. At this point, a filename must be provided ($a_file.st$).

2.2.2 Accessing/editing a description

Selecting a description is done through a drop-down list in *Architecture Designer* (figure 2.9).

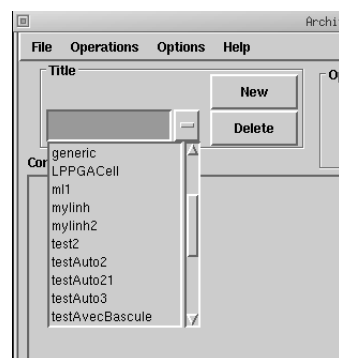


Figure 2.9: Selecting a description

It's possible to restrict the search by indicating to which category belongs the architecture (figure 2.10). This requires to choose *Options* and to validate the check box *category*. An new input field enables to select the target category.

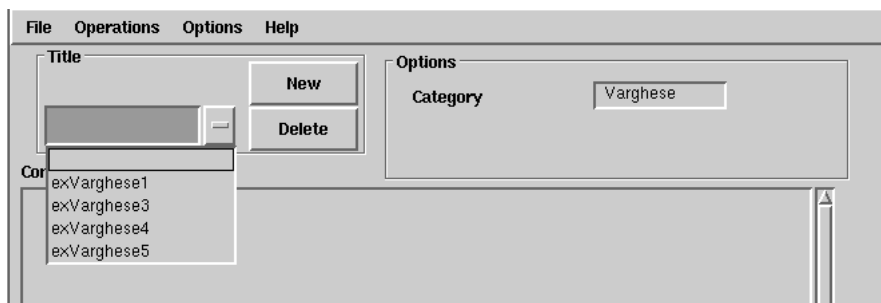


Figure 2.10: Choosing a category

2.2.3 Starting with a new description

Creating a new architecture goes through clicking on the button *New*. The name of this new architecture must be provided in the input field on the left. The architecture is described in the text editor. However the architecture is not recorded in the drop-down list until the description is compiled. Compilation starts when clicking the *compile* button.

2.2.4 Creating an architecture

The creation of an architecture defined in the text editor requires several stages. These stages are accessible through the buttons panel on the bottom of the window. Note that there is an inclusive relation between the operations from left to right (except the *open* button).

Compiling

The first stage, compilation, was already stated in a previous paragraph. It is represented by the button *compile* being on the interface, but can also be called by the menu *Operations* $\dot{\iota}$ *compiles*. This stage checks if the description of architecture is grammar compliant (see appendix B).

Generating Smalltalk classes

The creation of Smalltalk classes representing the architecture described in the text editor is done using the button *build* or with the menu *Operations* $\dot{\iota}$ *build*. The name of the classes as well as the categories conform to the directive provided within the textual description. This is useful for advanced programming.

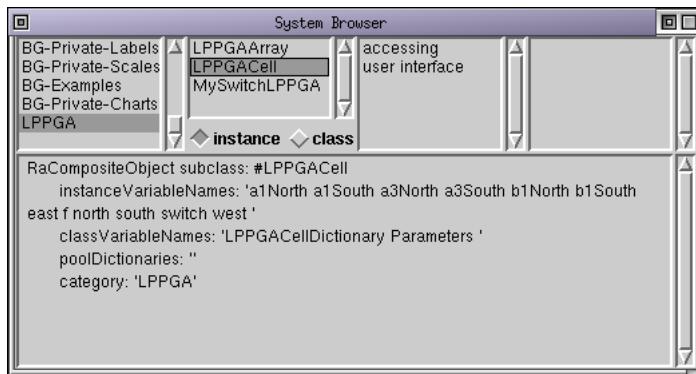


Figure 2.11: Code generated by *build*

The button *build* carries out compilation and classes' creation. Thus it's useless to click on *compile* before clicking on *build*. Options concerning the creation of the classes are available from the menu *Options*.

Architecture	Description
LPPGACell	Heterogeneous architecture
LPPGACell2	LPPGA Simplified architecture, including ugi
LPPGACell3	LPPGA Architecture with regular function IOs
LPPGACell4	LPPGA Architecture with IOBs
LPPGACell5	LPPGA Architecture with additional resources on borders

Table 2.1: Provided examples

In case the option *overwrite classes* is set then each use of *build* generates all the classes to be crushed. In a similar way, the option *overwrite representation* crushes the Draw methods (graphical representation).

It's necessary to pay attention to the use of these two options. Any changes made in the automatically generated code will be lost if one of both option is selected. Preserving a hand written user interface remains possible using the CUSTOM parameter (2.1.5)

User interface over the FPGA

For obtaining a representation of the architecture described in the text editor you can click on the button *UGI* or go in the menu *Operations & open UI*. This button includes the functionalities of the button *build*, it compiles description and builds the Smalltalk classes. In addition it calls the tool *FPGA Editor* to visualize the generated architecture.

It is possible to visualize the FPGA starting from the button *open*, however the classes representing this FPGA must be already generated.

2.2.5 Included examples

The code comes with a set of architectures definitions to enlighten the architectural grammar structure that have been presented in the previous section.

2.2.6 Managing dependencies between elements

By defining complex elements, some dependencies are introduced. As an example defining a channel made of several sub channels may bring architecture building failure if the sub elements are provided a name within the channel's container name space. This situation happens very after as soon as multiples naming are used.

Another example is the IOs definition of multiplexers that commonly refers to external resources. Of course, this requires those resources to be created before the multiplexer attempts to link its IOs to the resources.

These cases require to order the elements, under the designer responsibility. The default ordering is based on ascending name sorting, nevertheless, using the TREE VIEW, this order can be adapted by elements drag and drop.

The figure 2.12 illustrates that mechanism. Getting up this tree view is got by clicking on the TREE VIEW button of the Architecture designer.

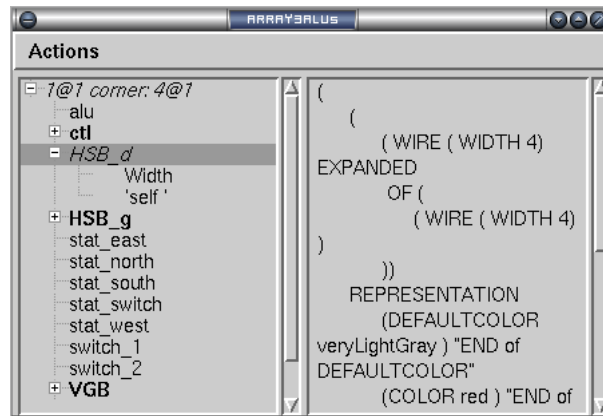


Figure 2.12: The tree representation of a composite

2.3 Generating documentation

In order to ease the architecture definition versionning, a documentation generator is included within MADEO-Bet. This tool, called GenDoc, requires a Unix like OS to run properly.

2.3.1 Principles

The main idea behind this is that writing documentation can be automated from the *UIArchitectureDesigner* tool. In fact, the documentation only provides a paper sheet with a tree view of the architecture, annotated by some icons to easily distinguish between the different kinds of elements, and by some comments the designer may add. The ?? appendix contains an example of report GenDoc can produce.

Motivation This automation is especially useful for joined works. Nevertheless, when letting an architecture description evolves, this allows to trace the evolution, as well as to set a version using the comment field (see figure 2.14).

Pre-requires The generator relies on \LaTeX . The report produces a \LaTeX file, with graphical commands in it. Some encapsulated postscript figures are needed so that one of the parameters is the \LaTeX directory.

2.3.2 Using MADEO-GENDOC

GenDoc is called from the *ArchitectureDesigner* tool by clicking on the right most button. This opens the 2.13 window. More complex parameter-

ization is possible by opening the 2.14 window. Note that default settings can be saved.

Actions The basic action is to generate the \LaTeX and the dvi files. A transcript window (figure 2.16) pops up after the job completes.

Getting some additional formats requires to select the adequate options. Several formats can be chosen as output : postscript, pdf, html. The pdf file is produced by the *ps2pdf* command, the html is made by calling *latex2html*. the html description (figure 2.17) includes hyper links that ease the navigation through the architecture elements.

The other actions are viewing the postscript, generating the pdf, viewing the pdf.

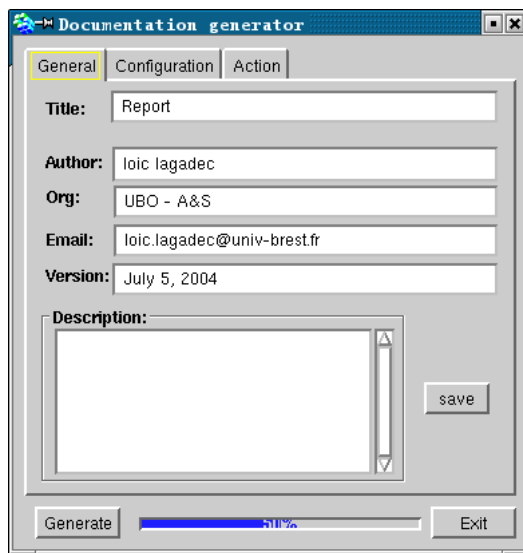


Figure 2.13: The GenDoc user interface

2.4 VHDL generation

MADEO-bet allows to bring up a set of tools to program any architecture before any hardware realization. This ensure good properties such as early architecture evaluation, that lets SOC use of reconfigurable make sense. Regarding the classical design flow starting with writing a VHDL description of the architecture before synthesizing it and finally re-writing tools to tune them in order to apply on the new architecture, Made-bet offers an alternative flow making the tools a starting point.

Nevertheless, after the designer has defined its target hardware, as an example by taking advantage of the hardware prospection capabilities of MADEO-BET, the VHDL writing step still remain to be done.

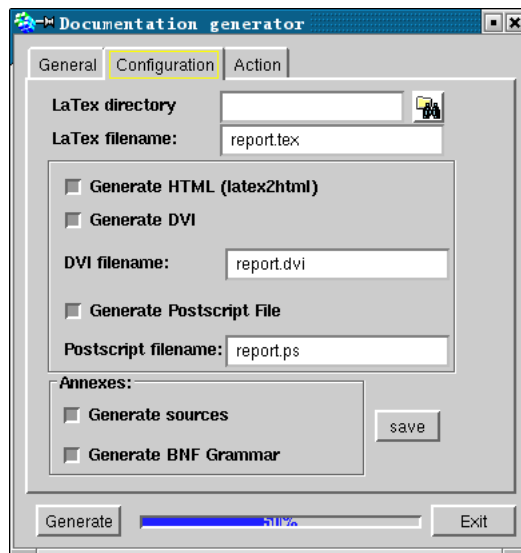


Figure 2.14: The GenDoc parameters

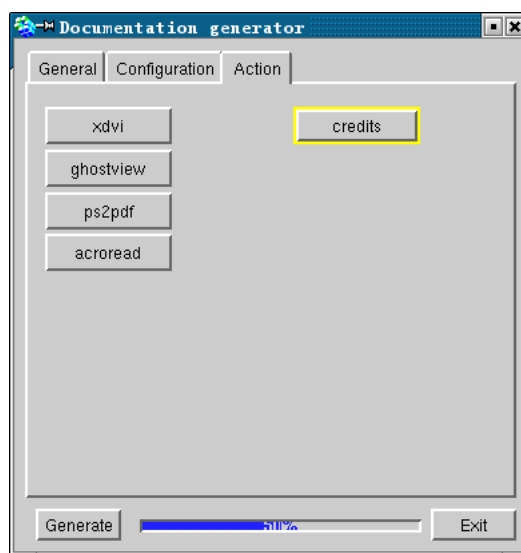


Figure 2.15: The GenDoc actions

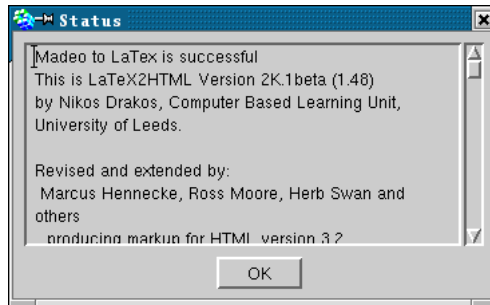


Figure 2.16: The GenDoc transcript

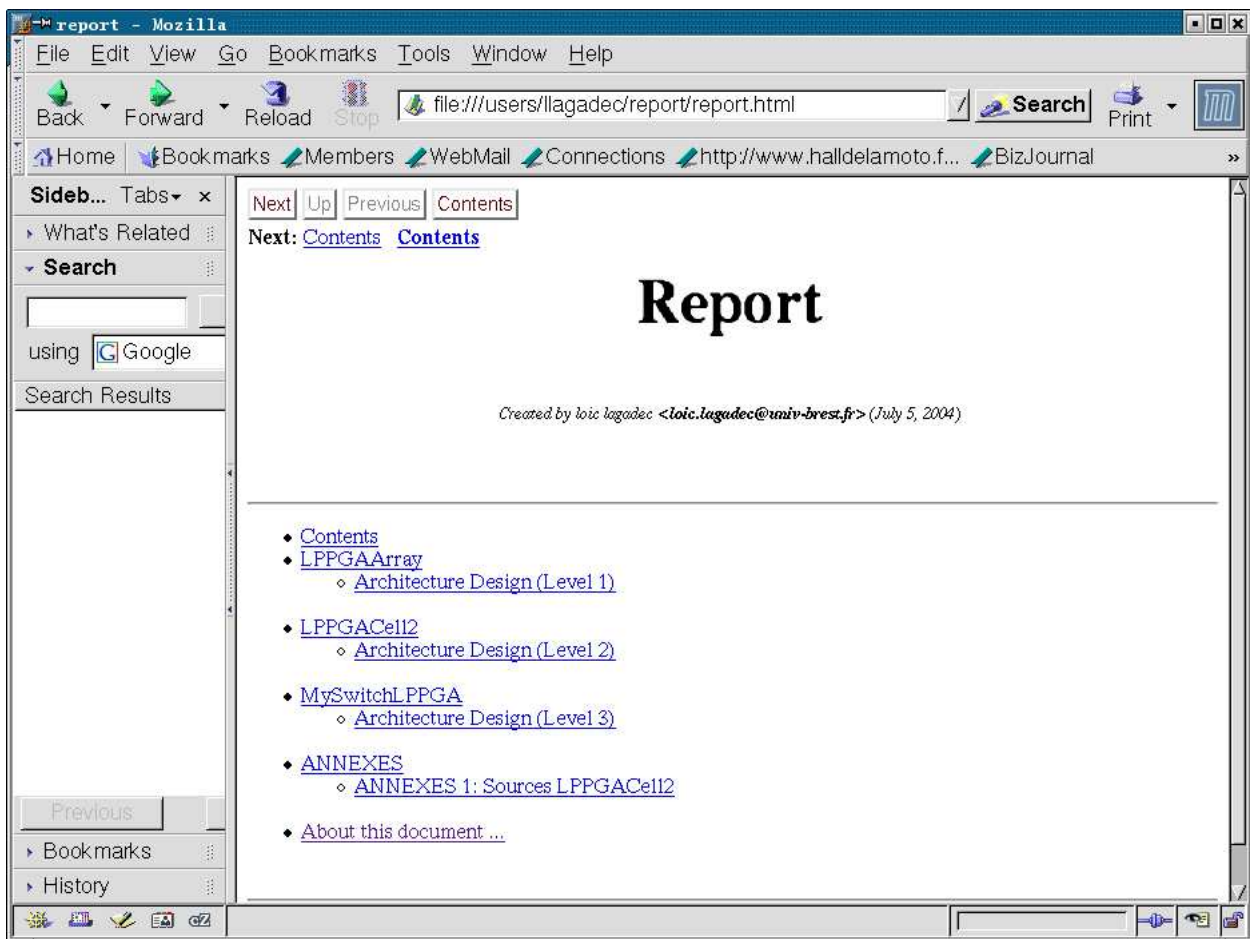


Figure 2.17: The GenDoc html output

2.4.1 Principles

To shortcut this step within the design flow a current work aims to provide automatic VHDL code production based on the grammar describing the hardware.

Motivation The goal is not to compete with "general purpose FPGAs" vendors such as Xilinx but rather to get an output enabling physical synthesis of custom dedicated FPGAs in a SOC scope.

This offer an after synthesis accurate feed-back to tune the algorithm parameters in one hand, and suppress the manual intervention of the designer in the other hand.

Pre-requires A VHDL model has been developed that encapsulates not only the syntax of VHDL but also supports the hierarchical approach, and modular decomposition of the code.

The set of classes is made of three types of objects:

- the classes that match the basic constructs of the language (entity, if then else, port, etc...),
- the classes that offer a support for recurrent constructs (luts, switches, etc...),
- some classes that handle internal constructs (parenthesis, indexes, etc...).

All the classes own a modular generation of textual representation, and are provided with examples¹.

The key issue is to validate an right instantiation of this model from the MADEO-BET model. As the MADEO-BET model is made of atomic objects and compositions mechanisms, the VHDL model supports atomic constructs and complex combinations of objects through a mechanism of port mapping.

2.4.2 Using *VHDL Producer*

This interface is not working at this time.

¹as class methods

Chapter 3

Programming an architecture

3.1 Introduction

This section describes what 'programming a reconfigurable architecture' means. As well, it highlights the difficulty to route FPGAs, as compared to ASICs.

3.2 Towards new needs

This section describes the new needs in term of flexibility (to different families as well as to defaulted instances of an architecture), short development time and algorithmic quality (modularity / reuse), support for upper layers and inter-operability.

This section argues as well that the tools must be used as early as possible in the design process of a new architecture, as it's the only way to get an accurate feed back after tuning finely the architecture description; and that tools must be decoupled from the target architecture.

3.2.1 Algorithms

This section introduces some well known algorithms and select some of them that have been picked up to be implemented within MADEO.

Chapter 4

Tools

4.1 Using *FPGA Editor*

FPGA Editor (see fig. 4.1) is a tool which makes it possible to visualize and to handle an architecture created with *Architecture Designer* .

The interface is made of three parts: the first one corresponds to the menu, the second one is a view of the FPGA and the third one is command line zone.

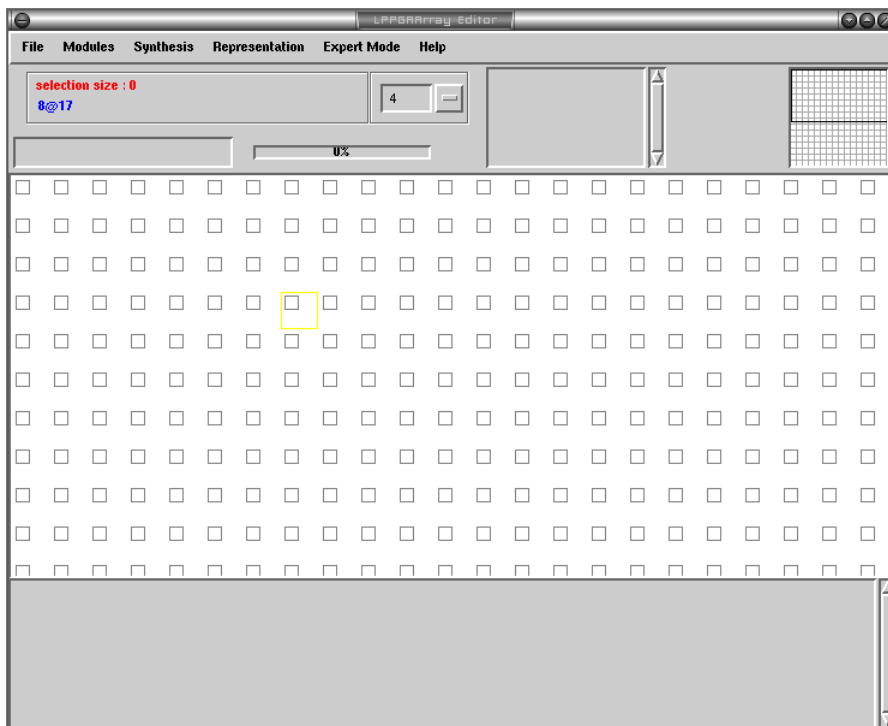


Figure 4.1: The interface of *FPGA Editor*

4.1.1 Application description

This describes the several levels of application description. The application is described over a set of objects which support the same API but own different levels of abstraction. All these objects are SCNodes; they encapsulate a computational description and own IO ports which handle dependencies in order to support change propagation over the values they retain.

The connectivity between nodes is relevant to the IO ports' pins.

What differentiates between the types of nodes is the kind of computational description they carry: BLIF, EDIF, Smalltalk block, etc. . . . As well there are some hierarchical constructs, as illustrated by the 4.5 example.

The figure 4.2 illustrates this formalism.

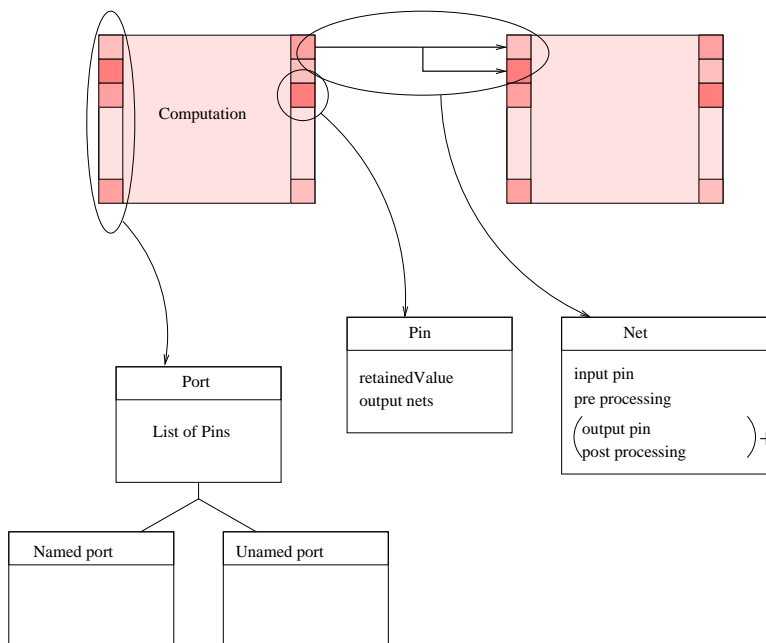


Figure 4.2: The application description formalism

4.1.2 Selecting BLIF/EDIF files

The *Modules > BLIF > Place and Route has BLIF description* menu (see fig. 4.3) enables to choose a file to be placed/routed on the architecture.

The appearing window lets the designer free to choose the file to place and route, to regulate the orientation of the routing and to give a position to start place and route.

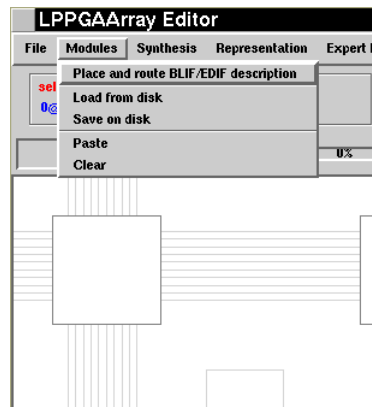


Figure 4.3: Selecting a BLIF/EDIF file from the menu

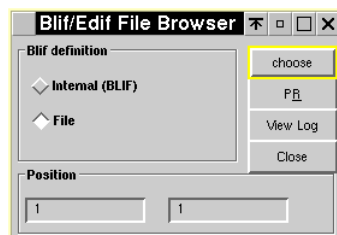


Figure 4.4: BLIF/EDIF File Browser

To select a file the user either types the name of the file with the complete path in the input field, or clicks on the button *choose*. Clicking on *choose* makes a *file list* to open (fig. 4.5).

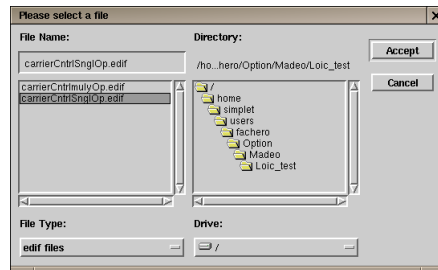


Figure 4.5: selecting a BLIF/EDIF file

4.1.3 Placing and routing a BLIF/EDIF file

Once parameterized the *BLIF file browser*, clicking on the button *PER* places and routes the modules. The name of the new module appears in the drop-down list of the menu (fig 4.6).

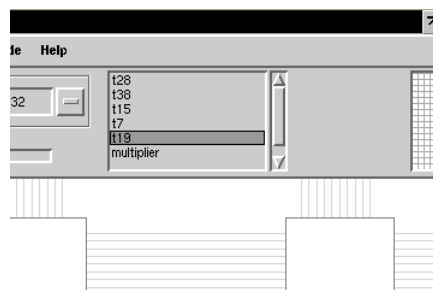


Figure 4.6: The list of available modules

Note that the EDIF format supports location constraints for the functions. These constraints are added using the FPOS property field. The location is relative to the bottom left corner of the module.

4.1.4 Pasting the module

The selected cell acts as the bottom left anchor when pasting a module. This cell is yellow bordered and is selected using the mouse. Pasting the module is done by choosing the correct option in the menu associated with the list of modules (figure 4.7). The figure 4.8 shows the result of pasting a module. Note that this action does not trigger a new place and route process but relies on a previous place and route, that produced the module.

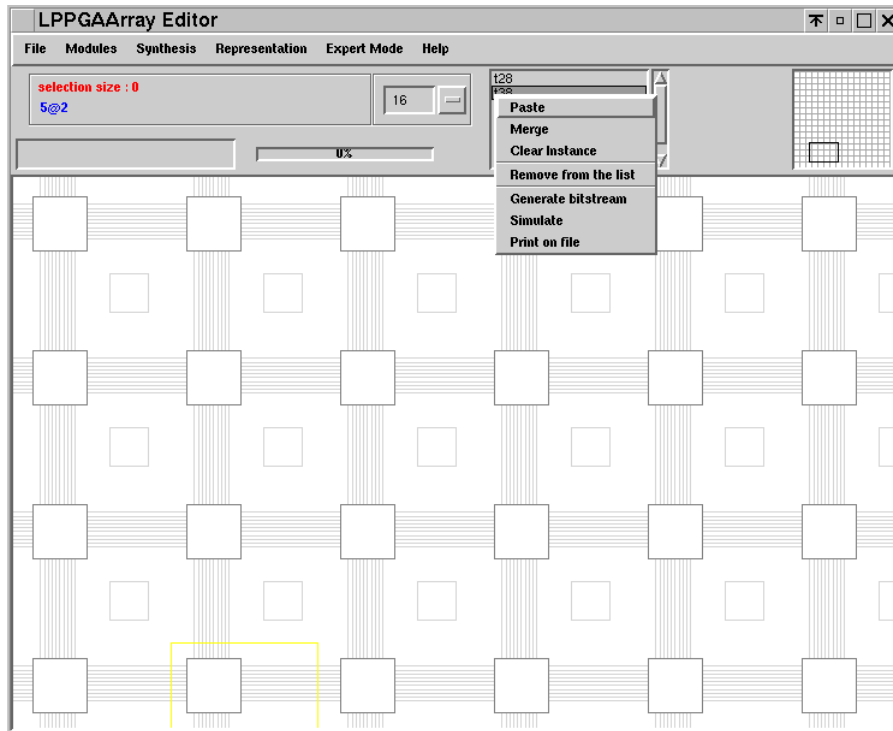


Figure 4.7: Pasting the module on the FPGA

If the display is too small, the enlarging of the FPGA can be changed by choosing a factor in the small drop-down list. This is illustrated through the figure 4.9.

Another representation option (depending on the grammar definition of the architecture) allows to draw either all the resources, which is useful to estimate the routing congestion, or only the used resources, which makes the drawing simpler. The figure 4.10 illustrates how to swap between the two modes.

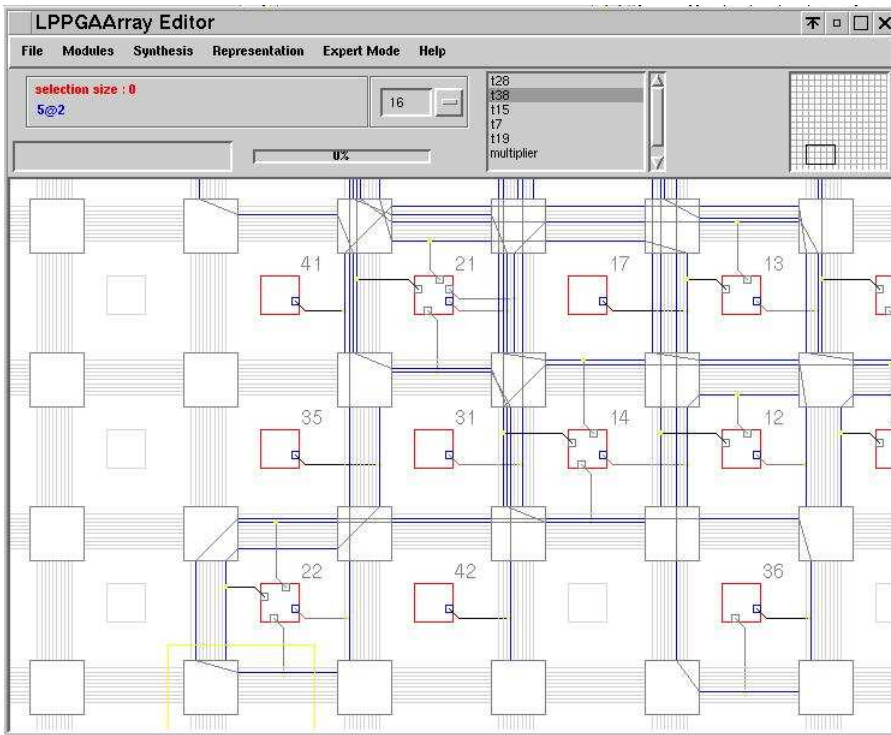


Figure 4.8: After pasting the module on the FPGA

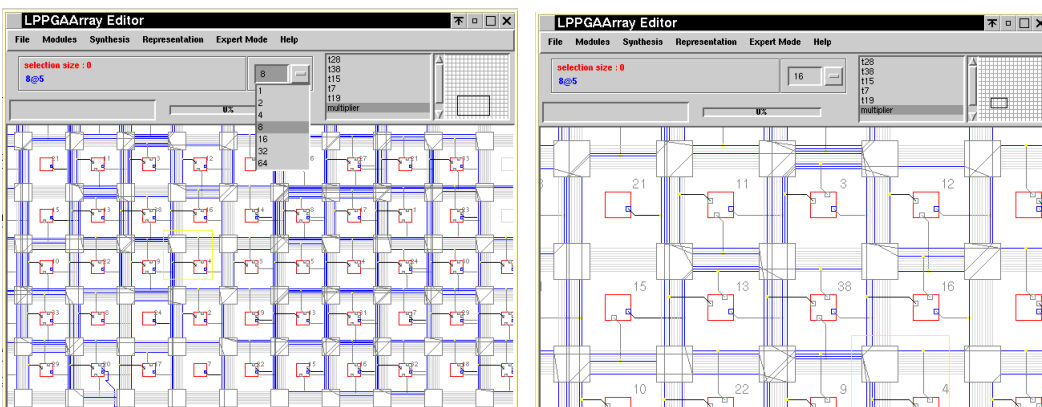


Figure 4.9: Selecting the zoom factor

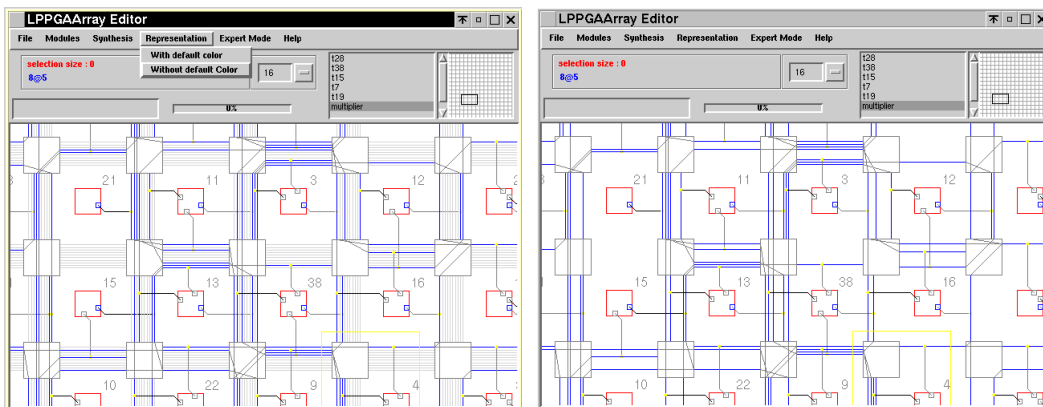


Figure 4.10: Representing either all or only used resources

4.1.5 Simulating the circuit

Once placed and routed, a circuit can be simulated as illustrated in the figure 4.11 by using the *simulate* option of the module's menu. The simulation process does not rely on the P&R information but limits itself to analyzing the netlist. Both simple and hierarchical descriptions can be simulated.

As an example the figure 4.11 shows a hierarchical circuit which submodules' bounding boxes are drawn, and on the right the result of simulation. Either the global circuit (front window) and a one of its submodules (back window) are simulated.

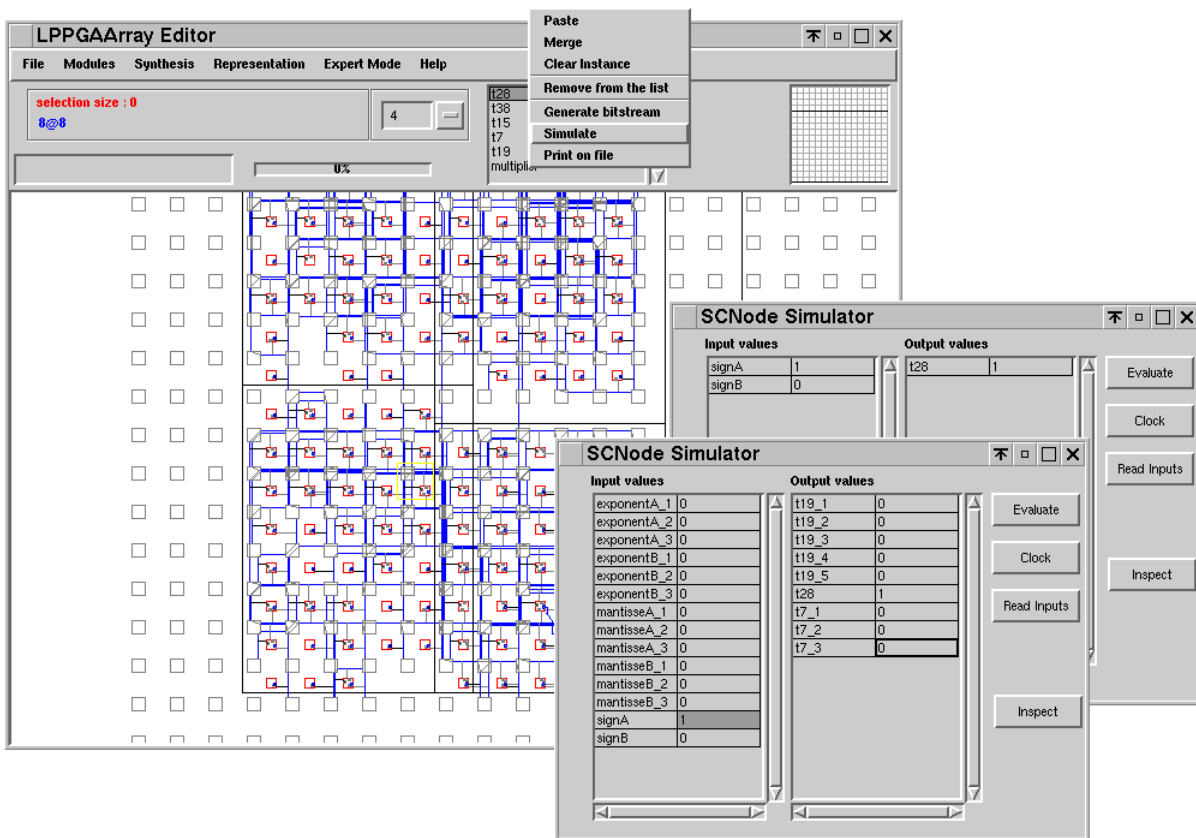


Figure 4.11: The simulation of P&R circuits

4.1.6 Using the window commands

The command line area of the window enables complex manipulations over the architecture, such as in depth inspection of the resources and debugging.

here is a list of simple examples of useful commands to recover information on the FPGA follow:

self model

Inspecting this code (button medium of the mouse and *inspect*) returns the Smalltalk object representing FPGA circuit.

self model modules values

Inspecting this code returns the list of circuits implemented within the FPGA (each of which owns several instances).

(self model modules values at: i) benchmark

Inspecting this code returns the place&route benchmark of the i^{th} module.

((self model modules values at: i) instances) at: #instance1**elements**

Inspecting this code returns the placement information of the instance #instance1 of the i^{th} module

((self model modules values at: i) instances) at: #instance1**routes asOrderedCollection**

Inspecting this code returns the routing information of the instance #instance1 of the i^{th} module

4.2 Floor planning

The floor planning is based on [12]. A floorplan is described as an assembly of rectangles and a set of constraints regarding the relative locations of the rectangles. Four basis operation apply on the floorplan :

rotate transposes the extent of the rectangular bounding box of a module

swap exchanges two modules

reverse reverses the relation between two modules (eg. $i \perp j \mapsto j \perp i$ where \perp is either *left* or *above* and i, j are two modules)

move modifies the type of relation (eg. *left* \mapsto *above*)

The floorplanner performs an annealing schedule over the floorplan, using these four operations. The cost function takes into account the global bounding box of the assembly, the part of this bounding box which does not match the constraint bounding box requirements and the interconnection cost (see code 4.1).

The penalty that applies to the bounding box slowly increases as the temperature decreases (see code 4.2). The default bounding box is computed as the default bounding box before optimization, scaled by a factor of 0.8, unless a bounding box is provided.

As the algorithm is not deterministic, the result can be sub optimal as illustrated by figures 4.12 (initial positions), 4.13 (optimal floorplan) and 4.14 (sub optimal floorplan).

After the annealing process completes, the bounding box associated with the nodes is taken as a constraint during the P&R stage.

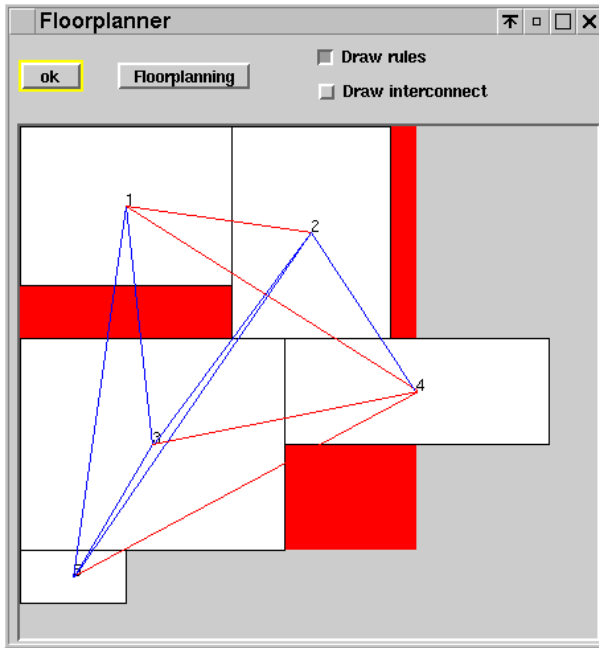


Figure 4.12: A circuit to be floorplanned

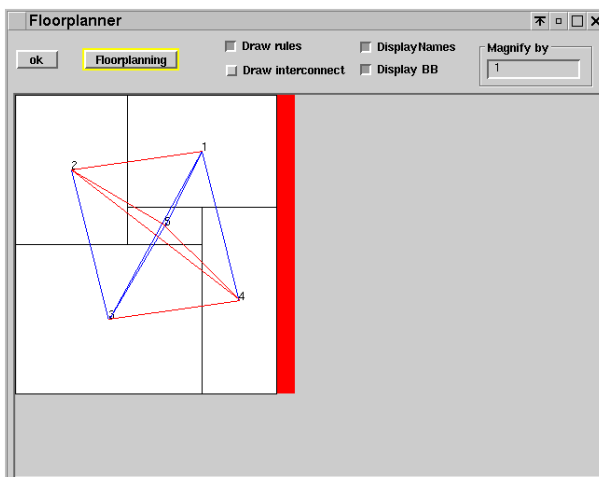


Figure 4.13: The expected result

```

cost
  | tmp |
  self compact.
  tmp := self boundingBox.
  ^self maxBoundingBox isNil
    ifTrue: [self boundingBox area]
    ifFalse:
      [tmp := (self boundingBox areasOutside: self maxBoundingBox)
        inject: 0 into: [:a :b | a + b area].
        self boundingBox area - tmp + (self penalty * tmp) * 100 * self
interconnectCost]

```

CODE EX. 4.1: The floorplanner cost function

Examples

Two examples referred as 4.3 and 4.4 are provided with the code.

The result of the 4.3 example is illustrated by the figure 4.2.

```

randomize: aNumberBetweenZeroAndOne
  | tmp |
  tmp := self copy.
  tmp penalty: (1 - unNumberBetweenZeroAndOne * self maxPenalty)
ceiling.
  ^ (tmp basicRandomize: aNumberBetweenZeroAndOne)
    ifTrue: [tmp]
    ifFalse: [self]

```

CODE EX. 4.2: Setting the penalty

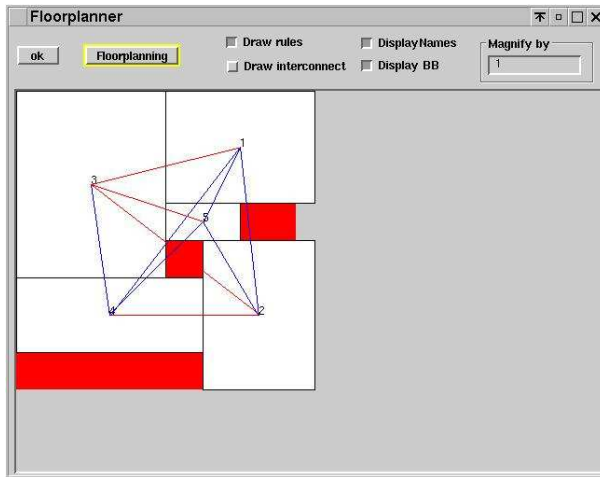


Figure 4.14: A non optimal result

```
self model exampleSamos
```

CODE EX. 4.3: First example

```
self model exampleGuild
```

CODE EX. 4.4: Second example

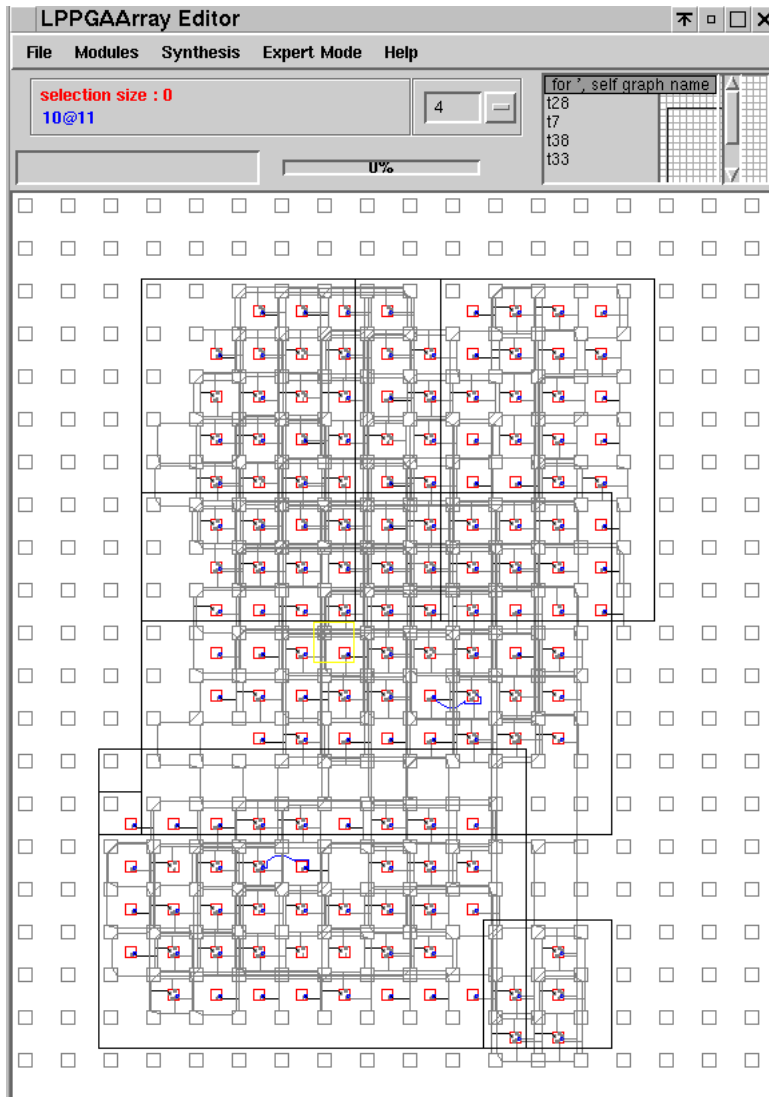


Figure 4.15: The result of the example 4.3

4.3 Drawing regular circuits

4.3.1 Introduction

Drawing regular circuit is useful when designing operators or in a more general fashion, highly parallel circuits. Such circuits can be described at an high level of abstraction such as recursive equation [11] but require to be represented with a specific semantic and specific annotations to let the tools operate on it. An example of such information is provided by the Alpha language, a subset of which, called AlphaHard, express the low level information that is needed when drawing the circuit.

The object carrying such an information in MADEO are called `SCCompositeNode`. Three types of `SCCompositeNode` exist:

`SCCompositeNode` is the more general level, which the floorplanner uses

`SCComposite1DNode` describes linear networks

`SCComposite2DNode` describes arrayed networks

The figure 4.16 illustrates the last two cases.

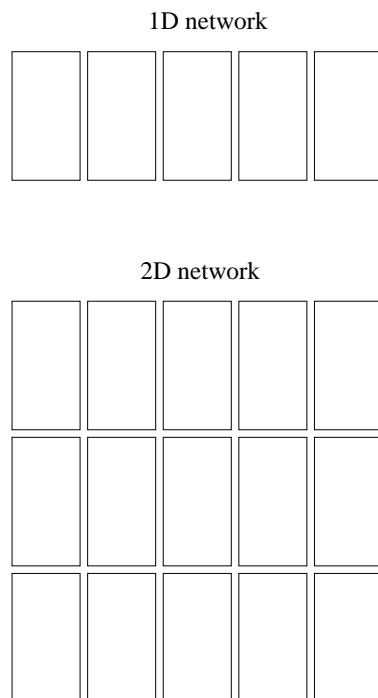


Figure 4.16: Both linear and arrayed network of nodes are supported

4.3.2 Structural representation

In addition to the structural description of the node (free, 1D, 2D), the connections between elements can be of several types:

Classical Classical interconnection links a source resource to a target one

Regular Regular interconnections are replicated in the same way that the internal nodes are replicated

With accessors interconnections with accessors allows to add a data processing to a base connection (such as accessing a given bit of)

The example 4.5 illustrates the practical way to define a regular circuit. In this example, the IOs ports are first described, and the `SCComposite1DNode` is built by replicating the internal nodes (some `SCBlif` nodes that perform a full adder). Once this is done, regular connections link the most significant bit output (the carry) to the *c* input of the neighbor node for every node. In the same time the less significant bit is routed to the output port. The last most significant bit makes the first (most significant) bit of the result output. The IOs connections are built using classical interconnections whereas the internal connections are regular.

4.3.3 Compatibility

All the representations of the circuit, which can be either structural, based on `SCCompositeNode`, or behavioral, based on `SCBlockNode`, or logical based on `SCBlifNode`, are functionally equivalent (see 4.1.1). This point is really important as this guarantees they can be exchanged one to the other depending on the contextual needs for more simplicity or, on the other hand, for more in depth description of the circuit.

All of these descriptions respect the same manipulation API, so that the real kind of description the designer uses never counts, except on demand.

A set of translators for converting object into another formalism is accessible to the designer (`asBlifNode`, `asSCCompositeNode`, etc...). This set of representations and associated translators is extendable if required, assuming the based API is preserved.

```

serialAdder: aSize
  "(self serialAdder:3) open"

  | tmp |
  tmp := self
        iPort: (SCNamedPort with: (#(#a #b)
                                collect: [:inputName | inputName ->
(SCUnNamedPort newInput: aSize)]))
        oPort: (SCUnNamedPort newOutput: aSize + 1)
        component: SCBlifNode fullAdder
        indexes: (Array new: aSize).

  tmp
    regularConnect: 'h' "most significant bit"
    to: 'c' "carry input"
    of: -1. "backward neighbour"
  tmp components keysAndValuesDo:
    [:index :value |
      (tmp iPort at: #a) "presenting the input "
        connect: index
        to: 'a'
        of: value.
      (tmp iPort at: #b) "presenting the input "
        connect: index
        to: 'b'
        of: value.
      value oPort "1 ... aSize bit of output"
        connect: 'l'
        to: index + 1
        of: tmp oPort].
  tmp components first "carry"
    connect: 'h'
    to: 1
    of: tmp oPort.
  ^tmp

```

CODE EX. 4.5: A serial adder

Chapter 5

Architectural prospection

5.1 Introduction

When designing a new architecture the architects need to evaluate several architectural alternatives. This requires the test architectures to share a common evaluation tool; and to make the prospection process fully automatic.

This section describes this stage, allowing to tune an architecture by fixing its resources; performing place/route and collecting results. This tool is called `UIProspection`.

Among architectural criteria, the logical grain of functions, the width of channel as well as the connections between elements can be modified to produce a new variant inside the architectural family defined through a grammar description.

The results can be collected under different criteria: routing cost, CPU time or bounding box. As some algorithms are non deterministic, the number of runs can be fixed to smooth raw results.

5.2 Tools

5.2.1 Starting `UIProspection`

`UIProspection` launches out from the Architecture Designer in the operation menu (figure 5.1).

The figure 5.2 represents a snapshot of the `UIProspection` interface.

5.2.2 Analyzing an architecture

The architecture to be prospected is the current architecture of the Architecture Designer tool. The architecture can be changed (figure 5.3) without leaving the prospection tool open.

The architecture's grammatical description can appear by clicking on the button viewer. This is useful to check the changes over the description resulting from a prospection process.

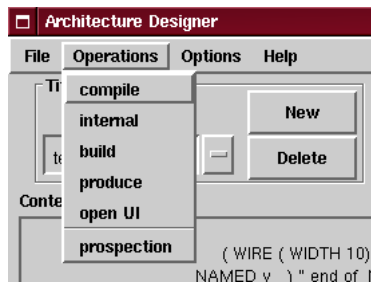


Figure 5.1: Launching UIProspection

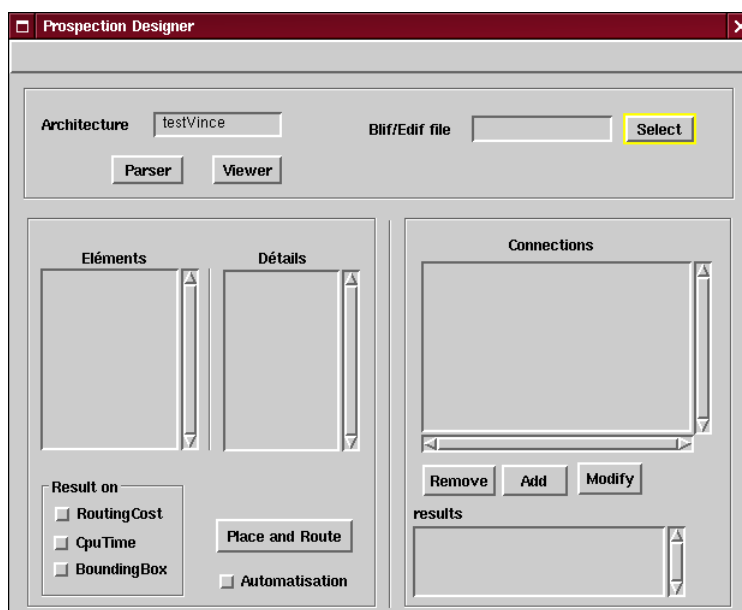


Figure 5.2: Prospection Designer

The elements that can be changed for prospection are collected after the description of the architecture is parsed (by clicking on the parser button) as illustrated by figure 5.4¹.

5.2.3 Selecting a BLIF/EDIF file

The prospection tool allows to place and route BLIF or EDIF file. The selection of the file is done by clicking on the select button (figure 4.5).

Performing a single place and route is done by clicking on the place and route button. In this case, for every MV-variable, only the first value is considered. After the place and route completes, the results appear on the window (figure 5.5).

The automation of several place and route is done by selecting the au-

¹Wires are characterized by 'name:width'

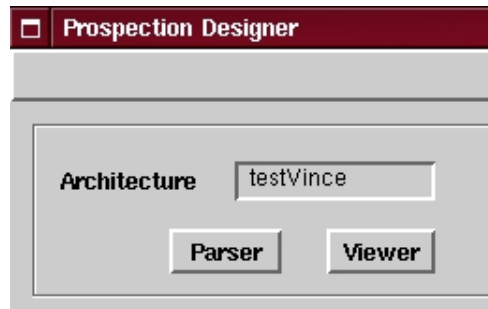


Figure 5.3: Viewer button

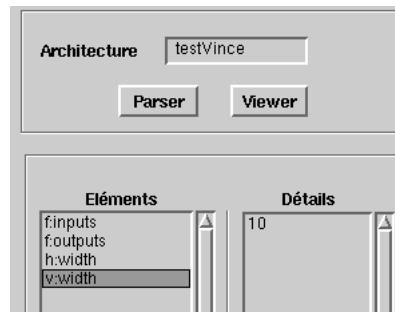


Figure 5.4: Detailed view of the selected element

tomation combo box before clicking on the place and route button (figure 5.6).

If the architecture does not contain any MV-variable, then no automation performs. In the contrary case, a new window called UIElements pops up.

5.2.4 Modifying the value of an element

The value of an element can be modified by double clicking on its value. A dialog box appears in order to capture the new value (figure 5.7).

Changing the value of an element goes through defining a variable, taking as value a collection of values (an interval or an array of values) as described in table 5.1.

Type of collection	Code
an interval	x to: y with x ζ y x to: y by: z
an array	#(x y z)

Table 5.1: Definition of an interval vs definition of an array

This kind of variable (referred as MV-variables) associated with a range of values can be expressed using the grammar. As an example, evaluating the

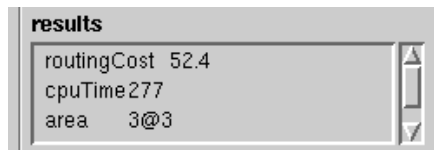


Figure 5.5: Result of a single place and route

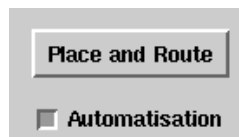


Figure 5.6: Automation of prospection

effect of the width of a wire brings about some changes in the grammatical description as illustrated by the example 5.1.

```
(WIRE (WIDTH 7)) -i (WIRE (WIDTH (VALUE name 'x to: y')))
```

CODE EX. 5.1: Defining MV-variables

Replacing a literal value such as an integer by a MV-variable implies to create such a variable; its name is automatically generated.

In order to force joined variation, several objects can share the same variable. This happens when writing a string in the dialog box (figure 5.7). This string refers to the name of a variable; that has been previously defined. The table 5.2 illustrates the syntax for referencing a variable.

5.2.5 Collecting the results

The result of a place and route process can be analyzed according to three criteria² : cost of routing, CPU time, and bounding box (figure 5.8).

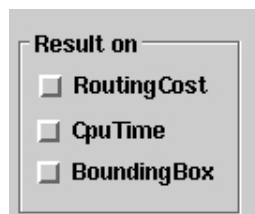


Figure 5.8: Selecting the type of result

²Some new criteria will be added later on

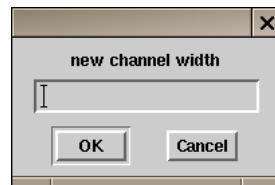


Figure 5.7: Keyboarding a new value

Definition	(WIRE (WIDTH (VALUE toto 'x to: y')))
Reference	(WIRE (WIDTH (VALUE toto)))

Table 5.2: Creation vs reuse of variable

5.2.6 Performing the prospection

Once the architecture has been analyzed and the variable parameters isolated, the designer is asked to select the parameters which impact must be quantified, with regards to the chosen criterion. When using several variable parameters, the way the loops over parameters are nested is within the competence of the designer.

Setting the parameters of the prospection is done using the UIElement windows (figure 5.9)

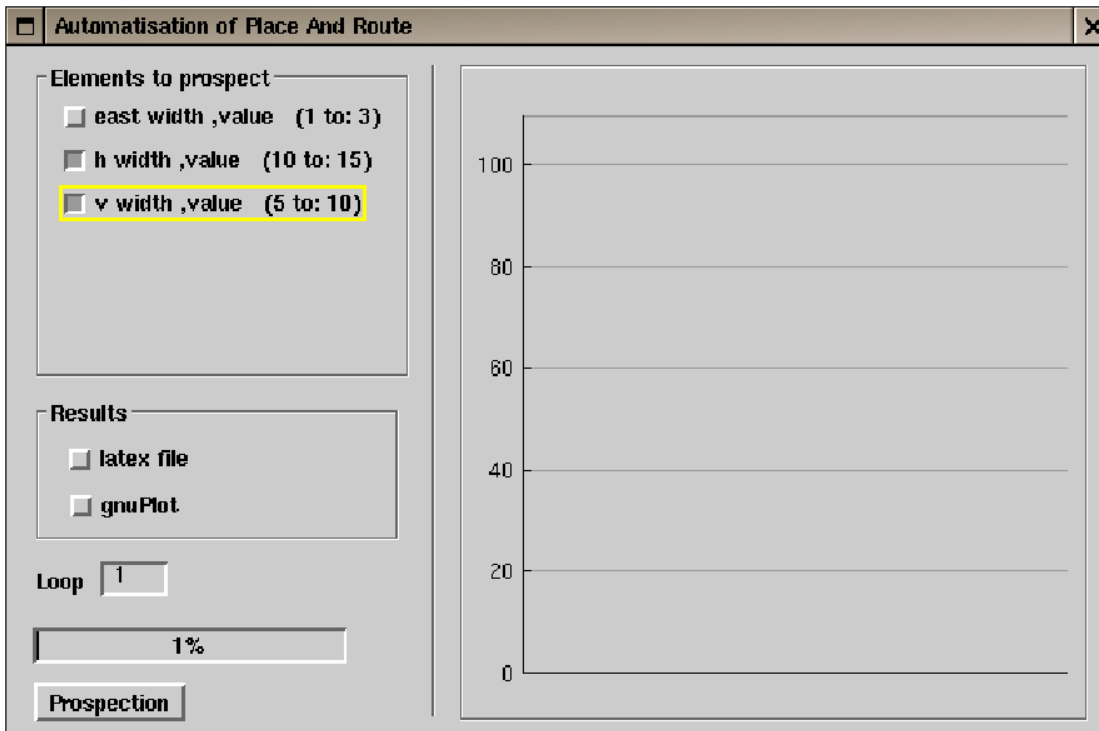


Figure 5.9: The UIElements interface

Every variable parameter is associated a combo box, allowing either to take it into account or to ignore it (figure 5.10). Note that combo boxes refer to MV-variables, not to architectural elements using these variables; a shared variable only appears once.

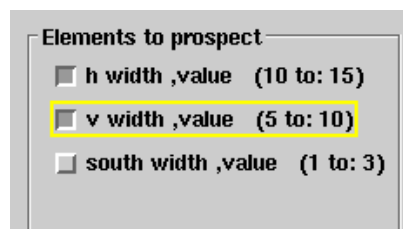


Figure 5.10: The elements to be prospected

The designer is free to define number of runs to be performed (figure 5.11). This should be a tradeoff between speed and reliability of the results, as the collected raw results are smoothed (avg).

The results can be produced in the form of a latex documentation (figure 5.12, appendix A.8) or a graphic. The latex report contains an history of each loop and an average.

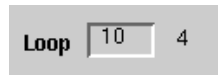


Figure 5.11: the number of runs and the current loop

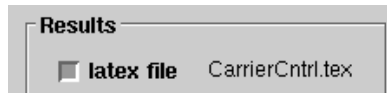


Figure 5.12: Latex output

Part II

MADEO FET

Chapter 6

introduction

Applications for fine grain reconfigurable architectures can be specialized without compromise, and they should be optimized in terms of space and performance. In our view, too much emphasis is placed on the local performance of standard arithmetic units in the synthesis tools and also in the specification languages.

A first consequence of this advantage is the restricted range of basic types coming from the capabilities of ALU/FPU's or memory address mechanisms. Control structures strictly oriented toward sequentiality are another aspect that can be criticized. As an example, programming for multimedia processor accelerators remains procedural in spite of all the experience available from the domain of data parallel languages. Hardware description languages have rich descriptive capabilities, however the necessity to use libraries has led the language designers to restrict their primitives to a level similar to C.

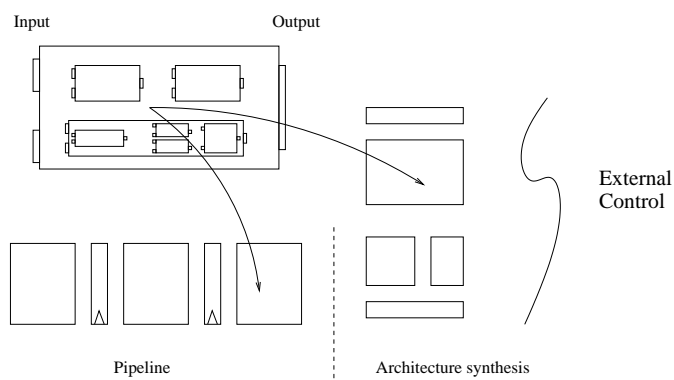


Figure 6.1: The modules can be either flat or hierarchical; the modules can be composed in order to produce pipelines or can be instantiated during architecture synthesis.

Our aim is to produce a more flexible specification level with direct and efficient coupling to logic. This implies allowing easy creation of specific arithmetics representing the algorithm needs, letting the compilers automatically tune data width, and modeling computations based on well un-

derstood object classes. The expected effect is an easy production of dedicated support for processes that need a high level of availability, or could waste processor resources in an integrated system. To reach this goal, we use specifications with symbolic and functional characteristics, jointly with separate definition of data on which the program is to operate.

Sequential computations can be structured in various ways by splitting programs on register transfers, either explicitly in the case of an architecture description, or implicitly during the compilation. Figure 6.1 shows these two aspects, with a circuit module assembled in a pipeline and in a data-path. In the case of simple control loops or state machines, high level variables can be used to retain the initial state with known values, the compiler retrieving progressively the other states by enumeration [10]. Figure 6.2 shows a diagram where registers are provided to hold state values associated to high level variables that could be instance variables in an object.

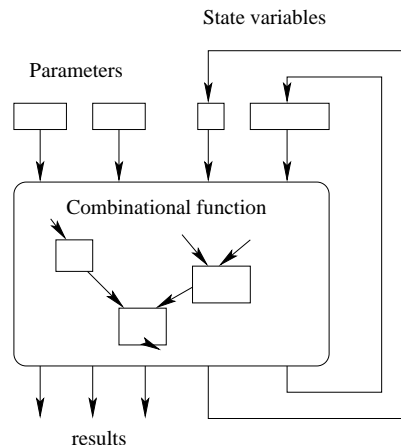


Figure 6.2: State machines can be obtained by methods operating on private variables having known initial values.

At this stage, we will consider the case of methods without side effect, operating on a set of objects. For sake of simplicity we will rename these methods '*functions*', and the set of objects, '*values*'. Interaction with external variables is not discussed there. The input language is Smalltalk-80, variant VisualWorks, also used to build the tools and to describe the application architectures.

6.1 Execution model

The execution model targeted by the compiler is currently a high level replication of LUT-based FPGAs. We define a '*program*' as a function that needs to be executed on a set of input values. Thus the notion of *program* groups at once the algorithm and the data description. Our *program* can be embedded in higher level computations of various kind, implying variables

or memories. Data descriptions are inferred from these levels. The resulting circuit is highly dependent from the data it is intended to process.

An execution is the traversal of a hierarchical network of lookup tables in which values are forwarded. A value change in the input of a table implies a possible change in its output that in turn induces other changes downstream. These networks reflect the effective function structure at the procedure call grain and they have a strong algorithmic meaning. Among the different possibilities offered for practical execution, there are cascaded hash table accesses, and use of general purpose arithmetic units where they are detected to fit.

Translation to FPGAs need binary representation for objects. This is achieved in two ways, by using a specific encoding known to be efficient, or by exchanging object *values* appearing in the input and output for *indexes* in the enumeration of values. Figure 6.3 shows a fan-in case with an aggregation of indexes in the input of function $h()$. Basically the low level representation of a node such as $h()$ is a *Programmable Logic Array* (PLA) having in its input the Cartesian product of the set of incoming indexes ($fout \times gout$), and in its output the set of indexes for downstream.

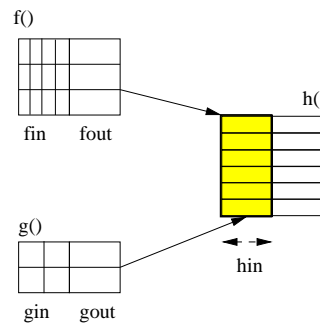


Figure 6.3: Fan-in from 2 nodes with $Card(fout \times gout) < Card(fin) \times Card(gin)$.

There are some important results or observations from this exchange:

1. data paths inside the network do not depend anymore on data width but on *the number of different values present on the edges*.
2. depending on the interfacing requirements, it will be needed to insert nodes in the input and output of the network to handle the exchanges between values and indexes.
3. logic synthesis tool capabilities are limited to medium sized problems. To allow compilation to FPGAs, algorithms must *decrease the number of values* down to nodes that can be easily handled by the bottom layer (SIS partitioning for LUT-n). Today, this grain is similar to algorithms coded for 8-bit microprocessors.

4. *decreasing the number of values* is the natural way in which functions operates, since the size of a Cartesian product on a function input values is the maximum number of values produced in the output. The number of values carried by edges is decreasing either in the hierarchy structure or in a graph flow. There is no possible divergence and the efficiency of an algorithm can be stated to be its ability to quickly decrease the data amplitude on which the logic complexity depends.

6.2 Type system

Language types appear to the programmers as annotations for checking code consistency and binding to architecture resources. The type system we are using does not restrict programming to this kind of binding. It is only intended to specify any possible set of values appearing in the program input or inside the computation network. In the object environment, it is supported by a set of classes supporting operations.

Implicit or explicit collections of values are denoted by intervals or sets. *Class-based types* are associated either to classes having a finite number of instances (booleans, bytes, small integers), or to user defined new functionalities, including arithmetics. *Unions* are resulting from operations on the two previous types.

Chapter 7

design flow and optimizations

7.1 design flow

The design process (see figure 7.1) starts with a code description of the task to be implemented. This description appears as a ST80 method. Modularity is preserved as the self pseudo-variable exists, what enables to describes a task by composing several methods of the same class.

The MADEO FET flow ends by producing a custom application description that MADEO BET handles as an input. This description is then placed and routed, floorplanned, etc ... We assume that the architecture description used within MADEO BET will drive some of the optimization steps; as an example by forcing some of the nodes to be merged together depending on the architecture grain (LUTs, memories, etc ...).

Restrictions: Instance variables are not supported at this time; class variables can be used as fixed parameters (during a design process the values are fixed).

By compiling this method, a static graph of operator is produced. At this stage, every node matches a Smalltalk message, and the vertices represent data flows.

This graph is re-organized regarding some optimizations (see section) before an equivalent logic representation is built.

Based on this RTL description, a (possibly hierarchical) module is placed and routed using the MADEO-bet generic back end tool set.

Definitions:

- a type is a set of possible values
- a variable is associated to a type
- a variable knows its producer (a node) and its consumers (some nodes)
- a producer of an input variable for a node is called parent node
- a consumer of an output variable for a node is called children node

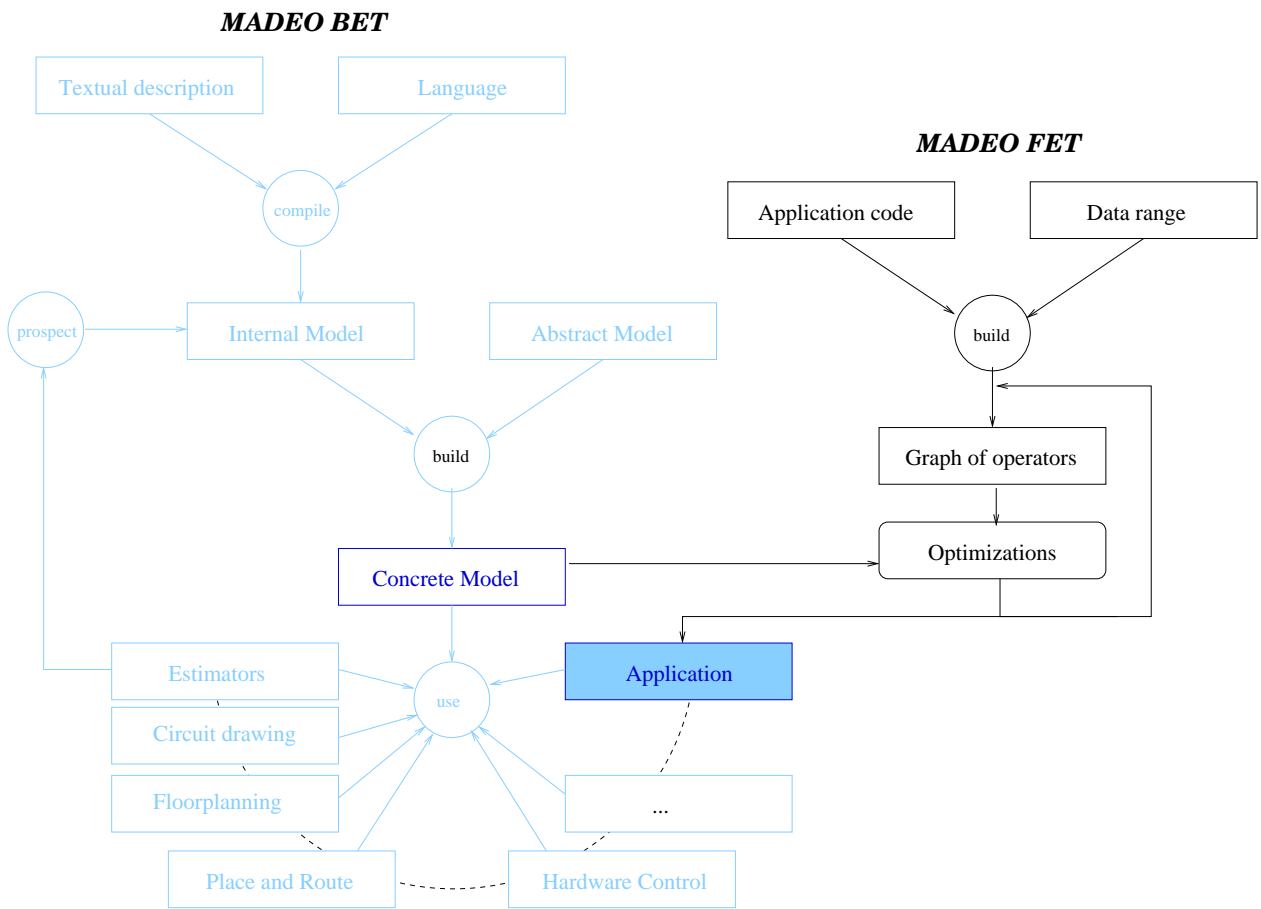


Figure 7.1: MADEO BET and MADEO FET flow

7.2 Possible Types

A type system is defined, based on a set of classes, some of which are presented below:

TYTypeLiteral new: aValue

CODE EX. 7.1: Literals

TYTypeInterval from: start to: stop

CODE EX. 7.2: Intervals

TYTypeRadix radix: aRadix digits: aNumberOfDigits

CODE EX. 7.3: Radix based

TYTypeUnion new: anArrayOfValues

CODE EX. 7.4: Unions

TYTypeGF16 new

CODE EX. 7.5: GF16

TYTypeGF128 new

CODE EX. 7.6: GF128

This set of class is extensible on demand, assuming a minimal API is preserve.

7.3 Optimizations

7.3.1 type inference

By presenting some input values in front of this graph, the type inference is performed.

Type inference consists in computing the output values of every node (using a lazy evaluation mechanism) depending on its input values.

This is done by evaluating all n-up in the inputs (figure 7.2) and results in a high-level object-oriented look-up table (HL OO LUT).

Types are propagated down within the graph so that every node carries the knowledge of its application field. This allows to strongly reduce the complexity of the produced logic (and frees up hardware resources).

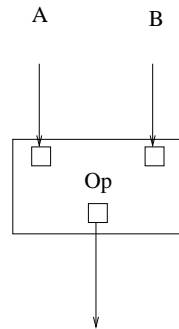


Figure 7.2: A node links an output to some inputs, and computes the output current value, depending on the inputs' current value in order to build the HL OO LUT

notations Let op be an operator taking in_i as inputs and producing out_j

- $|in_i|$ denotes the size of the i^{th} input in term of different values
- $|out_i|$ denotes the size of the i^{th} output in term of different values
- $|op_{in}|$ denotes the size of the inputs range in term of different n-up
- $|op_{out}|$ denotes the size of the outputs range in term of different n-up
- $R_{in}(op)$ denotes the input reduction factor. This represents the importance of the incompatible values among all the theoretical possible values.

$$R_{in}(op) = 1 - \frac{|op_{in}|}{\prod_{i=1}^{i=n} |in_i|} \quad (7.1)$$

Note that $R_{in}(op) = 1$ if and only if all the in_i are independent

- $R_{out}(op)$ denotes the output reduction factor.

$$R_{out}(op) = 1 - \frac{|op_{out}|}{|op_{in}|} \quad (7.2)$$

Programming considerations The $R_{out}(op)$ is a very significant metric over the code quality as the range reduction of type of the values carried between operators only depends on this factor.

A good programming mainly maximizes the $R_{out}(op)$ factors. Note that these factors are multiplicative along a data path.

Example The type inference is done in a depth first mode in order to manage coupled values. The OO HL LUTs are built incrementally.

Consider the following code: $a * b + (a * c)$ with a, b and c ranging from 1 to 10.

In a matter of simplicity, call the first $*$ operator $op1$, the $+$ operator $op2$ and the second $*$ operator $op3$.

$$|op1_{in}| = |a| * |b| = 100 \text{ and } R_{in}(op1) = 1$$

$|op1_{out}| = 42$ and $R_{out}(op1) = 58\%$. In this case, $a * b$ produces 42 different values (as well as $a * c$).

$$|op1_{out}| * |op2_{out}| = 42 * 42 = 1764$$

As there is a dependency between the two variables $a*b$ and $a*c$, because they share the a variable in their inheritance tree, only 798 different values among the theoretical 1764 values are compatibles.

$$|op3_{in}| = 798 \text{ so that } R_{in}(op3) = 1 - \frac{798}{1764} = 0.55.$$

$$\text{So that : } |op3_{out}| = 99 \text{ and } R_{out}(op3) = 1 - \frac{99}{798} = 0.88$$

These input values produces only 99 different output values.

A noticeable point is the earlier the reduction appears, the most efficient the reduction is in term of resources freeing up .

Combinational The type inference is commonly done n-up by n-up to ensure unused values (incompatible values) are not considered. A simple example of this situation is to compute $(a * 2) + (a * 3)$ both ways, either by considering an operator by operator inference and by considering an input n-up by input n-up inference. In the first case, the $|+_{in}| = |a|^2$ because the two branches don't seem to be linked, while in the second case $|+_{in}| = |a|$.

Unfortunately this good property comes with a severe restriction that is the number of inputs must be limited in order not to generate a large amount of n-up. Despite the benefits that can be observed, the second case doesn't appear as a good candidate in two cases. First, in case the inputs really are independent, a node by node inference is simpler and much faster. Secondly, the application field is restricted to cases with a small number of inputs (all the more so as the input's types are huge).

To prevent such a bad situation, both inference algorithms are implemented.

A specific case is to replicate a piece of code by, for example, unrolling a loop by hand. In that case, using a `ArrayedResult` rather than an `Array` to group several values as an output forces the node by node algorithm to be used. The two following graphs (figures 7.3 and 7.4) highlight the code rewriting.

The 7.3 illustrates this situation. The application consists in replicating three times the same process, which makes two additions and a division. All inputs own the same type so that

$$\forall i \in 1 \dots N, |in_i| = |in| \quad (7.3)$$

$$R_{in}(application) = 1 - \frac{|application_{in}|}{\prod_{i=1}^{i=N} |in_i|} \quad (7.4)$$

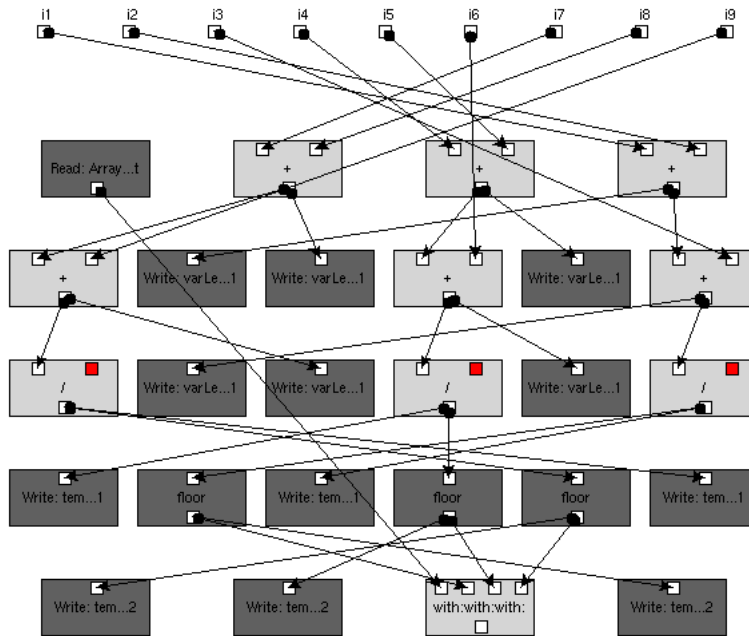


Figure 7.3: The starting code encapsulating an ArrayedResult. Note that the code is structured automatically by extracting all nodes that belong to the inheritance tree of the values

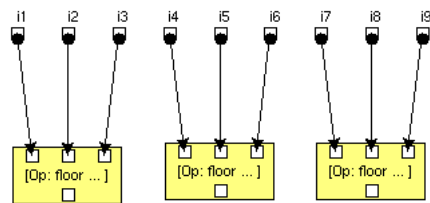


Figure 7.4: After the rewriting process, only three nodes remain. Note the yellow color that denotes the hierarchical nodes.

where

$$\prod_{i=1}^{i=N} |in_i| = |in|^N \quad (7.5)$$

with N the number of inputs.

If the graph can be splitted into p operators acting over q inputs each,

$$N = p \times q \quad (7.6)$$

$$|application_{in}| = p \times |in|^q \quad (7.7)$$

$$R_{in}(application) = 1 - \frac{p}{|in|^{N-q}} \quad (7.8)$$

In this example, $p = 3, q = 3, N=9$ and $|in| = 7 = 2^3$ then

$$R_{in}(application) = 1 - \frac{3}{7^6} = \frac{117646}{117649}.$$

A practical impact of this $R_{in}(application)$ value is that the node by node type inference requires $\approx 1s$ whereas the n-up by n-up type inference needs several minutes to complete. Indeed, when performing a node-by-node type inference, each node is responsible for enumerating its input n-ups based on the input range associated to every of its input values.

7.3.2 code factorization

Common sub expression are detected and reused rather than recomputed.

This relies on the designer implementation. As an example, $a * b * a * b$ cannot be simplified whereas $a * b * (a * b)$ can (see figure 7.5). This is due to the Smalltalk evaluating order, from left most to right most, when reducing arithmetic expressions ¹

7.3.3 dead code removal

A node with no child is designed as a “dead code node” ; it is useless and can be removed from the graph.

Note that removing this node may bring further improvements (eg: this node was the only child of another node, which then becomes itself a “dead code node”).

7.3.4 constant folding

A composite type owning only one value is automatically converted into a literal type, what implies its producer node is useless and can be removed from the graph. Conditional nodes are special cases: in case the condition is constant, only the one branch that makes sense is kept, replacing the conditional node itself. In case, one of its branches is constant, the branch is replaced by a literal.

¹Basically, operations are splitted into three groups, within which no priority exists, namely: unary operations (the one that take no parameter), binary operations (a limited set of operations such as $+ - * modulo$) and the keyword based operations (keywords end with the $:$ character). Unary operations are reduced first, then binary operations, and at the end the keyword based operations. For more information about this, please refer to the Smalltalk documentation.

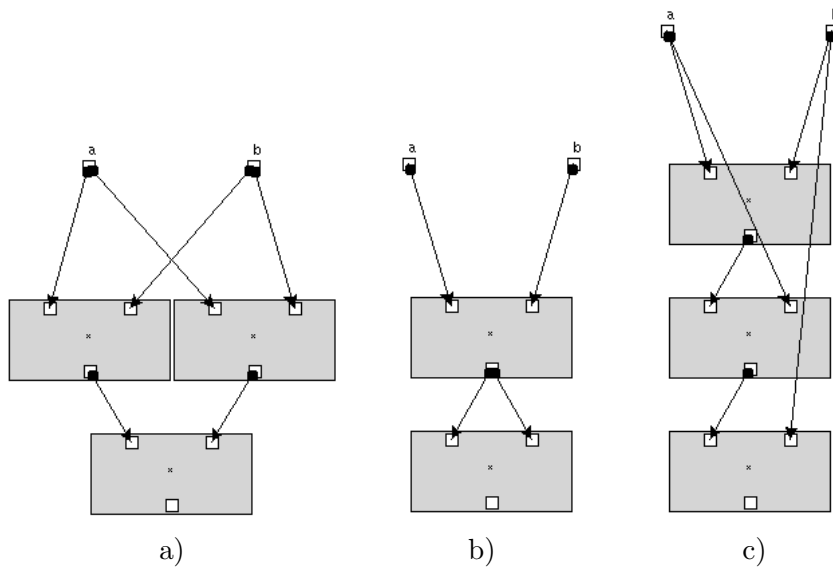


Figure 7.5: a) this code can be simplified as some common subexpression appear
 b) the result the simplification produces
 c) an equivalent expression in term of computing process that CANNOT be simplified

7.3.5 no op removal

As the only important information is the number of different values within a type, a bijective node appears as a “no-op node”. This operation reduces the graph by adding the inputs of the suppressed node to its consumers. This means that some node may own more inputs that their original message accepts.

7.3.6 operator fusion

For all values owning only one consumer, this consumer and the producer of the value can be merged without any information loss.

Another example is simplifying expressions over arrayed values, but removing the *at* : nodes which aim is only to pick up a sub value of the global value.

7.3.7 Automatic decomposition

At a starting point, the graph reflects the algorithmic decomposition. Nevertheless, if the inputs type are too big, the OO HL LUT won't fit the max complexity limit that the logic synthesis algorithm forces.

A solution is to split the biggest type on demand, and to apply the operator over the resulting sub-domains. This implies to check the type

inclusion, to make a branch-bound.

$$a \text{ op } b \rightarrow \text{if}(a \in |a|_1) \text{ then } a \text{ op}_1 b \text{ else } a \text{ op}_2 b \quad (7.9)$$

with op , op_1 and op_2 equivalent operators acting over several sets of values, which are then associated different OO HL LUTs.

7.3.8 Operator flattening

Hierarchical nodes allows to go further on when decomposing a process into a graph of operators. The only processes that can benefit from a hierarchical decomposition are built from some methods that the target class includes. This limitation comes from a will to ease the decision for the compiler. Sending messages to the pseudo variable *self* - which represents the object itself - results in adding hierarchical nodes to the graph (see example 9.1). Nevertheless, These nodes can be flatten if required.

Depending on the semantic of the operation, one can force some code specification owning *self* sending messages to be considered as some atomic nodes (as an example nodes with low fan in-fanout, but high internal complexity) ; in this case the node is only associated a OO HL LUT whereas classical hierarchical nodes are linked as well to a sub graph which nodes are provided a OO HL LUT.

This can happen twice, either because the designer knows hierarchical decomposition must be prevented or because of contextual information such as the size of the OO HL LUTs.

- Methods located within the 'do not synthesize' methods' category produce pseudo-atomic hierarchical nodes whatever data they act over.
- In addition, when producing the logic, any hierarchical node can be seen as a hierarchical node on demand.

7.4 Logic generation

7.4.1 Blif generation

Logic generation means to produce a RTL equivalent description back from another description, usually of a higher level of abstraction. In the MADEOfet design flow, logic generation means to produces BLIF description task from OO LUTs.

After the optimization stages, each node is provided a OO HL-LUT.

To ensure every object contained within the OO HL LUTs can be represented within the hardware, the objects are replaced by their index within their associated type.

Another solution is to use an ad-hoc encryption process (ex: GF 16) , as the binary representation policy of the objects has a strong impact over the logic density.

These new LUTs are called OO LL LUTs (OO Low Level LUTs) and only embed binary values.

They are used to produce a PLA representation which is minimized using EXPRESSO before the technology mapping stage is done by calling some of the SIS algorithms[3].

Binarisation impact Let's consider a simple process that swap the most significant and less significant parts of a word. Obviously this requires no logic as long as conventional binary representation is used.

Now consider the same operation over less data, what means the data's range may be preserved but $|op_{in}|$ reduces. By exchanging the data and their associated index, the process complexity hugely increases because there is a need for encoding and decoding the values/indexes couples.

The 7.7 BLIF description highlights the absence of logic when swapping the four most and less significant bits of a char. On the opposite, the 7.8-7.9 logic description, far from being simpler than the 7.7 one, as its simpler data range should have suggested, appears as much more complicated due to the semantic loss the indexes generate.

```
.model t5
.inputs a_1 a_2 a_3 a_4 a_5 a_6
.outputs t5_1 t5_2 t5_3 t5_4 t5_5 t5_6
.clocks
.names a_1 t5_1
1 1

.names a_2 t5_2
1 1

.names a_3 t5_3
1 1

.names a_4 t5_4
1 1

.names a_5 t5_5
1 1

.names a_6 t5_6
1 1

.end
```

CODE EX. 7.7: Swapping the most/less significant four bits for an integer ranging from 0 to 63

7.4.2 EDIF Generation

When using coarse grain architectures (e.g. reconfigurable datapath), it is important to cut off the inference process as no benefits can be expected of whereas this stage will consume a lot of time if not fail.

In that case, to enable place and route, a specific policy is used, which is to produce an EDIF netlist of operators that make references (through symbolic naming) to some primitives that are supported within the hardware (mult, add, etc. . .). This information must be coupled to the AMONG field of the grammar description of the architecture within MADEO-BET. An example is provided in appendix C.

7.4.3 Mixed netlist

A current work is going on to support heterogeneous netlist that is to allow mixed coarse grain and fine grain operators netlist. This aims to restrict the use of BLIF production to the operators for which a benefit can happen while saving time by relying on some libraries of more classical operators if not.

As an example, the control (automata) often requires only few bits and can expect huge improvement by tuning the operators to meet their requirements. On the contrary, the coarse grain controlled operators (such as ALUs) must be kept as symbols.

We guess that lots of DSP programs could take advantages of such an opportunity.

7.5 Tips

This section highlights several noticeable points when writing a code to be taken as input by MADEO-Fet.

7.5.1 Temporaries

When compiling the code, if a variable reference is met, the variable is added as input recursively to the outer context until either the context owns the variable as input or as temporary, or the outer context is nil (top block).

As a consequence, forgetting to declare temporaries results in adding invalid inputs, what may lead to incompatible typing. Another bad consequence is that temporaries are then handled as *Undeclared* what slows down the inference process.

7.5.2 ArrayedResult

One of the optimization steps is *dead code removal* so that the nodes with no consumers are destroyed. Using ArrayedResult allows to group several results into a single value. Note that this is only useful for the last operators of the method, and that this node shall not be asked to produce a BLIF description; the *produce blif without last* options is designed in this scope.

7.5.3 Array

Array is used to overcome the classical limitation of smalltalk that is only one value can be returned. By using Array, it is possible to group values

within a single value. In that case the *split* optimization will substitute this value by its internal values.

7.6 the *at* : message

The *at* : owns a special meaning when it applies to an array and when its argument is constant. In this case, the *operator fusion* results in destroying the node and in substituting its output by its input's *argth* internal value.

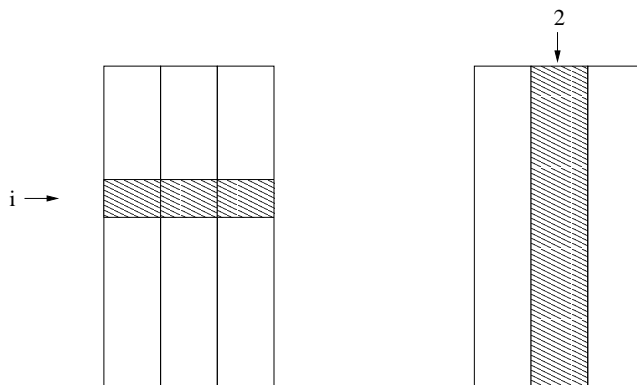


Figure 7.6: On the left, indexing by a variable results in picking a line (an n-up). On the right, indexing by a constant (or a value that became a constant due to the optimizations options) results in picking a column, that is extracting a subValue.

7.6.1 Comparisons

The *true* and *false* smalltalk pseudo variable can be used for typing (e.g. within a *TYTypeUnion*) but if so, they will not support the *ifTrue* :, *ifFalse* :, *ifTrue* : *ifFalse* : nor *ifFalse* : *ifTrue* : messages.

A solution is to compare the value with either *true* or *false* before calling the conditional message.

Another interesting point regarding comparisons, is that the arguments ordering is extremely important.

```
0 = 0 ifTrue:[ 0 ] ifFalse:[ b / a ] is correct
a = 0 ifTrue:[ 0 ] ifFalse:[ b / a ] is not
```

CODE EX. 7.10: when comparing a value to a literal, the literal must be the receiver

In the example 7.10 the second line is not correct and the condition is always evaluated as *false*. This implies that the operators = and *ifTrue* : *ifFalse* : are destroyed, and that as a consequence, *b/a* is evaluated when *a* equals zero, leading to an unrecoverable error.

```

.model t5
.inputs a_1 a_2 a_3 a_4 a_5 a_6
.outputs t5_1 t5_2 t5_3 t5_4 t5_5 t5_6
.clocks
.names a_1 [342] t5_1
11 1

.names a_2 [383] t5_2
11 1

.names a_3 [424] t5_3
11 1

.names a_4 [465] t5_4
11 1

.names a_5 [506] t5_5
11 1

.names a_6 [547] t5_6
11 1

.names a_2 a_3 a_4 [341]
0-- 1
-0- 1
--0 1

.names a_5 a_6 [341] [342]
0-- 1
-0- 1
--1 1

.names a_1 a_3 a_4 [382]
0-- 1
-0- 1
--0 1

.names a_5 a_6 [382] [383]
0-- 1
-0- 1
--1 1

```

CODE EX. 7.8: Swapping the most/less significant four bits for an integer ranging from 1 to 63

```
.names a_1 a_2 a_4 [423]
0-- 1
-0- 1
--0 1

.names a_5 a_6 [423] [424]
0-- 1
-0- 1
--1 1

.names a_5 a_6 [546] [465]
--1 1
-0- 1
0-- 1

.names a_4 a_6 [546] [506]
--1 1
-0- 1
0-- 1

.names a_1 a_2 a_3 [546]
0-- 1
-0- 1
--0 1

.names a_4 a_5 [546] [547]
0-- 1
-0- 1
--1 1

.end
```

CODE EX. 7.9: end of the 7.8 example

Chapter 8

Graphical tool

The graphical tool is accessible through an icon of the VisualLauncher toolbar as shown in figure 8.1



Figure 8.1: Clicking this icon opens the MADEO-FET graphical tool, as illustrated by figure 8.2

The graphical tool (see figure 8.2) eases choosing a class and an method as an entry point, as well as defining the inputs types; and provides the user a graphical feed-back by drawing the operators graph. Once this is done; the optimizations are accessible ¹

8.1 Tool

8.1.1 Control panel

The upper left part of the interface includes the control panel (CP) as shown on figure 8.3. This CP shows the selected class as well as the selected selector (ie the entry point). Under the class/selector information are the available environment information made of two list, and a textual field.

The left most list represents the available contexts out of which a given context is selected. This selected context is made of several variables that are shown in the second list. Each of these variables is associated a type, which textual definition is presented within the input field, and which definition can be replaced using the button next to the input field.

¹Note that the dependencies between the optimizations stages are not managed yet. Swapping two optimizations steps may result in uncontrolled changes of the overall process.

A future work will be to let the designer to choose among some optimizations scenario .

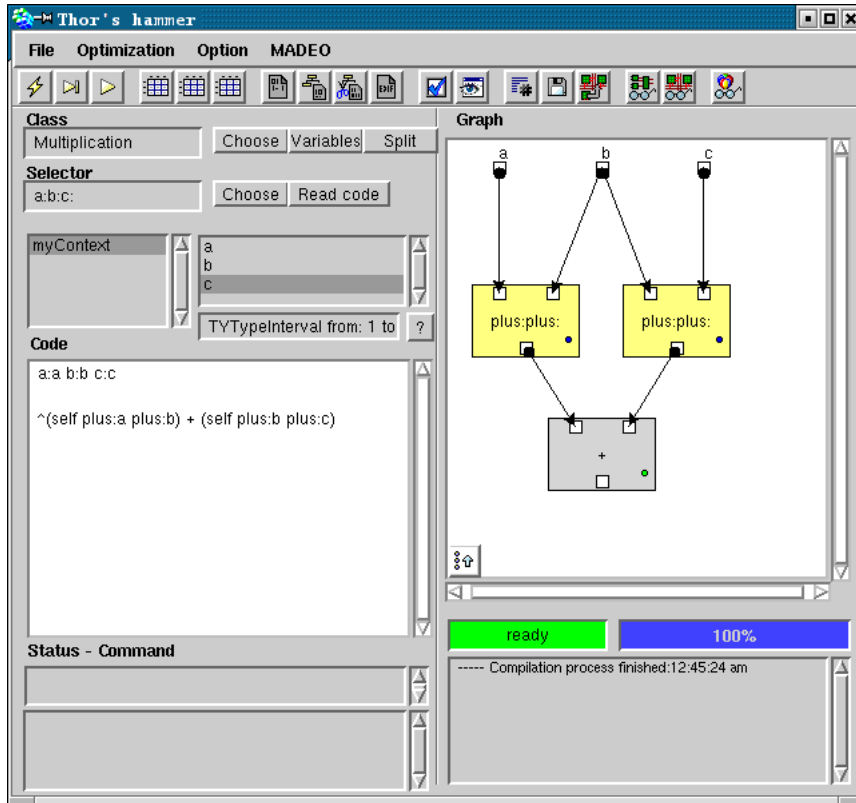


Figure 8.2: The MADEO FET user interface

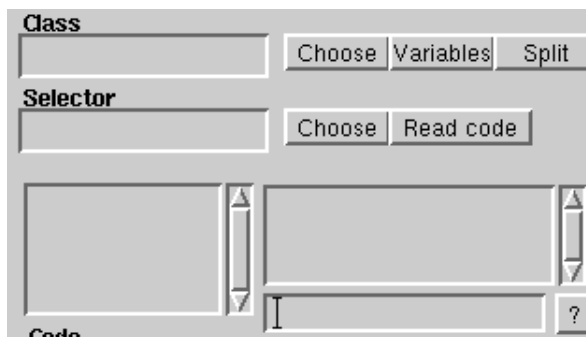


Figure 8.3: The control panel

The contexts can be saved on disk and loaded from using the context's menu as illustrated by figure 8.4. This menu allows as well as to perform the mapping between the context and the data flow graph, what means the inputs of the graph are associated a type the context looks up by its name.

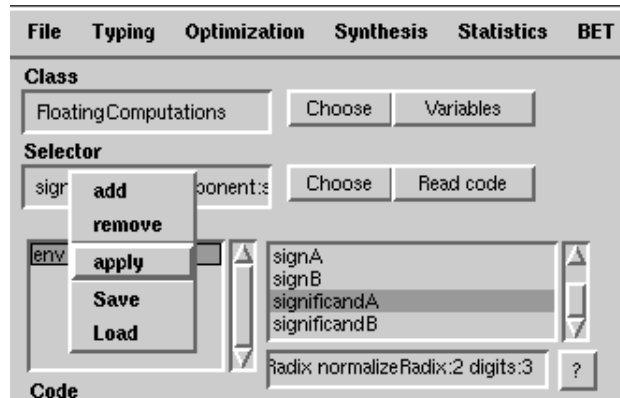


Figure 8.4: The context menu

8.1.2 The ToolBar



Figure 8.5: The Madeo-Fet toolBar

The toolbar allows to launch different process. From left to right:

full run compiles code, applies context, optimizes

build compiles code, applies context but do not optimize

optimize performs the selected optimizations

view LUT opens a windows over the global OO-Lut

view indexed LUT opens a windows over the global indexed LUT

view encoded LUT opens a windows over the global encoded LUT

simple lutify builds a flatten BLIF (green spot over the nodes' graphical representation)

complex lutify builds a hierarchical BLIF (blue spot over the nodes' graphical representation)

complex lutify without last builds a hierarchical BLIF ignoring the last operator of the graph

generate EDIF builds an EDIF description of the graph

check tests the resulting BLIF with regards to the original code behavior

simulate open a bit-accurate simulation window

generate method produces a structural description of the graph and stores it as a method

save saves this structural description as Binary Object Storage System (BOSS)

implement calls MADEO-BET in order to place and route the logic

schematic opens the schematic editor over over the graph's equivalent schema

made-bet opens MADEO-BET

browse opens a code browser over the target class

8.1.3 Code menu

The first item of the menu (see figure 8.6) allows either to validate a change in the code definition or to open a browser over the class

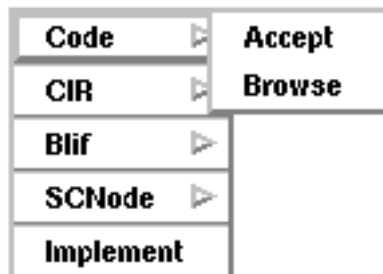


Figure 8.6: The code menu

The second item (see figure 8.7) allows to launch the compilation process as well as the selected optimizations procedures.

The result can be inspected and is represented in the right side of the window. The representation can be isolated within a stand-alone window if desired, and the graphical parameters such as the size of the graph's nodes and the spacing between nodes can be tuned.

The new run item automates the code modification, type forcing, and redraw of the resulting graph.

Also, this menu enables to open an interface over the global LUT representing the design if this is already built. This LUT is used to check the design accuracy.

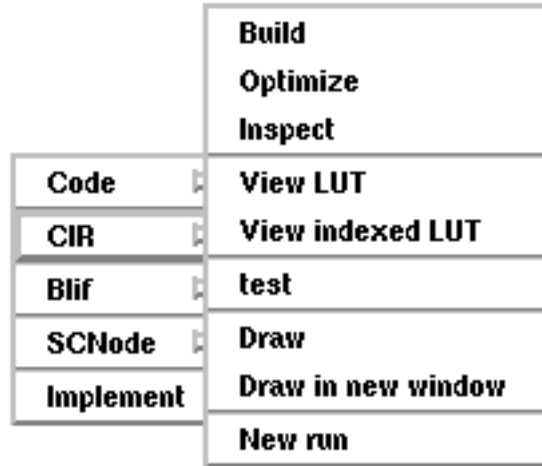


Figure 8.7: The CIR menu

The third menu item (see figure 8.8) deals with logic synthesis (BLIF building). Three modes are available:

simple lutify takes the OO LUT associated with the global lut as an entry point to generate a single flatten logic description.

complex lutify builds a logic description for all the nodes of the graph and a hierarchical description of the graph itself, by calling the sub modules associated with each node.

Complex lutify without last has the same semantic but ignores the last in case of aggregating several results in the last node (based on the Array class). Aggregating result prevents the dead code optimization to destroy valid nodes.

Once the BLIF description has been made, it can be inspected and simulated.



Figure 8.8: The BLIF menu

The SCNode is the articulation formalism between MADEO FET and

MADEO bet (see section 4.3). This carries the same information than the graph, with additional geometric properties. The figure 8.9 shows the action regarding the SCNode representation.

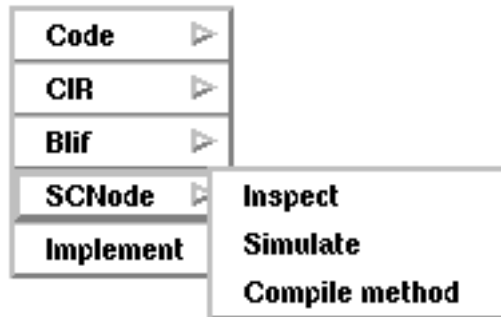


Figure 8.9: The SCNode menu

The last item permits to implement the application onto a reconfigurable target.

8.1.4 Graphic representation

As show in figure 8.10, the graph drawing makes use of coloring to distinguish between the different classes of nodes as follows:

Light gray denotes atomic nodes ,

Dark gray denotes terminal nodes,

Yellow denotes hierarchical nodes,

Red denotes hierarchical nodes that are seen as atomic ones.

The last two cases differ from one other either because the method is placed in the *do not synthesize* category or because the designer has set the flag manually using the contextual menu associated with the nodes. This is one of the reasons that motivated the second icon to be present in the toolbar as this allows to select the kind of node to be used for a method before performing the optimizations.

In the same way, the IOs of the node appear as white boxes except in case of literal type which are red painted.

Another graphical annotation concerns the BLIF description associated with the node, that appears either as a blue spot over the node (hierarchical BLIF) or as a green one (simple flatten BLIF).

The graph drawing area owns a menu that enables to browse the graph's elements and to go through the hierarchy levels (depending on the nodes' kind). Popping back the stack is accessible through a button in the bottom left corner of the drawing area.

Last, the option item from the global menu allows to set some parameters such as spacing between nodes to customize the drawing.

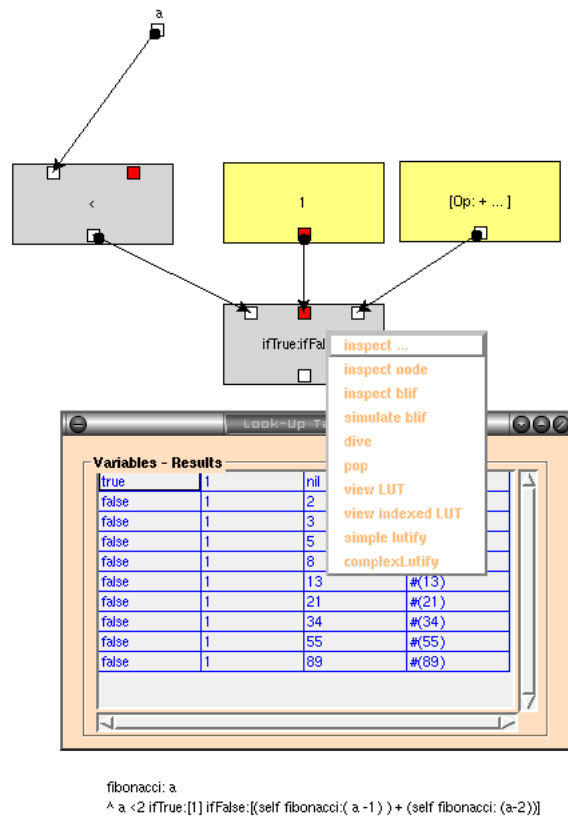


Figure 8.10: Example: Fibonacci process

8.2 Interfacing low level tools (MADEO-bet)

Interfacing MADEO-bet and MADEO-FET relies on two mechanisms:

Compiling a method which is responsible for producing the SCNode, is a common mechanism to store permanently a description. This description is then invoked through the classical message sending mechanism, from MADEO-bet. The low level tools and MADEO-FET, by sharing the SCNode, may interleave subcalls.

Realizing a place and route requires to select the target architecture, and results in a module (ie a piece of configuration) that implements the source application onto the selected architecture.

The figure 8.11 shows the menu item that pops up the dialog window in which the available architectures are listed.

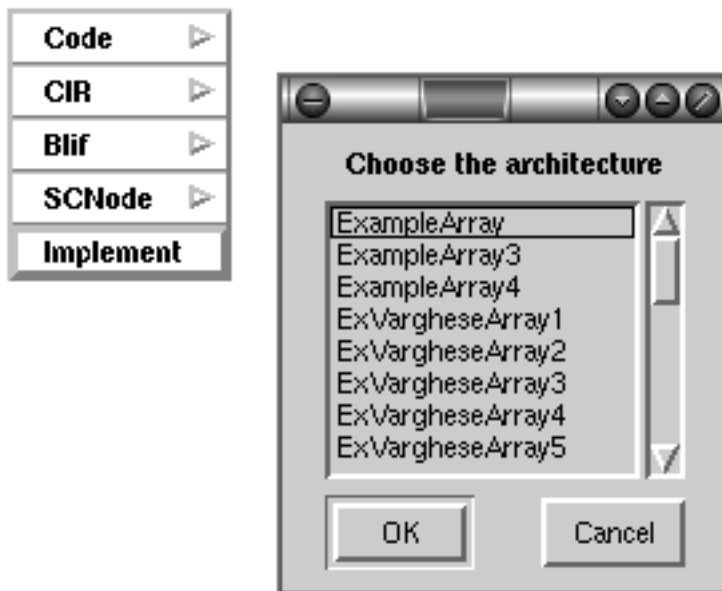


Figure 8.11: The interfacing menu item and the architecture selection window

Chapter 9

Example: Defining a floating point multiplier

9.1 Introduction

This chapter describes through an example how to define a new circuit. The chosen example is a floating point multiplier.

By precisely tuning its parameters (ie the value's range for the numbers), it's possible to get custom FP multipliers that can be stored into any operators library. The design is fully tested at high level, and is optimized to produce a well suited logic description. Again this description is ready to be placed and routed using the MADEO-bet layer.

This section introduces first the algorithm that we use before describing its implementation within the framework. The graph of operators which results from the optimizations stages is introduced and the placed and routed logic is shown.

9.2 The implemented algorithm

We refer to the 4.8 section of the [9] book

The numbers are described using a fixed number of bits. That means the precision is strictly known. The numbers are made of a sign, a significand and an exponent. The sign is either 0 or 1. the significand is a fraction ranging from 1 to $2^{-\epsilon}$ where ϵ depends on the number of bits. The (biased) exponent is a signed number ranging from $-n$ to $n - 1$ depending on the number of bits.

9.3 Implementation

9.3.1 The methods

```

sign: signA significand: significandA exponent: exponentA sign:
signB significand: significandB exponent: exponentB
  | sign exp significand normalize |
  sign := self computeSignFor: signA and: signB.
  significand := self computeSignificandFor: significandA and: significandB.
  exp := self computeExponentFor: exponentA and: exponentB.
  normalize := self normalizeSignificand: significand.
  ^Array
    with: sign
    with: (normalize at: 1)
    with: exp + (normalize at: 2)

```

CODE EX. 9.1: The entry point. Each float appears as three values : a sign, a significand, and an exponent

This code is the entry point; it makes use of four other methods that describes internal computation stages.

```

computeExponentFor: exponentA and: exponentB
  ^ exponentA + exponentB

```

CODE EX. 9.2: The first operation over the exponents

```

computeSignFor: signA and: signB
  ^ signA + signB \\ 2

```

CODE EX. 9.3: The operations over the signs

```

computeSignificandFor: significandA and: significandB
  ^ significandA * significandB

```

CODE EX. 9.4: The full operation over the significands, ignoring the wished data width

normalizeSignificand: aSignificand

“This methods normalize this parameter value by extracting two sub values. The first one is the value, shifted if needed, and truncated so that the max number of bits to represent the value is preserved. The second value is the shift factor that is to be added to the exponent value”

```

| shift string in tmp index |
string := ".
in := 1.
tmp := aSignificand.
[tmp = 0] whileFalse:
  [| a |
   a := tmp // (2 raisedTo: in).
   tmp := tmp \\ (2 raisedTo: in).
   string := string , a printString.
   in := in - 1].
index := string indexOf: $1.
shift := 2 - index.
string := string copyFrom: index to: ((index + SignificandBits) min:
string size).
^Array
  with: (Compiler evaluate: '2r' , string) / (2 raisedTo: string size -
1)
  with: shift

```

CODE EX. 9.5: The significand normalization implies to shift and truncate the value before returning it, but to answer as well the shift that was applied in order to carry that shift over the exponent value

Code comments Note that most of the internal stages are simple computations. Only the *normalizeSignificand*: (9.5) is made of operations that are not suited for an hardware implementation. This is handled by locating the method in the *do not synthesize* category of method. As a result this hierarchical node is considered as a simple node, owning a single LUT.

9.3.2 The types

Three types are used in this example.

The simplest type is the one associated with the signs. It's simply a boolean union (0 or 1). This type can be expressed either by using a simple interval or by defining a TYTypeRadix followed by two parameters : the radix and the number of digit (as shown in 9.6).

```
TYTypeRadix radix:2 digits:1
```

CODE EX. 9.6: First type

The second type is the one associated with the exponent. It's an interval of values ranging from $-n$ to $n + 1$

 TYTypeInterval from: -7 to: 8

 CODE EX. 9.7: Second type

 TYTypeRadix normalizeRadix:2 digits:3
 OrderedCollection (1 (9/8) (5/4) (11/8) (3/2) (13/8) (7/4) (15/8))
 OrderedCollection (1.0 1.125 1.25 1.375 1.5 1.625 1.75 1.875)

CODE EX. 9.8: Third type, with its associated values, either fraction or float, ranging from 1 to $2 - \epsilon$ with $\epsilon = 2^{-3}$

9.3.3 The resulting graph

The optimizations performed over the graph are illustrated by the following snapshots. The first snapshot represents the starting graph, which is simply built up based on a static analysis of the designer code.

The next snapshot (9.2) shows the graph after the type inference and the dead code removal are finished.

```

.model t38
.inputs t32_1 t32_2 t32_3 t32_4 t32_5 t32_6
.outputs t39_1 t39_2 t39_3 t40
.clocks
.names t32_5 t32_6 [41] [4]
101 1

.names [41] [634] [5]
11 1

.names [4] [26] [720] [6]
1-- 1
-11 1

.names t32_1 t32_6 [41] [7]
101 1
...

```

CODE EX. 9.9: The Blif description of the 9.4 table enlightens two characteristics of the node : first, the $R_{out}(op)$ reduction factor is important as the inputs are encoded using 6 bits and the outputs only require 4 bits, and secondly the outputs are called $t39_1..3$ and $t40$, what means the nodes owns two outputs, the first of which requires 3 bits and the other one only one bit.

By performing the operators fusions the at : access to the arrayed value produced by the $normalizeSignificand$: node disappear (figure 9.3) while this value is splitted into to values (figure 9.4) that the other nodes con-

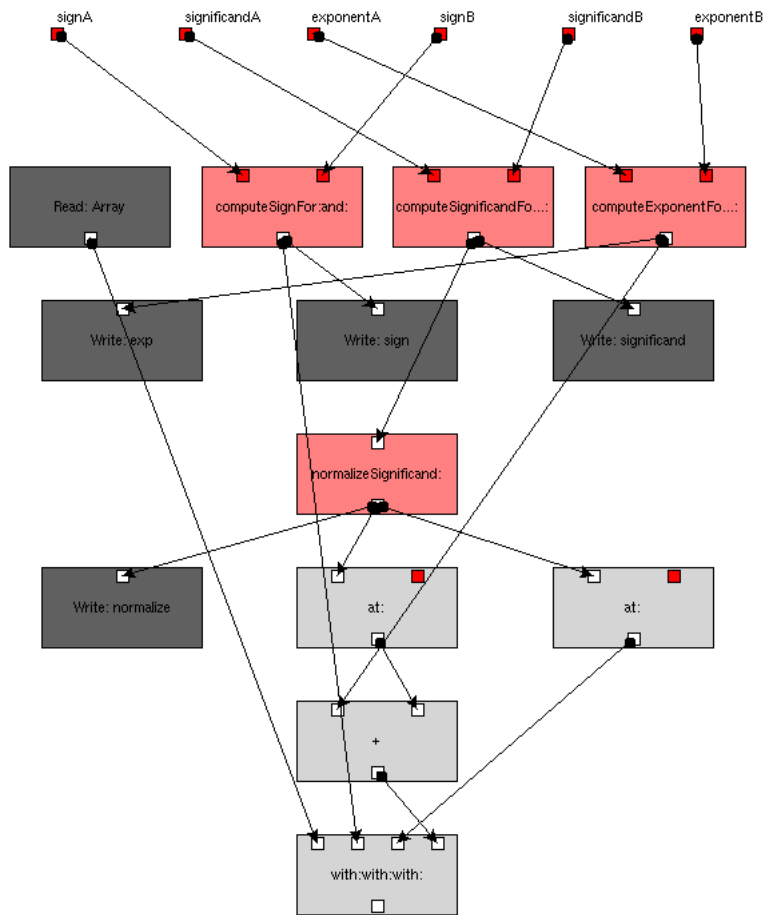


Figure 9.1: Example: A small floating point multiplier

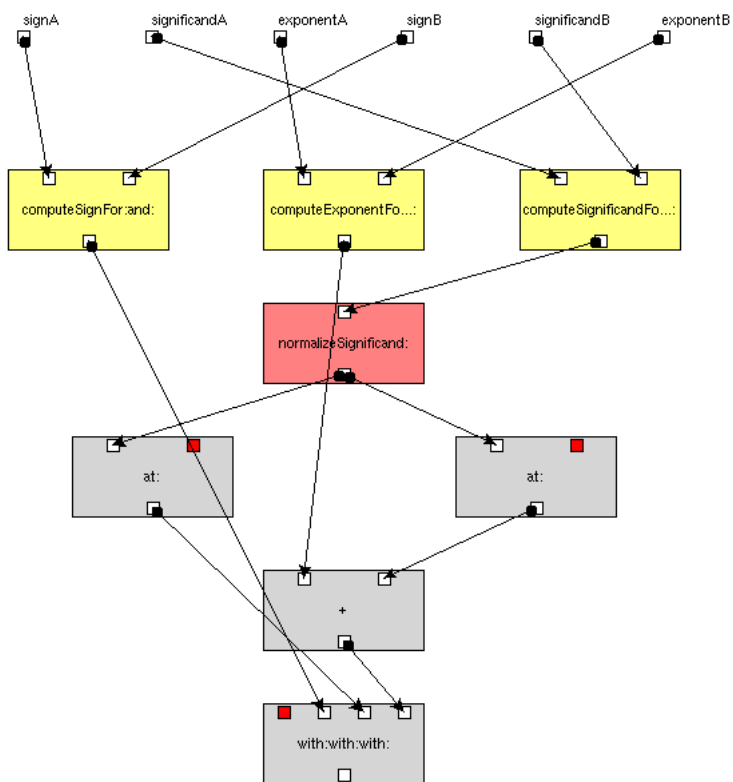


Figure 9.2: Example: The same graph after type inference and dead code removal

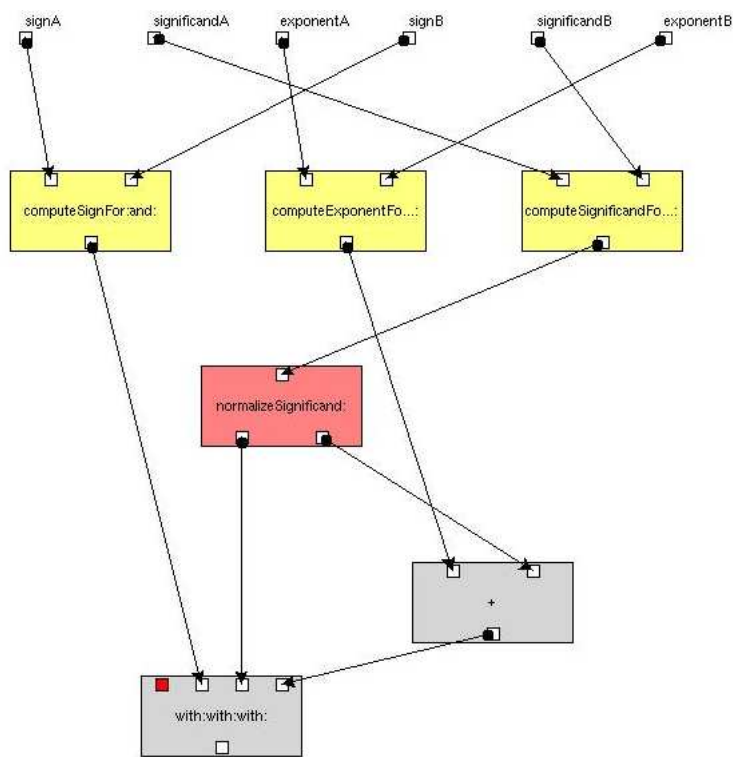


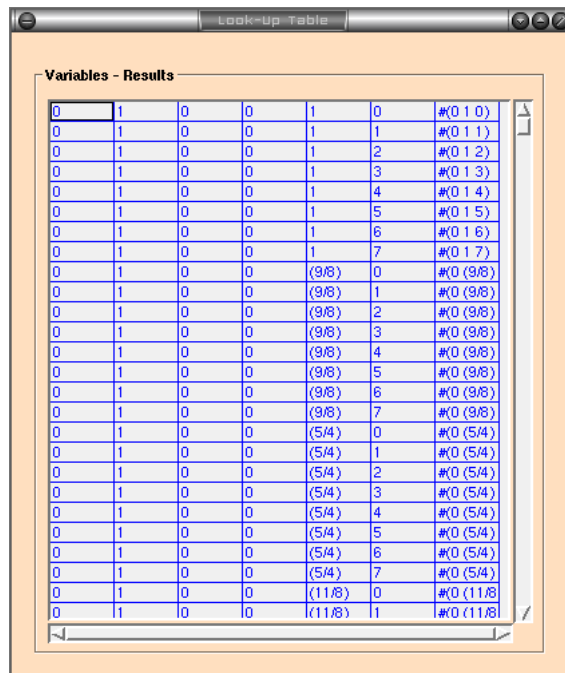
Figure 9.3: Example: The same graph after the operators fusion

Variables	Results	Results
1	1	0
(9/8)	(9/8)	0
(5/4)	(5/4)	0
(11/8)	(11/8)	0
(3/2)	(3/2)	0
(13/8)	(13/8)	0
(7/4)	(7/4)	0
(15/8)	(15/8)	0
(81/64)	(5/4)	0
(45/32)	(11/8)	0
(99/64)	(3/2)	0
(27/16)	(13/8)	0
(117/64)	(7/4)	0
(63/32)	(15/8)	0
(135/64)	1	1
(25/16)	(3/2)	0
(55/32)	(13/8)	0
(65/32)	1	1
(35/16)	1	1
(75/32)	(9/8)	1
(121/64)	(15/8)	0
(33/16)	1	1
(143/64)	1	1
(77/32)	(9/8)	1
(165/64)	(5/4)	1
(9/4)	(9/8)	1

Figure 9.4: Example: By analyzing the LUT of the *normalizeSignificand* : operator, it's easy to note that the operators has now two outputs (see code 9.9)

sume. This strongly reduces the complexity of the cartesian product the *normalizeSignificand* : produces as an output.

Once the optimizations are done, the designer gets a global LUT for the circuit on demand



The screenshot shows a window titled "Look-Up Table" with a sub-header "Variables - Results". The table contains 28 rows of data. The first 14 rows have binary inputs (0 or 1) and outputs ranging from 0 to 7. The next 14 rows have binary inputs (0 or 1) and outputs in hexadecimal notation, such as (9/8), (5/4), and (11/8).

Row	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
0	1	0	0	1	0	#(0 1 0)	
0	1	0	0	1	1	#(0 1 1)	
0	1	0	0	1	2	#(0 1 2)	
0	1	0	0	1	3	#(0 1 3)	
0	1	0	0	1	4	#(0 1 4)	
0	1	0	0	1	5	#(0 1 5)	
0	1	0	0	1	6	#(0 1 6)	
0	1	0	0	1	7	#(0 1 7)	
0	1	0	0	(9/8)	0	#(0 (9/8))	
0	1	0	0	(9/8)	1	#(0 (9/8))	
0	1	0	0	(9/8)	2	#(0 (9/8))	
0	1	0	0	(9/8)	3	#(0 (9/8))	
0	1	0	0	(9/8)	4	#(0 (9/8))	
0	1	0	0	(9/8)	5	#(0 (9/8))	
0	1	0	0	(9/8)	6	#(0 (9/8))	
0	1	0	0	(9/8)	7	#(0 (9/8))	
0	1	0	0	(5/4)	0	#(0 (5/4))	
0	1	0	0	(5/4)	1	#(0 (5/4))	
0	1	0	0	(5/4)	2	#(0 (5/4))	
0	1	0	0	(5/4)	3	#(0 (5/4))	
0	1	0	0	(5/4)	4	#(0 (5/4))	
0	1	0	0	(5/4)	5	#(0 (5/4))	
0	1	0	0	(5/4)	6	#(0 (5/4))	
0	1	0	0	(5/4)	7	#(0 (5/4))	
0	1	0	0	(11/8)	0	#(0 (11/8))	
0	1	0	0	(11/8)	1	#(0 (11/8))	

Figure 9.5: Example: The global lut associated with the design

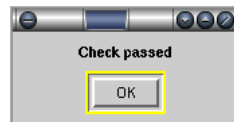


Figure 9.6: Example: The designer is free to check his circuit's accuracy

9.3.4 Towards infinite precision operators

This example exhibits some good properties, one of which is that the precision loss is only due to the *normalizeSignificand* : method semantic rather than being intrinsic to the implementation. When designing an operation, it's the designer choice to round or not the value. By pointing precisely the location within the graph at which to round the values, the designer explicit the precision loss.

Another solution is to prevent this loss by letting the algorithms carry the accurate values, and postponing the truncation till the logic synthesis stage. This could be done by applying a restriction to the type encoding after the type inference completes. This bring further optimization by reducing set of output values (increasing $R_{out}(op)$).

A smart implementation of this mechanism consists in encapsulating the target node within a custom *precision reduction* node which carries the specific encoding for the output values. This is not archived yet, but is to be done.

9.3.5 Conclusion

This example illustrates how to write a piece of software that will result in designing a circuit over a reconfigurable architecture. A clear benefit of the method we introduced is that arbitrary circuits can be specified at a symbolic level, letting the algorithms optimize them to produce a well suited logic, which is specialized either by the data range and possibly later on by the target architecture. The algorithmic formulation can be tested at both high level (symbolic code) and low level (RTL).

By coupling MADEO BET to this layer, the designer can tune his coding and evaluate the impact of the changes over the produced circuit. That is, in the same way that MADEO BET allows to prospect the hardware field in front of fixed application, MADEO FET allows to prospect the algorithms to be implemented in front of a fixed hardware. At this time, a promising feature is to bring closer abstract algorithms such as efficient arithmetic operators (which can benefit from non conventional number representations) and the resulting mapping of these algorithms over reconfigurable hardware. A special care must be carried to analyzing the matching between the expected performances, based on abstract evaluation, and the measured performances, based on the placed and routed circuit.

Chapter 10

Schematic design

10.1 Introduction

In order to ease the definition of simple circuit, a schematic editor is provided within MADEO. However, this user interface does not support the full semantic of the SCNodes. Mainly, the `SCComposite1DNode` and the `SCComposite2DNode` which include regular patterns in terms of nodes and connections cannot be expressed using this tool. Such nodes remain accessible to the user so that it is possible to build more complicated circuit by composing sub nodes with no restriction on the node's kind.

10.2 Position within the flow

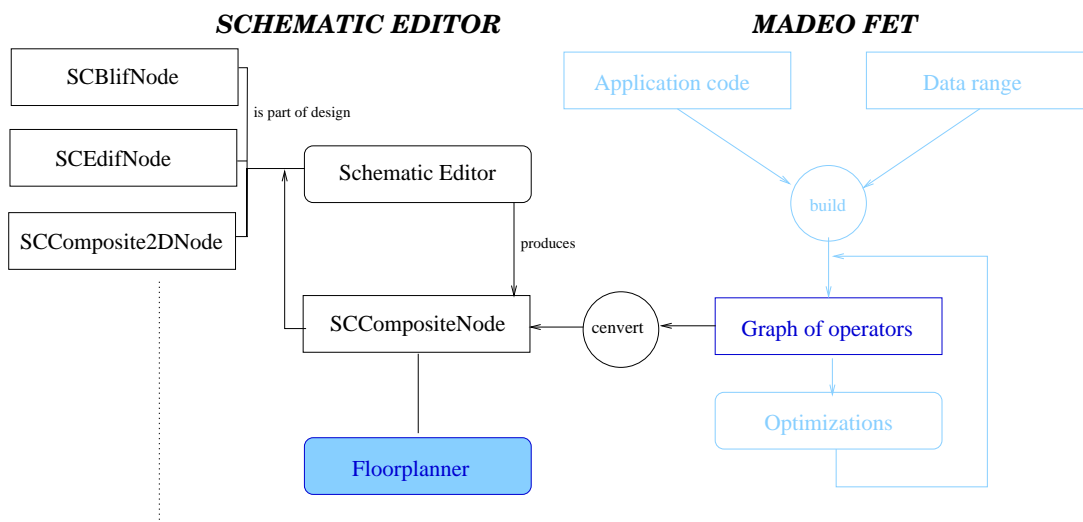


Figure 10.1: Interfacing MADEO-FET and the schematic editor

The schematic editor can reuse any node MADEO-FET has produced, in the same way than any other kind of node (`SCBlifNode`, `SCEdifNode`, etc. . .).

Once the design is done, it results in a new node as shown in figure 10.1. The nodes the schematic editor produces are commonly used by the floorplanner (see chapter 4.3).

10.3 Tool

The Schematic Editor is accessible through an icon of the VisualLaunch toolbar as shown in figure 10.2



Figure 10.2: Clicking this icon opens the schematic editor interface, as illustrated by figure 8.2

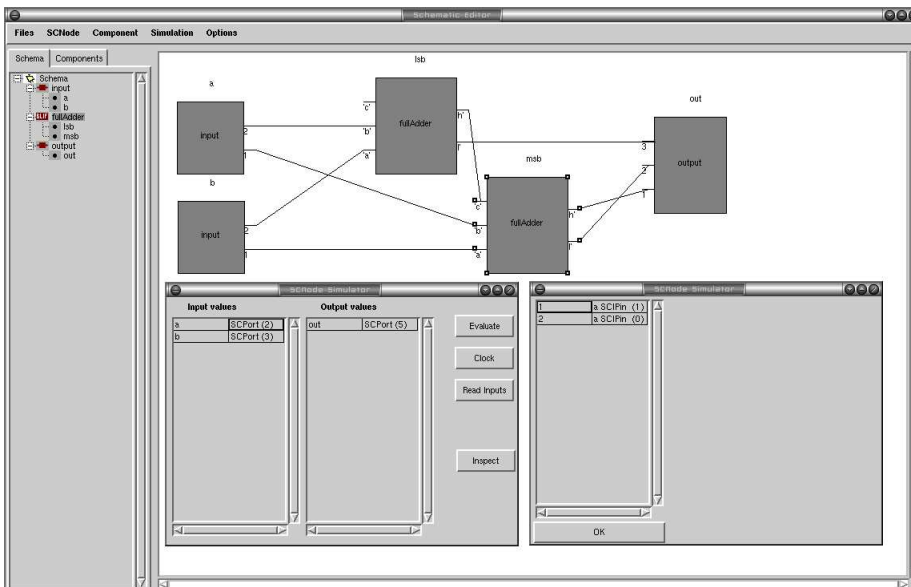


Figure 10.3: Example: Designing a two bit adder by composing two full adders

The interface is split into two sub windows. The left one shows either the elements the library contains or the elements the design includes. The tree representation highlights the instantiations of the modules. The right side of the interface shows the schematic drawing. The nodes are chosen using the SCNodeChooser interface (figure 10.4). The node chooser produces nodes which retain the way they were created. The result of the design

then produces a new composite node, because each of its subnodes can be re-created on demand. This new node can be simulated as shown in the snapshot and may also be inserted within this library on demand, by providing a method's name.

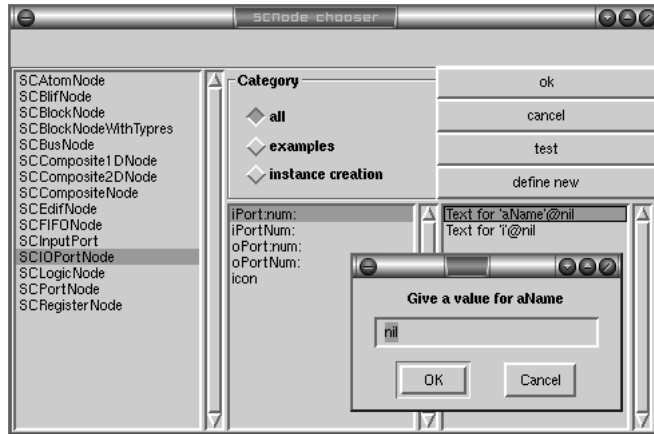


Figure 10.4: The SCNode Chooser interface enables to pick a node up and to tune its parameter if any

10.3.1 Example

This tool is illustrated by a simple example. By combining two full adders ($+_1$ and $+_2$), a simple 2 bits adder is built up.

$$out_0 = +_1^{low}(a_0, b_0) \quad (10.1)$$

$$out_1 = +_2^{low}(a_1, b_1, +_1^{high}(a_0, b_0)) \quad (10.2)$$

$$out_2 = +_2^{high}(a_1, b_1, +_1^{high}(a_0, b_0)) \quad (10.3)$$

CODE EX. 10.1: The equation description of the adder

The code 10.2 illustrates the coding result of the test case design (10.1) represented in the 10.3 figure.

```

twoBitsAdder
|tmp c lsb msb |

    c := (Array new: 2).
c at: 1 put: (lsb := SCBlifNode fullAdder).

c at: 2 put: (msb := SCBlifNode fullAdder).

tmp := SCCompositeNode
    iPort: [[tmp| tmp := SCNamedPort new:2.
tmp add: 'b' -> ( [[tmp| tmp := SCUnNamedPort new:2.
tmp add: 1 -> (SCIPin new container: tmp).
tmp add: 2 -> (SCIPin new container: tmp).] value container: tmp).
tmp add: 'a' -> ( [[tmp| tmp := SCUnNamedPort new:2.
tmp add: 1 -> (SCIPin new container: tmp).
tmp add: 2 -> (SCIPin new container: tmp).] value container: tmp).] value
    oPort: [[tmp| tmp := SCNamedPort new:1.
tmp add: 'out' -> ( [[tmp| tmp := SCUnNamedPort new:3.
tmp add: 1 -> (SCOPin new container: tmp).
tmp add: 2 -> (SCOPin new container: tmp).
tmp add: 3 -> (SCOPin new container: tmp).] value container: tmp).] value
    components: c.

(tmp iPort at: 'a') connect: 2 to: 'b' of: lsb.

(tmp iPort at: 'b') connect: 2 to: 'a' of: lsb.

(tmp iPort at: 'b') connect: 1 to: 'a' of: msb.

(tmp iPort at: 'a') connect: 1 to: 'b' of: msb.

lsb
    connect: 'h' to: 'c' of: msb.
lsb
    connect: 'l' to: 3 of: (tmp oPort at: 'out').
msb
    connect: 'h' to: 1 of: (tmp oPort at: 'out').
msb
    connect: 'l' to: 2 of: (tmp oPort at: 'out').
^ tmp

```

CODE EX. 10.2: The two bits adder

10.3.2 Composite nodes and hierarchical visiting

The composite nodes can be flattened in order to visit their sub nodes, as shown in figure 10.5. The 10.5 figure illustrates the translation of the 9.3 example of the section 9, with additional IOs ports. The connections are not any more high level connections (representing signals) but flatten bit-per-bit connections.

In this example t28 represents the *computeSignFor : and :* node, t32 the *computeSignificandFor : and :*, t36 the *computeExponentFor : and :* node, t38 the *normalizeSignificand :* node, and t10 the + node. Note that

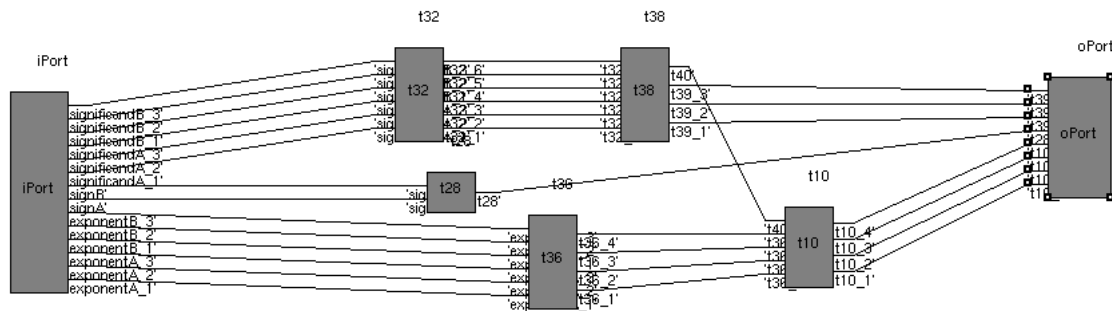


Figure 10.5: Example: By flattening a node, its sub nodes become accessible. This example reflects the 9.3 example of the section 9, with additional IOs ports

t38 owns to outputs, called t40 and t39, as illustrated in the 9.3 figure.

10.3.3 Conclusion

This tool does not claim to be as expressive as the textual grammar the designer might use to code an operator by hand. In particular the regular connections cannot be expressed at this time.

On the other hand, the tool neither allows to automatically tune the operators depending on the inputs values. This can only be realized using the MADEO-FET front end.

Nevertheless, the schematic editor enables to compose existing macros into new ones what offer a way to build libraries of operators. By fully supporting the SCNode formalism, the schematic editor inter operates with the other tools of the MADEO framework, and can, as others, be extended if needed.

Two extensions are envisaged by now: the first one consists in adding a support for regular constructs, and the second one is to support parametrized constructs.

Part III
Appendix

Appendix A

MADEO BET appendix

A.1 Some examples of architectures description using the grammar

A.1.1 First basic example

```
ex1 (
  (ARRAY
    (DOMAIN 1 1 10 10) "END of DOMAIN"
    (
      (COMPOSITE
        (
          (
            (FUNCTION
              (INPUTS
                (
                  ( WIRE ( WIDTH 4 )
                    NAMED in ) " end of NAMED "
                )
                (OUTPUTS
                  (
                    ( WIRE ( WIDTH 1 )
                      NAMED out ) " end of NAMED "
                    )) "END of FUNCTION"
                )
              NAMED f ) " end of NAMED "
            (
              ( WIRE ( WIDTH 10 )
                NAMED h ) " end of NAMED "
            (
              (LINK '(self relativeAt: 0@1) v') "END of LINK"
              NAMED leftH ) " end of NAMED "
            (
              (SWITCHBLOCK
                (RESOURCES
                  (
                    upV
                    leftH
```

```

    h
    v))
'self v connectTo: self h'
'self v connectTo: self upV'
'self v connectTo: self leftH'
'self h connectTo: self upV'
'self h connectTo: self leftH'
'self upV connectTo: self leftH') "END of SWITCHBLOCK"
NAMED switch ) " end of NAMED "

```

```

PRODUCE ExampleSwitch ) " END of PRODUCE "
(
  (LINK '(self relativeAt: 0@1) v') "END of LINK"
  NAMED upV ) " end of NAMED "
(
  ( WIRE ( WIDTH 10) )
  NAMED v ) " end of NAMED "
) "END of ELEMENTS"

```

```

(CONNECTION
'self v connectTo: #in of: self f '
'(self f at:#out ) connectTo: self h'
) "END of CONNECTION"
) "END OF COMPOSITE"

```

```

PRODUCE ExampleTile ) " END of PRODUCE "
) "END of ARRAY"

```

```

PRODUCE ExampleArray ) " END of PRODUCE "

```


A.1.2 Example 2

```

ex2 (
  (ARRAY
    (DOMAIN 1 1 10 10) "END of DOMAIN"
    (
      (COMPOSITE
        (
          (
            (FUNCTION
              (INPUTS
                (
                  ( WIRE ( WIDTH 4 )
                    NAMED in ) " end of NAMED "
                )
              (OUTPUTS
                (
                  ( WIRE ( WIDTH 1 )
                    NAMED out ) " end of NAMED "
                )) "END of FUNCTION"
              REPRESENTATION
                (COLOR red ) "END of COLOR"
                (RECTANGLE 10 10 30 30 ) "END OF RECTANGLE"
            ) "END of REPRESENTATION"
            NAMED f ) " end of NAMED "
          )
          (
            (
              ( WIRE ( WIDTH 10 )
                REPRESENTATION
                  (COLOR red ) "END of COLOR"
                  (LINE 10 10 90 10 PROPORTIONAL 1 ) "END OF LINE"
              ) "END of REPRESENTATION"
              NAMED h ) " end of NAMED "
            )
            (LINK '(self relativeAt: 1@0) h ') "END of LINK"
            NAMED leftH ) " end of NAMED "
          )
          (
            (SWITCHBLOCK
              (RESOURCES
                (
                  upV
                  leftH
                  h
                  v))
              'self v connectTo: self h'
              'self v connectTo: self upV'
              'self v connectTo: self leftH'
              'self h connectTo: self upV'
              'self h connectTo: self leftH'
              'self upV connectTo: self leftH') "END of SWITCHBLOCK"
            NAMED switch ) " end of NAMED "
          )
        )
      )
    )
  )
  PRODUCE ExampleSwitch ) " END of PRODUCE "

```

```

(
  (LINK '(self relativeAt: 0@1) v') "END of LINK"
  NAMED upV ) " end of NAMED "
(
  (
    ( WIRE ( WIDTH 10 )
      REPRESENTATION
        (COLOR red ) "END of COLOR"
        (LINE 10 10 10 90 PROPORTIONAL 1 ) "END OF LINE"

    ) "END of REPRESENTATION"
  NAMED v ) " end of NAMED "
) "END of ELEMENTS"

(CONNECTION
  'self v connectTo: #in of: self f '
  '(self f at:#out ) connectTo: self h'
) "END of CONNECTION"
) "END OF COMPOSITE"
REPRESENTATION

) "END of REPRESENTATION"

PRODUCE ExampleTile ) " END of PRODUCE "
) "END of ARRAY"
PRODUCE ExampleArray ) " END of PRODUCE "

```

A.1.3 Example 3

```

ex3 (
  (ARRAY
    (DOMAIN 1 1 10 10) "END of DOMAIN"
    (
      (COMPOSITE
        (
          (
            (FUNCTION
              (INPUTS
                (
                  ( WIRE ( WIDTH 4 )
                    NAMED in ) " end of NAMED "
                )
              (OUTPUTS
                (
                  ( WIRE ( WIDTH 1 )
                    NAMED out ) " end of NAMED "
                )
              ) "END of FUNCTION"
            )
            REPRESENTATION
            (COLOR red ) "END of COLOR"
            (RECTANGLE 10 10 30 30 ) "END OF RECTANGLE"
          ) "END of REPRESENTATION"
          NAMED f ) " end of NAMED "
        )
        (
          ( WIRE ( WIDTH 10) EXPANDED)
          REPRESENTATION
          (COLOR red ) "END of COLOR"
          (CHANNEL 20 10 90 10 0 1 ) "END OF RECTANGLE"
        )
        ) "END of REPRESENTATION"
        NAMED h ) " end of NAMED "
      )
      (
        (LINK '(self relativeAt: -1@0) h ') "END of LINK"
        NAMED rightH ) " end of NAMED "
      )
      (
        ((
          (SWITCHBLOCK
            (RESOURCES
              (
                upV
                rightH
                h
                v))
            'self v connectTo: self h'
            'self v connectTo: self upV'
            'self v connectTo: self rightH'
            'self h connectTo: self upV'
            'self h connectTo: self rightH'
            'self upV connectTo: self rightH') "END of SWITCHBLOCK"
          NAMED switch ) " end of NAMED "
        )
        REPRESENTATION "-- visual representation of the switch --"
        (COLOR gray ) "END of COLOR"
      )
    )
  )
)

```

```

(h (LINE 20 10 20 20 ) "END OF LINE")
(v (LINE 10 20 20 20 ) "END OF LINE")
(upV (LINE 10 10 20 10 ) "END OF LINE")
(rightH (LINE 10 10 10 20 ) "END OF LINE"))
  PRODUCE ExampleSwitch3 ) " END of PRODUCE "
  (
    (LINK '(self relativeAt: 0@1) v') "END of LINK"
    NAMED upV ) " end of NAMED "
  (
    (
      ( WIRE ( WIDTH 10) EXPANDED)
      REPRESENTATION
      (COLOR red ) "END of COLOR"
      (CHANNEL 10 20 10 90 1 0 ) "END OF RECTANGLE"

    ) "END of REPRESENTATION"
    NAMED v ) " end of NAMED "
  ) "END of ELEMENTS"

(CONNECTION
  'self v connectTo: #in of: self f '
  '(self f at:#out ) connectTo: self h'
  ) "END of CONNECTION"
) "END OF COMPOSITE"
REPRESENTATION

) "END of REPRESENTATION"

PRODUCE ExampleTile3 ) " END of PRODUCE "
) "END of ARRAY"
PRODUCE ExampleArray3 ) " END of PRODUCE "

```

A.1.4 Example 4 with custom representation

```

exo4 (
  (ARRAY
    (DOMAIN 1 1 10 10) "END of DOMAIN"
    (
      (
        (COMPOSITE
          (
            (
              (FUNCTION
                (INPUTS
                  (
                    ( WIRE ( WIDTH 5) EXPANDED )
                    NAMED in ) " end of NAMED "
                  )
                (OUTPUTS
                  (
                    ( WIRE ( WIDTH 1) EXPANDED )
                    NAMED out ) " end of NAMED "
                  )
                )) "END of FUNCTION"
              REPRESENTATION
                (COLOR red ) "END of COLOR"
                (RECTANGLE 30 30 40 40 ) "END OF RECTANGLE"
            ) "END of REPRESENTATION"
          NAMED f ) " end of NAMED "
        )
        (
          (
            ( WIRE ( WIDTH (VALUE myWidth '#(10)') ) EXPANDED )
            REPRESENTATION
              (COLOR red ) "END of COLOR"
              (CHANNEL 20 10 90 10 0 1 ) "END OF CHANNEL"
          ) "END of REPRESENTATION"
        NAMED h ) " end of NAMED "
      )
      (
        (LINK '(self relativeAt: -1@0) h ') "END of LINK"
        NAMED rightH ) " end of NAMED "
      )
    )
  )
  (
    (SWITCHBLOCK
      (RESOURCES
        (
          upV
          rightH
          h
          v))
      'self v connectTo: self h'
      'self v connectTo: self upV'
      'self v connectTo: self rightH'
      'self h connectTo: self upV'
      'self h connectTo: self rightH'
      'self upV connectTo: self rightH') "END of SWITCHBLOCK"
    REPRESENTATION
      (COLOR gray ) "END of COLOR"
  )
)

```

```

( h (LINE 20 10 20 20 ) "END OF LINE")
( rightH (LINE 10 10 10 20 ) "END OF LINE")
( upV (LINE 10 10 20 10 ) "END OF LINE")
( v (LINE 10 20 20 20 ) "END OF LINE")

```

```

) "END of REPRESENTATION"
NAMED switch ) " end of NAMED "

```

```

PRODUCE ExampleSwitch4 ) " END of PRODUCE "
(
  (LINK '(self relativeAt: 0@1) v') "END of LINK"
  NAMED upV ) " end of NAMED "
(
  (
    ( WIRE ( WIDTH (VALUE myWidth)) EXPANDED )
    REPRESENTATION
    (COLOR red ) "END of COLOR"
    (CHANNEL 10 20 10 90 1 0 ) "END OF CHANNEL"
  )
  ) "END of REPRESENTATION"
  NAMED v ) " end of NAMED "
) "END of ELEMENTS"

```

```

(CONNECTION
  'self v connectTo: #in of: self f '
  '(self f at:#out ) connectTo: self h'
) "END of CONNECTION"
) "END OF COMPOSITE"
REPRESENTATION

```

```

(CUSTOM myRepresentationWrittenByHand) "END of CUSTOM"
) "END of REPRESENTATION"

```

```

PRODUCE ExampleTile4 ) " END of PRODUCE "
) "END of ARRAY"
PRODUCE ExampleArray4 ) " END of PRODUCE "

```

A.1.5 LPPGA

```

(
  (ARRAY
    (DOMAIN 1 1 20 20) "END of DOMAIN"
    (
      (COMPOSITE
        (
          "----- declaration of the function: f-----"
          (
            (FUNCTION
              (INPUTS a1 a2 a3 b1 b2 )
              (OUTPUTS o1 o2 o3 )) "END of FUNCTION"
            REPRESENTATION "-- visual representation of the function --"
          )
        )
      )
    )
  )
)

```

A.1. SOME EXAMPLES OF ARCHITECTURES DESCRIPTION USING THE GRAMMAR 127

```

        (COLOR red ) "END of COLOR"
        (RECTANGLE 50 50 70 70 ) "END OF RECTANGLE"
        (TEXT 60 40 'function output') "END of TEXT"
    ) "END of REPRESENTATION"
NAMED f ) " end of NAMED "

"----- declaration of the vertical wire: south -----"
(
    (
        ( WIRE ( WIDTH 5) EXPANDED )
        REPRESENTATION "-- visual representation of the wire --"
        (COLOR gray ) "END of COLOR"
        (CHANNEL 14 34 14 90 4 0 ) "END OF CHANNEL"
    ) "END of REPRESENTATION"
NAMED south ) " end of NAMED "

"----- declaration of the horizontal wire: east -----"
(
    (
        ( WIRE ( WIDTH 5) EXPANDED )
        REPRESENTATION "-- visual representation of the wire --"
        (COLOR gray ) "END of COLOR"
        (CHANNEL 34 14 90 14 0 4 ) "END OF CHANNEL"
    ) "END of REPRESENTATION"
NAMED east ) " end of NAMED "

"----- declaration of the horizontal wire: west -----"
(
    (LINK '(self relativeAt: -1@0) east') "END of LINK"
NAMED west ) " end of NAMED "

"----- declaration of the vertical wire: north -----"
(
    (LINK '(self relativeAt: 0@1) south') "END of LINK"
NAMED north ) " end of NAMED "

"----- wires for connections between a1 and o1 -----"
"-- wire for diagonal connections --"
(
    ( WIRE ( WIDTH 1) )
NAMED a1North ) " end of NAMED "
"-- wire for vertical and horizontal connections --"
(
    ( WIRE ( WIDTH 1) )
NAMED a1South ) " end of NAMED "

"----- wires for connections between b1 and o2 -----"
"-- wire for diagonal connections --"
(
    ( WIRE ( WIDTH 1) )
NAMED a3North ) " end of NAMED "
"-- wire for vertical and horizontal connections --"
(
    ( WIRE ( WIDTH 1) )
NAMED a3South ) " end of NAMED "

"----- wires for connections between a3 and o3 -----"

```

```

"-- wire for diagonal connections --"
(
  ( WIRE ( WIDTH 1 ) )
  NAMED b1North ) " end of NAMED "
"-- wire for vertical and horizontal connections --"
(
  ( WIRE ( WIDTH 1 ) )
  NAMED b1South ) " end of NAMED "

"----- declaration of the switch: switch-----"
(
  (
    (
      (SWITCHBLOCK
        (RESOURCES
          (
            south
            east
            north
            west
          )
        )
        'self north connectTo: self east'
        'self north connectTo: self south'
        'self north connectTo: self west'
        'self west connectTo: self east'
        'self west connectTo: self south'
        'self east connectTo: self south'
      ) "END of SWITCHBLOCK"
      REPRESENTATION "-- visual representation of the switch --"
      (COLOR gray ) "END of COLOR"
      ( north (LINE 14 10 34 10 ) "END OF LINE")
      ( south (LINE 14 34 34 34 ) "END OF LINE")
      ( west (LINE 10 14 10 34 ) "END OF LINE")
      ( east (LINE 34 14 34 34 ) "END OF LINE")
    ) "END of REPRESENTATION"
    NAMED switch ) " end of NAMED "
    PRODUCE MySwitchLPPGA ) " END of PRODUCE "
    CATEGORY LPPGA ) "END of CATEGORY"

) "END of ELEMENTS"

"----- declaration of vertical, horizontal and diagonal ----"
"----- connections between cells -----"
"-- declaration of connections between inputs and wires --"
(CONNECTION
'self a1North connectTo: #a1 of: self f'
'self a1South connectTo: #a1 of: self f'
'self a3North connectTo: #a3 of: self f'
'self a3South connectTo: #a3 of: self f'
'self b1North connectTo: #b1 of: self f'
'self b1South connectTo: #b1 of: self f'
) "END of CONNECTION"

"-- declaration of connections between outputs and wires --"
(CONNECTION

```


A.1. SOME EXAMPLES OF ARCHITECTURES DESCRIPTION USING THE GRAMMAR 129

```

    '(self relativeAt: -1 @ 1) f connect: #o1 to: self a1North'
    '(self relativeAt: 1 @ 1) f connect: #o1 to: self a1North'
    '(self relativeAt: 1 @ -1) f connect: #o1 to: self a1North'
    '(self relativeAt: -1 @ -1) f connect: #o1 to: self a1North'
    '(self relativeAt: 0 @ 1) f connect: #o1 to: self a1South'
    '(self relativeAt: 1 @ 0) f connect: #o1 to: self a1South'
    '(self relativeAt: 0 @ -1) f connect: #o1 to: self a1South'
    '(self relativeAt: -1 @ 0) f connect: #o1 to: self a1South'

    '(self relativeAt: -1 @ 1) f connect: #o2 to: self b1North'
    '(self relativeAt: 1 @ 1) f connect: #o2 to: self b1North'
    '(self relativeAt: 1 @ -1) f connect: #o2 to: self b1North'
    '(self relativeAt: -1 @ -1) f connect: #o2 to: self b1North'
    '(self relativeAt: 0 @ 1) f connect: #o2 to: self b1South'
    '(self relativeAt: 1 @ 0) f connect: #o2 to: self b1South'
    '(self relativeAt: 0 @ -1) f connect: #o2 to: self b1South'
    '(self relativeAt: -1 @ 0) f connect: #o2 to: self b1South'

    '(self relativeAt: -1 @ 1) f connect: #o3 to: self a3North'
    '(self relativeAt: 1 @ 1) f connect: #o3 to: self a3North'
    '(self relativeAt: 1 @ -1) f connect: #o3 to: self a3North'
    '(self relativeAt: -1 @ -1) f connect: #o3 to: self a3North'
    '(self relativeAt: 0 @ 1) f connect: #o3 to: self a3South'
    '(self relativeAt: 1 @ 0) f connect: #o3 to: self a3South'
    '(self relativeAt: 0 @ -1) f connect: #o3 to: self a3South'
    '(self relativeAt: -1 @ 0) f connect: #o3 to: self a3South'
) "END of CONNECTION"

"----- declaration of connections between outputs function ----"
"----- and wires ----"
(CONNECTION
  'self f connect: #o1 to: (self relativeAt: 1@0) south'
  'self f connect: #o2 to: self south'
  'self f connect: #o3 to: (self relativeAt: 0@-1) east'
) "END of CONNECTION"

"----- declaration of connections between inputs function ----"
"----- and wires ----"
(CONNECTION
  '(self relativeAt:1@0) south connectTo: #a1 of: self f'
  'self east connectTo: #a2 of: self f'
  'self south connectTo: #a3 of: self f'
  '(self relativeAt:0@-1) east connectTo: #b1 of: self f'
  'self east connectTo: #b2 of: self f'
) "END of CONNECTION"

) "END OF COMPOSITE"

PRODUCE LPPGACell ) " END of PRODUCE "
CATEGORY LPPGA ) "END of CATEGORY"
) "END of ARRAY"
PRODUCE LPPGAArray ) " END of PRODUCE "
CATEGORY LPPGA ) "END of CATEGORY"

```

A.2 Grammar

anytype ::= ((word)— (((string)— (((number)— (((ivariableparameter_i)— (ivariableparametercall_i))))))))

array ::= '(iarray_i [iinterface_i] idomain_i jelement_i + . [imagesize])'

arrayproduceclass ::= ((iarrayproduceclasswithcategory_i)— (iarrayproduceclasswithoutcategory_i))

arrayproduceclasswithcategory ::= '((iarray_i 'PRODUCE' word)' 'CATEGORY' word)'

arrayproduceclasswithoutcategory ::= '(iarray_i 'PRODUCE' word)'

atom ::= ((iswitchblock_i)— ((iswitch_i)— (((ifunction_i)— ((iwire_i)— (((imultiplexer_i)— (((iregister_i)— (((ireference_i)— (((itristate_i)— (ilink_i))))))))))))))))

basecost ::= '(ibasecost_i number)'

channel ::= '(ichannel_i number number number number [number number])'

clock ::= '(iclock_i word)'

color ::= '(icolor_i word)'

command ::= '(icommand_i word)'

commonrepresentation ::= irepresentation_i [icolor_i] [igeometric_i] [isquale_i] [itext_i] [ioffset_i] [icustom_i]

composite ::= '(icomposite_i [iinterface_i] [iresources_i] '(iflatten_i)— jelement_i + .)' icconnecting_i + . [imagesize])' [itext_i]

concreteswitch ::= '(iswitch_i ((iinputwidth_i ipositions_i)— ipositions_i))'

connecting ::= ((iconnection_i)— ipip_i)

connection ::= '(iconnection_i string +)'

custom ::= '(icustom_i word)'

domain ::= '(idomain_i number number number number)'

donotproduceclass ::= ((inamedelement_i)— iunamedelement_i) [iparameter_i]

```

element ::= ((iproduceclassi )— idonotproduceclassi )[iparameteri ]

flatten ::= '(' iflatteni jelementi ')'

function ::= '(' ifunctioni iinputsi ioutputsi [ipossiblefunctionsi ] ')'

functionalswitch ::= '(' iswitchi iinputwidthi (((('BY_OUTPUT' ioutputresourcei + )—
('BY_INPUT' iinputresourcei + )))— (((('BY_OUTPUT' resource + )— ('BY_INPUT'
iinputresourcei + )))' )' )

geometric ::= ((irectanglei )— (((ilinei )— (((ichanneli )— (((ihorizontalchanneli )—
iverticalchanneli ))))))))

horizontalchannel ::= '(' ihorizontalchanneli number number number number ')'

input ::= '(' iinputi isignali ')'

inputresource ::= '(' iinputi ((string )— word )'TO' '(' (string )— word + ')' ')'

inputs ::= '(' iinputsi isignalsi ')'

inputwidth ::= '(' iinputwidthi number "x" number "y" ')'

interface ::= '(' iinterfacei isignalsi ')'

iob ::= '(' ijobi '(' (iopadi )— jelementi + . ')' iconnectingi + . ')'

iobs ::= '(' ijobsi '('

iopad ::= '(' iopadi iinputsi ioutputsi ')'

line ::= '(' ilinei number number number number ['PROPORTIONAL' ] ')'

link ::= '(' ilinki string ['IFNONE' jelementi ] ')'

method ::= word ((iarrayproduceclassi )— iproduceclassi )

multiplexer ::= '(' imultiplexeri iinputsi ioutputi ')'

namedelement ::= ((inamedelementwithrepresentationi )— (inamedelementwithoutrepresentationi ))

namedelementwithoutrepresentation ::= '(' inamedelementi 'NAMED' word ')'

namedelementwithrepresentation ::= '(' inamedelementwithoutrepresentationi irepresentationi ')'

```

namedwire ::= '(? |wire_i 'NAMED' word)'
 offset ::= '(? |offset_i number number)'
 output ::= '(? |output_i |signal_i)'
 outputresource ::= '(? |output_i ((string)— word)'FROM' '(? (string)— word +)')'
 outputs ::= '(? |outputs_i |signals_i)'
 parameter ::= '(? 'SET' word ((number)— word)?)' [|parameter_i]
 pip ::= '(? |pip_i string string string string)'
 pipdefinition ::= '(? '(? string +)' 'PRODUCE' word)'
 piptype ::= '(? |piptype_i ((|reference_i)— |pipdefinition_i)?)'
 position ::= '(? |position_i number number |piptype_i)'
 positions ::= '(? |position_i +)'
 possiblefunctions ::= '(? 'AMONG' word +)'
 produceclass ::= ((|produceclasswithcategory_i)— (|produceclasswithoutcategory_i))
 produceclasswithcategory ::= '(? '(? |donotproduceclass_i 'PRODUCE' word)' 'CATEGORY' word)'
 produceclasswithoutcategory ::= '(? |donotproduceclass_i 'PRODUCE' word)'
 rectangle ::= '(? |rectangle_i number number number number)'
 reference ::= '(? |reference_i word)'
 register ::= '(? |register_i |input_i |output_i [|type_i] [|clock_i])'
 representation ::= ((|switchrepresentation_i)— (|commonrepresentation_i))
 resources ::= '(? |resources_i '(? word +)')'
 signal ::= (((word)— string)— (|namedwire_i))
 signals ::= |signal_i +

squale ::= '(' i_squale_i number)'

switch ::= ((i_concreteswitch_i)— (i_functionalswitch_i))

switchblock ::= '(' i_switchblock_i ((i_resources_i string +)— string +))'

switchrepresentation ::= i_representation_i i_color_i [i_color_i] '(' word i_line_i)' '(' word i_line_i)' + .

text ::= '(' i_text_i number number string)'

tristate ::= '(' i_tristate_i i_input_i i_output_i [i_command_i])'

type ::= '(' i_type_i word)'

unamedelement ::= ((i_unamedelementwithrepresentation_i)— (i_unamedelementwithoutrepresentation_i))

unamedelementwithoutrepresentation ::= ((i_composite_i)— (((i_job_i)— (((i_array_i)— (i_atom_i))))))

unamedelementwithrepresentation ::= '(' i_unamedelementwithoutrepresentation_i i_representation_i)'

variableparameter ::= '(' 'VALUE' word string)'

variableparametercall ::= '(' 'VALUE' word)'

verticalchannel ::= '(' i_verticalchannel_i number number number number)'

width ::= '(' i_width_i i_anytype_i)'

wire ::= '(' i_wire_i i_width_i [i_basecost_i] ['EXPANDED'] [i_basecost_i])'

A.3 Heterogeneous architectures

Defining an heterogeneous architecture goes through defining several domains. Every domain is associated with an element to be replicated; all the domains are rectangular zones of the architecture.

The following code example illustrates the creation of such an architecture. In a concern of clarity the elements are described through references to pre-existing classes.

```
LPPGACell (
  (
    (ARRAY
      (DOMAIN 1 1 10 20) "END of DOMAIN"
      (REFERENCE FirstElement) "END of REFERENCE"
      (DOMAIN 11 1 20 20) "END of DOMAIN"
      (REFERENCE SecondElement) "END of REFERENCE"
      (DOMAIN 21 1 30 10) "END of DOMAIN"
      (REFERENCE ThirdElement) "END of REFERENCE"
    ) "END of ARRAY"
    PRODUCE NewArray ) "END of PRODUCE "
    CATEGORY LPPGA ) "END of CATEGORY"
  )

```

CODE EX. A.1: An heterogeneous architecture

The textual description presented in the table A.1 produces the method **new** (code A.2).

```
new
  "This method was generated by UIDefiner. Any edits made here
  may be lost whenever methods are automatically defined."

  | col elts |
  elts := (1 to: 20)
    collect: [:aY | (1 to: 30)
      collect: [:aX | RaNilObject position: aX @ aY]].
  1 to: 20 do: [:y | 1 to: 10 do: [:x | (elts at: y)
    at: x put: (FirstElement new position: x @ y)].
  1 to: 20 do: [:y | 11 to: 20 do: [:x | (elts at: y)
    at: x put: (SecondElement new position: x @ y)].
  1 to: 10 do: [:y | 21 to: 30 do: [:x | (elts at: y)
    at: x put: (ThirdElement new position: x @ y)].
  col := OrderedCollection new.
  elts do: [:a | col addAll: a].
  ^self
    x: 30
    y: 20
    with: col)
  addInterface: nil

```

CODE EX. A.2: The **new** method generated from A.1

A.4 Inspecting changes over the definition of an architecture

The prospection mechanism impacts the architectural description as illustrated by the following figure.

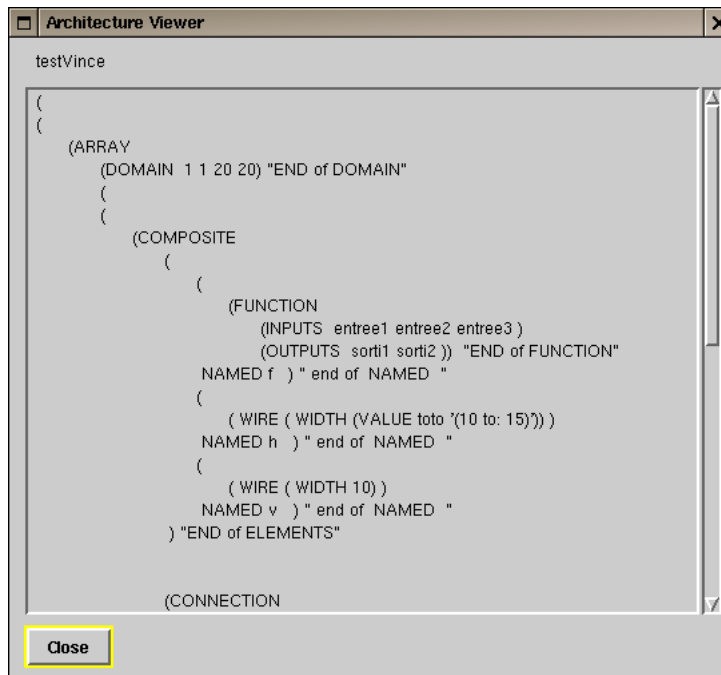


Figure A.1: Architecture viewer

A.5 Textual output

The place&route stage results in configuring the resources of the model. This information can be written into a file (cf codes A.3, A.4) using the modules'menu.

This information can be used on demand to produce a bitstream, assuming the full bitstream structure is known.

```
(
'carrierCntrl' ( instance1 ) over LPPGAArray

%
Placement
%
clock          12 10  f
data_valid     13 13  f
dv_regx1x     13 10  f
first_stage_hold 13 11  f
inter_hold     8 15   f
inv_pilot      10 10  f
mc_dv          13 14  f
pilot          12 15  f
prbs_hold      12 11  f
reset          12 14  f
U10            11 10  f
U11            8 10   f
U12            8 12   f
U15            10 13  f
U16            9 14   f
U17            11 14  f
U18            11 13  f
U20            10 15  f
U21            10 14  f
U22            11 12  f
U24            10 11  f
U26            9 10   f
U28            12 12  f
U29            13 15  f
U32            9 13   f
U33            11 11  f
U34            8 14   f
U36            9 11   f
U41            8 11   f
U45            9 15   f
U46            9 12   f
U51            10 12  f
zeros          11 15  f
```

CODE EX. A.3: Placement information

```

%
Routing
%

carrCnt_2_ :
11 12      #f_#o1
10 13      10@13_#a1North
10 13      #f_#a1
11 12      #f_#o1
12 12      #south_4
12 13      #south_4
12 14      #south_4
11 14      #east_4
10 14      #east_4
9 14       #east_4
9 14       #f_#b2
11 12      #f_#o1
12 12      #south_4
12 13      #south_4
11 13      #f_#a1
11 12      #f_#o1
12 12      #south_4
12 13      #south_4
12 14      #south_4
11 14      #east_4
10 14      #east_4
9 14       #east_4
8 14       #east_4
8 14       #f_#a2
11 12      #f_#o1
12 12      #south_4
11 12      #east_4
10 12      #east_4
9 12       #east_4
9 12       #south_4
9 11       #south_4
8 11       #f_#a1

data_valid :
13 10      #f_#o1
14 10      #south_0
14 11      #south_0
etc ...

```

CODE EX. A.4: Routing information

A.6 Changing the cost function of the placer

Every routing resource owns a cost that is used to compute the global routing cost. However during placement, the placer uses an approximation called 'the cost function' to determine the quality of a solution. This approximation serves as a basis for the annealing schedule.

As this function drives the behavior of the placer, replacing the function results in modifying the placer behavior, what is needed when finely tuning the placer to match a given architecture's requirements.

This goes through implementing a specific instance method in the class modeling the array. This class inherits from the `RaArrayedObject` in which a common method exists.

`evaluatorBlock`

```

^
[:thePlacerRouter :aRoute |
 | allX allY pos sourcePosition |
 allX := SortedCollection new.
 allY := SortedCollection new.
 aRoute destinations
   do:
     [:aDestination |
      pos := thePlacerRouter positionOf: aDestination graphNode.
      allX add: pos x.
      allY add: pos y].
 sourcePosition := thePlacerRouter positionOf: aRoute source graphNode.
 allX add: sourcePosition x.
 allY add: sourcePosition y.
 allX last - allX first + allY last - allY first]

```

CODE EX. A.5: Example of cost function

This method must return a smalltalk block owning two parameters. The first parameter is the placer-router which retains the placement information, as the placement is not effective yet and cannot be accessed through the model itself. The last parameter is the route which cost is to be computed. The block must return a number.

A.7 Prospection results

The following four figures represent different criteria that have been analyzed with regard to the same circuit, letting the channels' width vary from three to height.

This enables to determine the minimum channel width that ensures the circuit is routable, and to balance the hardware resource saving/ cpu time tradeoff.

The bounding box's growth indicates that the routing cannot be bounded to the placement bounding box for the modules.

The routing cost only takes into account the signals that have been successfully routed so that an increase in the number of unrouted signal can result in a reduction of the global routing cost.

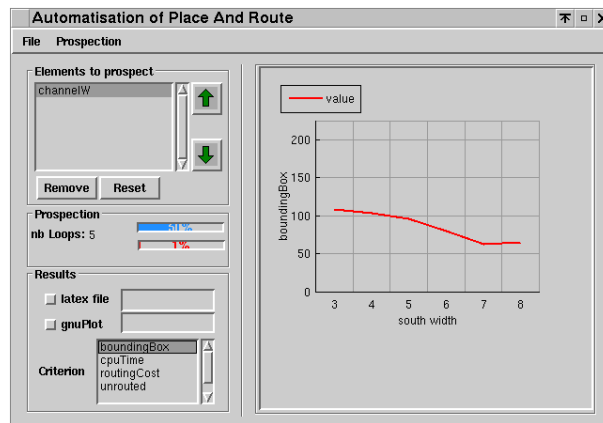


Figure A.2: Analyzing the bounding box

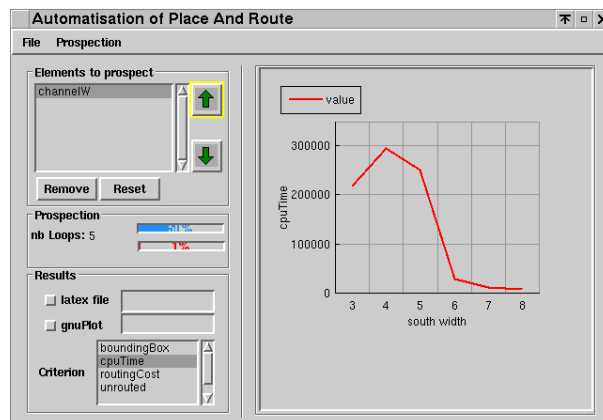


Figure A.3: Analyzing the CPU time

When several criteria are jointly selected, the order of the parameters within the list determines the X label of the chart (see figure A.7).

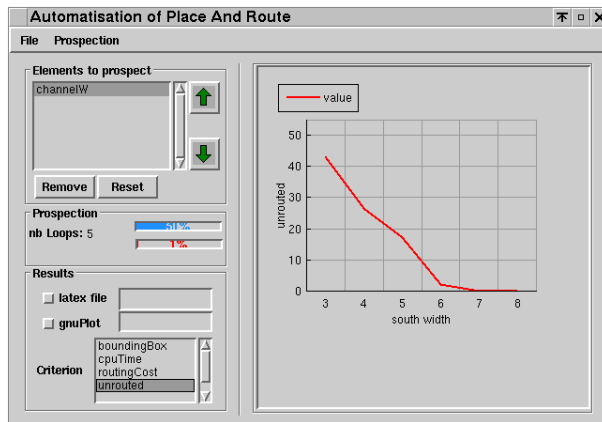


Figure A.4: Analyzing the number of unrouted signals

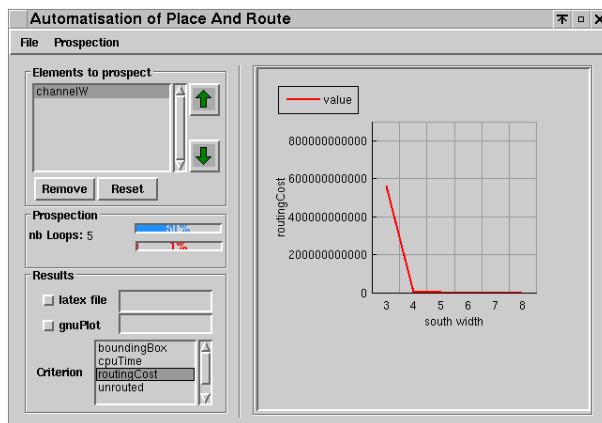


Figure A.5: Analyzing the routing cost

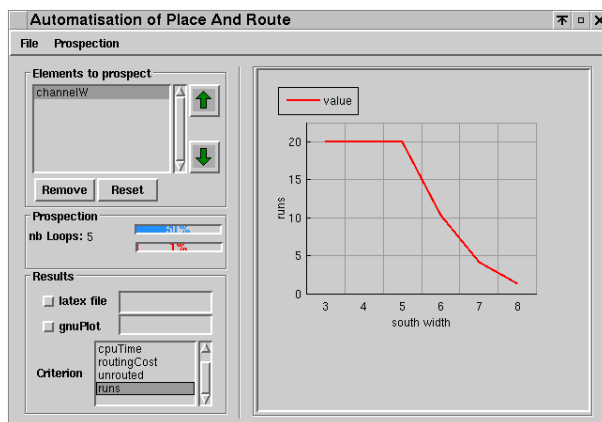


Figure A.6: Analyzing the number of runs (20 max)

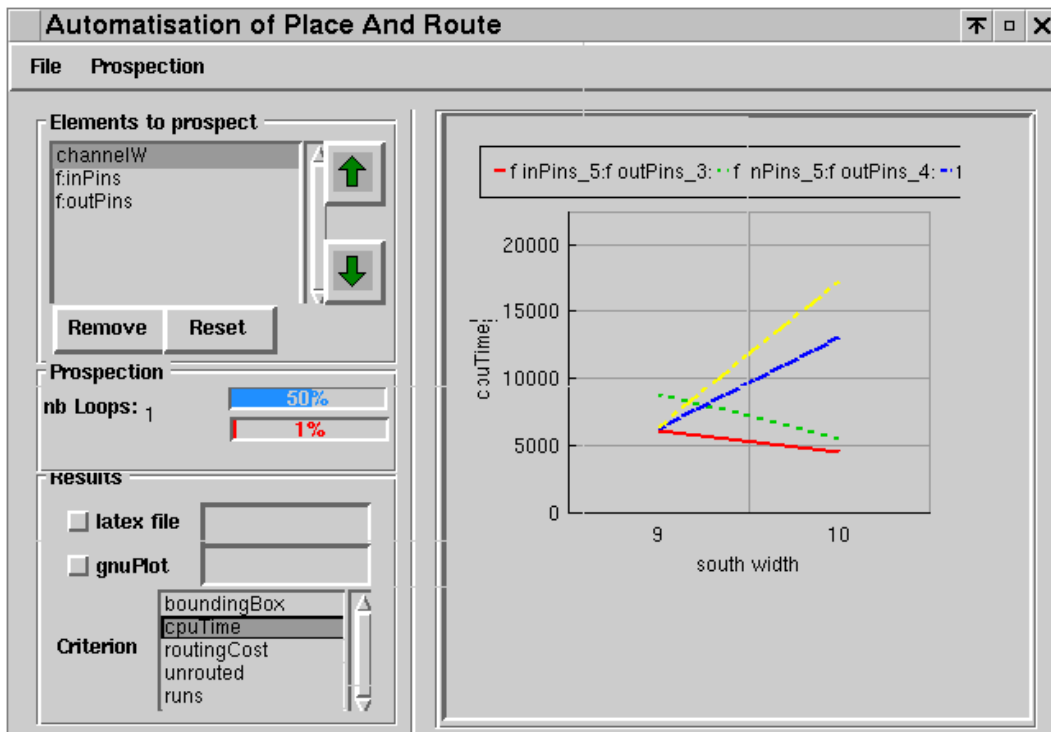


Figure A.7: Analyzing several criteria

A.8 Example of prospection

Architecture: LPPGACell3

File: carrierCntrl-SnglOp.edif

Number of Loop: 5

Result on boundingBox,cpuTime,routingCost,unrouted,runs

Number Of Pins: inputs 1, outputs 1

A.8.1 History of prospecting result

Iteration : 1

'channelW'	'value'
3	5.46518e11
4	6.62941e9
5	3.08402e9
6	2.17366e6
7	9477.4
8	2715.5

Iteration : 2

'channelW'	'value'
3	5.26826e11
4	5.74228e9
5	3.83936e9
6	1.02325e9
7	5450.6
8	2536.0

Iteration : 3

'channelW'	'value'
3	5.23947e11
4	6.99895e9
5	4.56927e9
6	4996.6
7	4011.5
8	2841.6

Iteration : 4

'channelW'	'value'
3	5.1847e11
4	6.89081e9
5	6.36456e9
6	21240.8
7	2613.3
8	3083.7

Iteration : 5

'channelW'	'value'
3	6.8288e11
4	4.07851e9
5	2.71565e9
6	60279.5
7	2857.3
8	2036.0

A.8.2 Average of prospecting result

'channelW'	'value'
3	5.59728e11
4	6.06799e9
5	4.11457e9
6	2.05103e8
7	4882.02
8	2642.56

Appendix B

GenDoc results

This chapter illustrates the results of the GenDoc process through an example.

Report

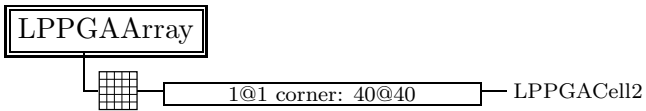
Created by loic lagadec | loic.lagadec@univ-brest.fr (July 5, 2004)

B.1 LPPGAArray

- Need : LPPGA Cell2

- Used by : none

B.1.1 Architecture Design (Level 1)



B.2 LPPGACell2

- **Need** : MySwitchLPPGA
- **Used by** : LPPGAArray

CONNECTIONS

```

self a1North connectTo: a1 of: self f
self a1South connectTo: a1 of: self f
self a3North connectTo: a3 of: self f
self a3South connectTo: a3 of: self f
self b1North connectTo: b1 of: self f
self b1South connectTo: b1 of: self f
(self relativeAt: -1 @ 1) f connect: o1 to: self a1North
(self relativeAt: 1 @ 1) f connect: o1 to: self a1North
(self relativeAt: 1 @ -1) f connect: o1 to: self a1North
(self relativeAt: -1 @ -1) f connect: o1 to: self a1North
(self relativeAt: 0 @ 1) f connect: o1 to: self a1South
(self relativeAt: 1 @ 0) f connect: o1 to: self a1South
(self relativeAt: 0 @ -1) f connect: o1 to: self a1South
(self relativeAt: -1 @ 0) f connect: o1 to: self a1South
(self relativeAt: -1 @ 1) f connect: o2 to: self b1North
(self relativeAt: 1 @ 1) f connect: o2 to: self b1North
(self relativeAt: 1 @ -1) f connect: o2 to: self b1North
(self relativeAt: -1 @ -1) f connect: o2 to: self b1North
(self relativeAt: 0 @ 1) f connect: o2 to: self b1South
(self relativeAt: 1 @ 0) f connect: o2 to: self b1South
(self relativeAt: 0 @ -1) f connect: o2 to: self b1South
(self relativeAt: -1 @ 0) f connect: o2 to: self b1South
(self relativeAt: -1 @ 1) f connect: o3 to: self a3North
(self relativeAt: 1 @ 1) f connect: o3 to: self a3North
(self relativeAt: 1 @ -1) f connect: o3 to: self a3North
(self relativeAt: -1 @ -1) f connect: o3 to: self a3North
(self relativeAt: 0 @ 1) f connect: o3 to: self a3South
(self relativeAt: 1 @ 0) f connect: o3 to: self a3South
(self relativeAt: 0 @ -1) f connect: o3 to: self a3South
(self relativeAt: -1 @ 0) f connect: o3 to: self a3South
self f connect: o1 to: (self relativeAt: 1@0) south
self f connect: o2 to: self south
self f connect: o3 to: (self relativeAt: 0@-1) east
(self relativeAt:1@0) south connectTo: a1 of: self f
self east connectTo: a2 of: self f
self south connectTo: a3 of: self f
(self relativeAt:0@-1) east connectTo: b1 of: self f
self east connectTo: b2 of: self f

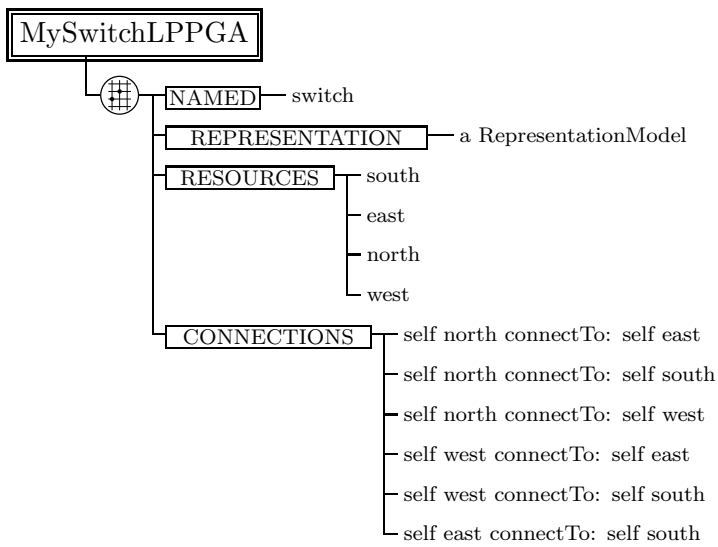
```

B.3 MySwitchLPPGA

- Need : none

- Used by : LPPGACell2

B.3.1 Architecture Design (Level 3)



ANNEXES

ANNEXES 1: Sources LPPGACell2

```

LPPGACell2 "
  Method: LPPGACell2
  Date: lundi 17 mai 2004
  Defines classes :
    - LPPGArray
    - LPPGACell2
    - MySwitchLPPGA
"

(
(
  (ARRAY
    (DOMAIN 1 1 40 40) "END of DOMAIN"
    (
    (
      (COMPOSITE
        (
          (
            (FUNCTION
              (INPUTS
a1 a2 a3 b1 b2          )
              (OUTPUTS
o1 o2 o3                )) "END of FUNCTION"
            REPRESENTATION
              (DEFAULTCOLOR veryLightGray ) "END of DEFAULTCOLOR"
              (COLOR red ) "END of COLOR"
              (RECTANGLE 50 50 70 70 ) "END OF RECTANGLE"
              (TEXT 60 40 'function name') "END of TEXT"
              (CUSTOM drawPerso) "END of CUSTOM"
            ) "END of REPRESENTATION"
            NAMED f ) " end of NAMED "
          (
            (
              (FUNCTION
                (INPUTS
a1 a2 a3 b1 b2          )
                (OUTPUTS
o1 o2 o3                )) "END of FUNCTION"
              REPRESENTATION
                (DEFAULTCOLOR veryLightGray ) "END of DEFAULTCOLOR"
                (COLOR red ) "END of COLOR"

```



```

        (RECTANGLE 50 50 70 70 ) "END OF RECTANGLE"
        (TEXT 60 40 'function name') "END of TEXT"
        (CUSTOM drawPerso) "END of CUSTOM"
    ) "END of REPRESENTATION"
NAMED f    ) " end of  NAMED  "
(
    (
        ( WIRE ( WIDTH (VALUE width '#(10)')) EXPANDED )
        REPRESENTATION
        (DEFAULTCOLOR veryLightGray ) "END of DEFAULTCOLOR"
        (COLOR blue ) "END of COLOR"
        (CHANNEL 14 38 14 90 2 0 ) "END OF CHANNEL"

    ) "END of REPRESENTATION"
NAMED south    ) " end of  NAMED  "
(
    (
        ( WIRE ( WIDTH (VALUE width)) EXPANDED )
        REPRESENTATION
        (DEFAULTCOLOR veryLightGray ) "END of DEFAULTCOLOR"
        (COLOR blue ) "END of COLOR"
        (CHANNEL 38 14 90 14 0 2 ) "END OF CHANNEL"

    ) "END of REPRESENTATION"
NAMED east    ) " end of  NAMED  "
(
    (LINK '(self relativeAt: -1@0) east') "END of LINK"
NAMED west    ) " end of  NAMED  "
(
    (LINK '(self relativeAt: 0@1) south') "END of LINK"
NAMED north    ) " end of  NAMED  "
(
    ( WIRE ( WIDTH 1 ) )
NAMED a1North    ) " end of  NAMED  "
(
    ( WIRE ( WIDTH 1 ) )
NAMED a1South    ) " end of  NAMED  "
(
    ( WIRE ( WIDTH 1 ) )
NAMED a3North    ) " end of  NAMED  "
(
    ( WIRE ( WIDTH 1 ) )
NAMED a3South    ) " end of  NAMED  "
(
    ( WIRE ( WIDTH 1 ) )
NAMED b1North    ) " end of  NAMED  "

```

```

(
  ( WIRE ( WIDTH 1 ) )
  NAMED b1South ) " end of NAMED "
(
(
(
(
  (SWITCHBLOCK
    (RESOURCES
      (
        south
        east
        north
        west))
      'self north connectTo: self east'
      'self north connectTo: self south'
      'self north connectTo: self west'
      'self west connectTo: self east'
      'self west connectTo: self south'
      'self east connectTo: self south') "END of SWITCHBLOCK"
    REPRESENTATION
      (COLOR gray ) "END of COLOR"
      ( east (LINE 38 14 38 34 ) "END OF LINEX")
      ( north (LINE 14 10 34 10 ) "END OF LINEX")
      ( south (LINE 14 38 34 38 ) "END OF LINEX")
      ( west (LINE 10 14 10 34 ) "END OF LINEX")

      ) "END of REPRESENTATION"
    NAMED switch ) " end of NAMED "

  PRODUCE MySwitchLPPGA ) " END of PRODUCE "
  CATEGORY LPPGA ) "END of CATEGORY"

) "END of ELEMENTS"

(CONNECTION
  'self a1North connectTo: #a1 of: self f'
  'self a1South connectTo: #a1 of: self f'
  'self a3North connectTo: #a3 of: self f'
  'self a3South connectTo: #a3 of: self f'
  'self b1North connectTo: #b1 of: self f'
  'self b1South connectTo: #b1 of: self f'
) "END of CONNECTION"

(CONNECTION

```

```

'(self relativeAt: -1 @ 1) f connect: #o1 to: self a1North'
'(self relativeAt: 1 @ 1) f connect: #o1 to: self a1North'
'(self relativeAt: 1 @ -1) f connect: #o1 to: self a1North'
'(self relativeAt: -1 @ -1) f connect: #o1 to: self a1North'
'(self relativeAt: 0 @ 1) f connect: #o1 to: self a1South'
'(self relativeAt: 1 @ 0) f connect: #o1 to: self a1South'
'(self relativeAt: 0 @ -1) f connect: #o1 to: self a1South'
'(self relativeAt: -1 @ 0) f connect: #o1 to: self a1South'
'(self relativeAt: -1 @ 1) f connect: #o2 to: self b1North'
'(self relativeAt: 1 @ 1) f connect: #o2 to: self b1North'
'(self relativeAt: 1 @ -1) f connect: #o2 to: self b1North'
'(self relativeAt: -1 @ -1) f connect: #o2 to: self b1North'
'(self relativeAt: 0 @ 1) f connect: #o2 to: self b1South'
'(self relativeAt: 1 @ 0) f connect: #o2 to: self b1South'
'(self relativeAt: 0 @ -1) f connect: #o2 to: self b1South'
'(self relativeAt: -1 @ 0) f connect: #o2 to: self b1South'
'(self relativeAt: -1 @ 1) f connect: #o3 to: self a3North'
'(self relativeAt: 1 @ 1) f connect: #o3 to: self a3North'
'(self relativeAt: 1 @ -1) f connect: #o3 to: self a3North'
'(self relativeAt: -1 @ -1) f connect: #o3 to: self a3North'
'(self relativeAt: 0 @ 1) f connect: #o3 to: self a3South'
'(self relativeAt: 1 @ 0) f connect: #o3 to: self a3South'
'(self relativeAt: 0 @ -1) f connect: #o3 to: self a3South'
'(self relativeAt: -1 @ 0) f connect: #o3 to: self a3South'
) "END of CONNECTION"

```

```
(CONNECTION
```

```

'self f connect: #o1 to: (self relativeAt: 1@0) south'
'self f connect: #o2 to: self south'
'self f connect: #o3 to: (self relativeAt: 0@-1) east'
) "END of CONNECTION"

```

```
(CONNECTION
```

```

'(self relativeAt:1@0) south connectTo: #a1 of: self f'
'self east connectTo: #a2 of: self f'
'self south connectTo: #a3 of: self f'
'(self relativeAt:0@-1) east connectTo: #b1 of: self f'
'self east connectTo: #b2 of: self f'
) "END of CONNECTION"
) "END OF COMPOSITE"

```

```

PRODUCE LPPGACell2 ) " END of PRODUCE "
CATEGORY LPPGA ) "END of CATEGORY"
) "END of ARRAY"
PRODUCE LPPGAArray ) " END of PRODUCE "
CATEGORY LPPGA ) "END of CATEGORY"

```


Appendix C

Madeo fet appendix

C.1 Blif vs Edif description

C.1.1 smalltalk code

The differences between blif and edif are illustrated through a simple example. The code is as follows:

```
a:a b:b c:c
```

```
^(self plus:a plus:b) + (self plus:b plus:c)
```

where *plus : plus* is a method performing a simple addition, but allowing to keep a structural decomposition. The typing is extremely simple and symmetrical to preserve the readability:

```
TYTypeInterval from: 1 to: 4
```

What can be noticed is that the EDIF description does not require any type inference as it makes use of references to operators. These operators are based on the IOs of the nodes of the graph, with compaction as several nodes linked to the same message result in a single EDIF CELL which is instantiated several times.

On the opposite, the BLIF description cannot be obtained without going through the type inference process. The designer is free to get a hierarchical or a flatten BLIF. The flatten BLIF might be more compact but drives a need to more computing power.

C.1.2 Resulting EDIF

```
(edif testEdif
(edifVersion 2 0 0)
(edifLevel 0)
(keywordMap (keywordLevel 0))
(status
(written(timestamp 7 5 2004 3 54 3)
```



```
(direction INPUT) )
(port c
(direction INPUT) )
(port t9
(direction OUTPUT) )
)
(contents
(instance t5
(viewRef myView (cellRef plus:plus:
(libraryRef myLibrary))
)
)
(instance t7
(viewRef myView (cellRef plus:plus:
(libraryRef myLibrary))
)
)
(instance t4
(viewRef myView (cellRef +
(libraryRef myLibrary))
)
)
)
(net t5
(joined
(portRef out1 (instanceRef t5))
(portRef in1 (instanceRef t4))
)
)
)
(net t7
(joined
(portRef out1 (instanceRef t7))
(portRef in2 (instanceRef t4))
)
)
)
(net t4
(joined
(portRef out1 (instanceRef t4))
(portRef t9)
)
)
)
)
(net a
(joined
(portRef a)
(portRef in1 (instanceRef t5))
)
)
)
```



```
.names t14_3 [1541] [1542] t4_1
01- 1
1-1 1
```

```
.names t14_3 [1543] [1544] t4_2
01- 1
1-1 1
```

```
.names t14_3 [1545] [1546] t4_3
0-1 1
-01 1
110 1
```

```
.names t14_3 [1547] [1548] t4_4
010 1
100 1
```

```
.names t11_1 t11_2 t11_3 t14_1 t14_2 [1541]
10-1- 1
1-01- 1
01111 1
```

```
.names t11_1 t11_2 t11_3 t14_1 t14_2 [1542]
10-10 1
1-010 1
10101 1
11001 1
01110 1
```

```
.names t11_1 t11_2 t11_3 t14_1 t14_2 [1543]
01-01 1
10-01 1
11011 1
01110 1
00110 1
01010 1
```

```
.names t11_1 t11_2 t11_3 t14_1 t14_2 [1544]
0110- 1
1000- 1
01-01 1
00101 1
01010 1
```

```
.names t11_1 t11_2 t11_3 t14_1 t14_2 [1545]
1011- 1
```

```

0110- 1
101-1 1
10-11 1
-0101 1
011-0 1
0-100 1

```

```

.names t11_1 t11_2 t11_3 t14_1 t14_2 [1546]
10--1 1
-0-01 1
01--0 1
-1010 1

```

```

.names t11_1 t11_2 t11_3 t14_1 t14_2 [1547]
011-- 1
0-1-0 1
1011- 1
101-1 1
-0101 1

```

```

.names t11_1 t11_2 t11_3 t14_1 t14_2 [1548]
111-- 1
--011 1
0001- 1
00-11 1
11-00 1
10100 1

```

```

.end

```

```

.model t15
.inputs b_1 b_2 c_1 c_2
.outputs t14_1 t14_2 t14_3
.clocks
.subckt t14 c_2=c_2 t14_1=t14_1 b_1=b_1 b_2=b_2 t14_2=t14_2 t14_3=t14_3 c_1=c_1

```

```

.model t14
.inputs b_1 b_2 c_1 c_2
.outputs t14_1 t14_2 t14_3
.clocks
.names b_1 b_2 c_1 c_2 t14_1
1-1- 1
11-1 1
-111 1

```

```

.names b_1 b_2 c_1 c_2 t14_2

```

```
001- 1
100- 1
0-10 1
1-00 1
1111 1
0101 1

.names b_2 c_2 t14_3
01 1
10 1

.end

.model t12
.inputs a_1 a_2 b_1 b_2
.outputs t11_1 t11_2 t11_3
.clocks
.subckt t11 b_1=b_1 t11_1=t11_1 a_1=a_1 b_2=b_2 t11_2=t11_2 t11_3=t11_3 a_2=a_2

.model t11
.inputs a_1 a_2 b_1 b_2
.outputs t11_1 t11_2 t11_3
.clocks
.names a_1 a_2 b_1 b_2 t11_1
1-1- 1
11-1 1
-111 1

.names a_1 a_2 b_1 b_2 t11_2
001- 1
100- 1
0-10 1
1-00 1
1111 1
0101 1

.names a_2 b_2 t11_3
01 1
10 1

.end
```

C.2 Resulting Flatten Blif

```

sis¿ read_blif exempleDeBlifSimple.blif
sis¿ print_stats
exempleDeBlifSimple pi= 6 po= 4 nodes= 7 latches= 0
lits(sop)= 157

```

Table C.2: Flatten blif statistics

```

.model exempleDeBlifSimple
.inputs a_1 a_2 b_1 b_2 c_1 c_2
.outputs t9_1 t9_2 t9_3 t9_4
.clocks
.names a_2 c_2 [1251] [1267] t9_1
1101 1
0-1- 1
--10 1

.names a_1 b_1 b_2 c_1 [1252] t9_2
111-1 1
000-1 1
11-11 1
-1111 1
00-01 1
-0001 1
101-0 1
010-0 1
10-10 1
-0110 1
01-00 1
-1000 1

.names a_1 a_2 b_2 c_1 c_2 t9_3
1011- 1
0001- 1
0010- 1
1000- 1
1-110 1
0-010 1
0-100 1
1-000 1
01111 1
11011 1
11101 1
01001 1

```

```
.names a_2 c_2 t9_4
01 1
10 1

.names a_1 b_1 b_2 c_1 [1251]
111- 1
11-1 1
-111 1

.names a_1 a_2 b_2 c_1 c_2 [1252]
11111 1
01011 1
01101 1
11001 1

.names a_1 b_1 b_2 c_1 [1267]
1011 1
0101 1
0110 1
1100 1

.end
```


Appendix D

Sis scripts

```
sweep; eliminate -1
simplify
eliminate -1
sweep; eliminate 4
simplify
resub -a
gkx -abt 30
resub -a; sweep
gkx -bt 30
resub -a; sweep
gkx -abt 10
resub -a; sweep
gkx -bt 10
resub -a; sweep
gkx -ab
resub -a; sweep
gkx -b
resub -a; sweep
eliminate 0
decomp -g *
eliminate -1; sweep
xl_part_coll -m -g 2 -n 5
xl_coll_ck -n 5
xl_partition -m -n 5
sweep
simplify
xl_imp -n 5
xl_partition -t -n 5
xl_cover -n 5
```

CODE EX. D.1: The 5lut.script doing the technology mapping for 5-Luts

Appendix E

Inference

This appendix highlights some internal mechanisms used during the inference process.

inferereOutputs drives the evaluation of the operator, based on its inputs' current values supported by:

- CIR_Block (see code E.3)
- CIR_MethodGraph (build its equivalent block and subcontracts to it or act as an atomic node (see code E.2))
- CIR_Operator (Checks if there is an input owning a strong type. Computes currentValue. See E.6)
- CIR_Value (do nothing)

infeTypes enumerates the inputs' n-up and call *inferereOutputsTypes* for each of those n-up supported by:

- CIR_Block (enumerates its n-up list and inferere output types in a loop. see code E.4)
- CIR_BlockWithIndependentNodes (perform a node by node inference)
- CIR_MethodGraph (build its equivalent block and subcontract to it. see E.1)

infeType Computes the current value supported by:

- CIR_Block (see code E.3)
- CIR_GeneratedValue (calls its source operator *inferereOutputsType*)
- CIR_PseudoValue (dispatches to its realValue)
- CIR_Value (do nothing)

E.1 CIR_MethodGraph

infernTypes

```
self cir isNil ifTrue: [self buildCIR].
self cir value infernTypes.
Screen default ringBell
```

CODE EX. E.1: the *infernTypes* method

infernOutputsType

“two cases:

- or the node must be seen as an atomic one, in which case a new instance of the target class is built prior to evaluation, an the local OOLUT is filled up with a new inValue – outIndex couple*
- either the corresponding block (cir) is build on demand prior to hierarchical evaluation”*

```
| out inV index |
self mustSynthesize
  ifTrue:
    [cir isNil
      ifTrue:
        [self buildCIR.
          self cir value name: self cir key.
          self theOutputValue source: nil].
        self cir value infernOutputsType.
        ^self cir value].

self infernInputsOnDemand.
inV := self currentRealInputsValue asArray.
index := self inValues indexOf: inV.
index = 0
  ifTrue:
    [out := TYTypeLiteral
      new: (self theTargetClass new perform: theTar-
getSelector
          withArguments: self currentInputsValue).
      self theOutputValue currentValue: out.
      self theOutputValue type add: out.
      self inValues add: inV.
      self outIndexes add: (Array
        with: (self theOutputValue type collection set
indexOf: out) - 1)]
    ifFalse:
      [self theOutputValue currentValue: (self theOutputValue type
valueAt: (outIndexes at: index) first + 1)].
  ^self
```

CODE EX. E.2: The *infernOutputsType* method

E.2 CIR_Block

```

inferType
  | inputsTypes |
  operators do: [:oper | oper output currentValue: nil].
  operators := operators collect: [:oper | oper inferOutputsType].
  inputsTypes := self inputs collect: [:a | a currentValue].
  self output typeInfo add: self output currentValue.
  self inValues detect: [:elt | elt = inputsTypes]
    ifNone:
      [self inValues add: inputsTypes.
       outIndexes add: (Array
                        with: (self output typeInfo indexFor: self output
currentValue) - 1)]

```

CODE EX. E.3: the *inferType* method

```

inferTypes
  | inputsTypes |
  inputsTypes := inputs collect: [:a | a type].
  self enumerate do:
    [:anInputCombination |
     operators := operators do: [:oper | oper output currentValue:
nil].
     anInputCombination with: self inputs do: [:a :b | b currentValue:
a].
     operators := operators collect: [:oper | oper inferOutputsType].
     inValues add: anInputCombination.
     outIndexes add: (Array with: self output currentValueIndex -
1)].
  self inputs with: inputsTypes do: [:a :b | a type: b]

```

CODE EX. E.4: the *inferTypes* method

```

inferOutputsType
  | inV index |
  self output currentValue isNil
    ifTrue:
      [inputs isEmpty
        ifFalse: [self inferType]
        ifTrue:
          [operators := operators collect: [:oper | oper output
currentValue: nil].
          operators := operators collect: [:oper | oper inferOut-
putsType]]].
  inV := self currentRealInputsValue asArray.
  index := self inValues indexOf: inV.
  index = 0
    ifTrue:
      [self outIndexes
        add: (Array with: (self output type indexOf: self output
currentValue))].
  ^self

```

CODE EX. E.5: the *inferOutputsTypes* method

E.3 CIR_Operator

infernOutputsType

“added by loic”

```

| in inTypes subCall strongInput |
strongInput := self inputs detect: [:a | a type forceOutputType]
                ifNone: [nil].
strongInput isNil iffFalse: [self output type: strongInput type copy].
self output currentValue isNil
    ifTrue:
        [self selector = #ifTrue:iffFalse:
            ifTrue: [self inferInputsOnDemand]
            iffFalse: [self inputs do: [:a | a inferType]].
        self receiver isNil
            iffFalse:
                [self receiver canBeEnumerated
                    ifTrue: [self computeOutputsType]
                    iffFalse:
                        [in := self allInputs asOrderedCollection.
                            in removeFirst.
                            inTypes := in asArray collect: [:a | a typeInfo].
                            subCall := CIR_MethodGraph
                                targetClass: (Smalltalk at: self
receiver typeInfo nameClass)
                                targetSelector: self selector
                                withTypes: inTypes.
                            subCall buildCIR.
                            subCall cir value inputs with: inTypes
                                do: [:aParam :anAssociatedType |
aParam type: anAssociatedType].
                            subCall optimize.
                            output type: subCall cir value output type]]]

```

CODE EX. E.6: The *inferOutputsTypes* method

Bibliography

- [1] Visualworks smalltalk. <http://www.cincom.com>.
- [2] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *Field Programmable Logic and Application*, LNCS, 1997.
- [3] E.M. Sentovich et al. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, Berkeley, May 1992.
- [4] T. Miyazaki et al. Cad-oriented fpga and dedicated cad system for telecommunications. In *Field Programmable Logic and Applications*, volume 1304 of LNCS, 1997.
- [5] V. George. *Low Energy Field Programmable Gate Array*. PhD thesis, 2000.
- [6] L. Lagadec. Madeo bet web page. <http://penarvir.univ-brest.fr/~llagadec/MADEOBET>.
- [7] L. Lagadec. *Abstraction, Modélisation et outils de CAO pour les architectures reconfigurables*. PhD thesis, Université de Rennes 1, 2000.
- [8] L. Lagadec and B. Pottier. Object oriented meta-tools for reconfigurable architectures. In *Conference on Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Application, SPIE Proceedings 4212 in Photonics East 2000 International Symposium on Intelligent Systems for Advanced Manufacturing*, nov 2000.
- [9] David A. Patterson and John L. Hennessy. *Computer Organization and rd-design: The Hardware/Software Interface*. Morgan Kaufmann, harcover 2nd edition, 1997.
- [10] B. Pottier and J-L. Llopis. Revisiting smalltalk-80 blocks: A logic generator for fpgas. In *FCCM'96*, 1996.
- [11] P. Quinton and Y. Robert. *Algorithmes et Architectures Systoliques*. Masson, 1989.
- [12] G.-M. Wu, Y.-U. Chang, and Y.-W. Chang. Rectilinear block placement using B*-Trees. pages 351–356.