Math2mat User Manual

HES-SO // Isys project

Partners:

HEIG-VD
hepia
EIA-FR
HES-SO//VS
HE-Arc

User manual of Math2mat GUI
Version: 1.0
Date of issue: February, 2011
Report originator: Yann Thoma (HEIG-VD)

Revision history

Version	Date	Description
1.0	16.02.2011	First draft

Contents

1 Introduction **4**
1.1 Conventions 4

2 Requirements **5**
2.1 Simulation 5
2.2 Ubidule tests 6

3 User's guide **6**
3.1 Project management 7
3.1.1 Empty project creation 7
3.1.2 Project creation from .m 7
3.1.3 Folder validation 7
3.2 Folders structure 7
3.3 Mathematical grammar 8
3.3.1 EBNF 9
3.4 Views 10
3.4.1 Editor 10
3.4.2 Dynamic view 11
3.4.3 Flat/tree view 12
3.4.4 Messages view 12
3.5 Code generation 13
3.6 Verification 13
3.6.1 Reference generation 14
3.6.2 Simulation 15
3.7 Ubidule tests 15
3.8 External tools 17

4 Tutorial **18**

5 Commands **19**
5.1 Menu File 19
5.2 Menu Edit 20
5.3 View 21
5.4 Tools 22
5.5 Configure... 23
5.6 Help 24

1 Introduction

The aim of the Math2mat software is to automatically generate a VHDL description of a mathematical function using floating point operators. It also generates a full SystemVerilog testbench that allows to validate the hardware description by comparing it to the initial mathematical description. The initial description corresponds to the Matlab/Octave syntax, and allow operations such as addition, subtraction, multiplication, division, and structures such as if/then/else and for loops.

The main features of Math2mat are:

- Editor for typing Octave code
- Dynamic code analysis, during typing
- Generation of synthesizable VHDL code
- Floating point operations
- SystemVerilog testbench generation
- Fully automated test suite for the generated code
- Graphical interface with internal structure viewing

As the software is still under development, don't hesitate to submit a wish on the Wiki page of Math2mat:
http://www.tobecompleted.org

1.1 Conventions

The following conventions apply in this manual:

- Octave code is presented in this **`format`**
- VHDL code is presented int this **`format`**
- Menu selection is shown **This→Way**
- Files and folders are shown `/like/this`
- File content is shown in a box that also highlights the filename:

filename.ext

```
Various source code;
And more;
```

2 Requirements

Math2mat has been developped in Java, with the Eclipse framework. The release should be self-contained, and should run on Windows and Linux. The following systems have been tested:

- Windows XP
- Windows 7
- Linux Ubuntu 10.10

VHDL code can be generated without additional software.

The software allows to view the internal representation of the structure that will lead to the VHDL generation. This view, far from being user-friendly, can be useful for developers, and is generated using dot. This software can be downloaded from

http://www.graphviz.org/

2.1 Simulation

For validating the generated files, Octave and QuestaSim are required. Octave is an open-source software closely related to Matlab, as the accepted syntax is the almost the same. Octave can be downloaded from

http://www.gnu.org/software/octave/

QuestaSim is an HDL simulator, provided by MentorGraphics. It allows simulation of SystemVerilog designs and testbenches. It can be purchased from

http://www.mentor.com/

Math2mat has been validated with QuestaSim versions

- 6.5
- 6.6

Tests were run with these versions, but the system should work with further versions of QuestaSim. Previous versions of QuestaSim are not supported, as they do not fully accept SystemVerilog constructs. One of OVM or UVM methodology is also required for correct simulation of the testbench. The files of the methodology are supplied by Math2mat, but the simulator needs to support these features.

Please not that for Linux users, dot and Octave can usually installed with the help of the software manager.

2.2 Ubidule tests

Math2mat allows to automatically generate a bitstream for a Xilinx Spartan3 embedded on a board called ubidule. The automatic bitstream creation requires Xilinx ISE to be installed. Tests have been conducted with version 12.2, but further versions should work. ISE can be retrieved by visiting the Xilinx website:

http://www.xilinx.com

3 User's guide

Math2mat provides a graphical interface as show in figure 1. Command line operations using a subset of the Java classes are also possible, and are documented in the developer guide.

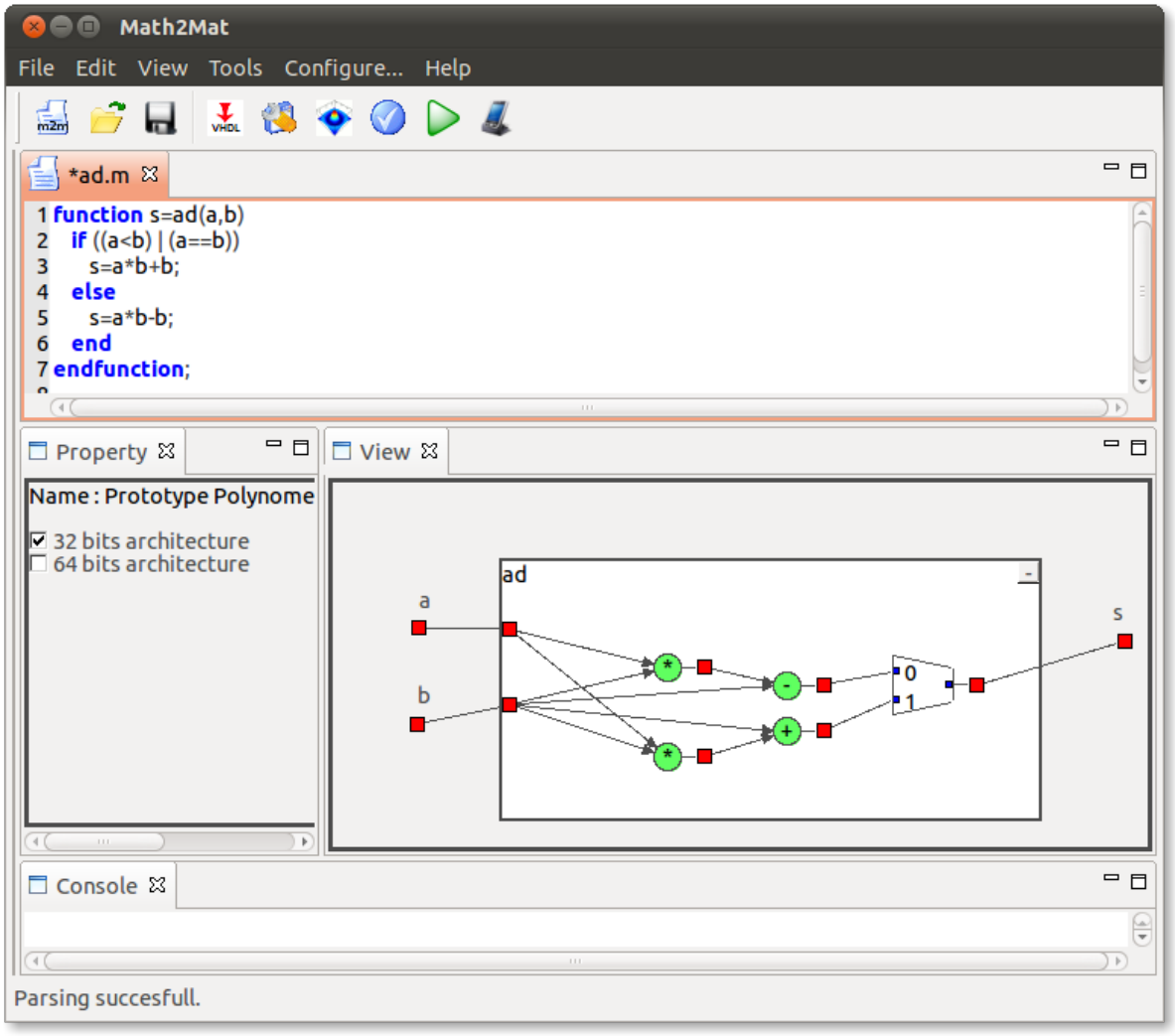


Figure 1: Math2mat graphical interface

3.1 Project management

A Math2mat project consists in a .m file that contains the mathematical source code, and in a certain number of parameters.

A project file has the extension "m2m", and contains a XML description of the entire project. This only file describing the entire project, the .m file is automatically regenerated by the tool if it is missing.

Standard Open, Save and Save as commands allow to open and save the project. Creation of a project can be executed in two ways: Through an empty project, or by importing a .m file.

3.1.1 Empty project creation

The creation of an empty project can be triggered through menu **File→Create empty project**. In that case, a template .m file is generated, and contains the following code:

<FileName>.m

```
% Here is a simple function that adds two numbers
% Feel free to modify it

function s=<FileName>(a,b)
    s=a+b;
endfunction
```

Where <FileName> is the name of the project.

3.1.2 Project creation from .m

The creation of a project from an existing .m file allows the user to select the name of the project, and then the .m file to import. The .m file is copied into the project directory, and so the original one won't be modified by Math2mat.

3.1.3 Folder validation

There is a possibility to launch the validation on a folder by selecting **Tools→Validate projects**. It recursively open every project found in the hierarchy, and launch the full verification process on the projects. At the end, a report in the console view indicates if errors were found.

3.2 Folders structure

During the creation of a new Math2mat project, a folder with the name `m2m_<funcName>` is created in the same directory as the project file (<funcName> is the function name). Inside this folder, a couple of other folders are created:

/

```
m2m_<funcName>
├── report ..... Contains the reports from simulation.
├── src_VHDL ..... Contains all the VHDL files generated by the software
│   ├── wrappers ..... Contains generic wrappers for the generated system
│   └── M2M_Test ..... Contains VHDL files for the ubidule implementation
├── src_SV ..... Contains all the SystemVerilog files generated for the
│                simulation
│   ├── ovm ..... Contains modified OVM source files
│   ├── ovm_original  Contains original OVM source files
│   └── uvm ..... Contains original UVM source files
├── src_m2m ..... Contains the Octave source file
├── src_OCT ..... Contains all the other Octave files needed for generat-
│                ing the reference outputs
├── sim ..... Contains the script that will be executed by QuestaSim
│   │            for the simulation
│   ├── comp ..... Will contain the compiled simulation files
│   └── iofiles ..... Will contain the input and expected output data files
│                generated by Octave
└── ubidule ..... Contains the files needed for testing the system on a
                 real ubidule hardware.
```

3.3 Mathematical grammar

The source file of a Math2mat project consists in a file that must conform to the Octave language. To date, the parser accept a subset of the language :

- Calculus operator (+,-,*,/,$\sqrt{x}$) ;
- Logical operator (and, or, $=$, $\neq$, $<$, $>$, $\leq$, $\geq$) ;
- Loop "for" ;
- Conditional structure (if/then/else/elseif) .

A function accepts any number of inputs and outputs.
The following code illustrates a simple function with 2 inputs and 1 output:

example01.m

```
function s=example01(a,b)
    s = a*2;
    s = s+b;
endfunction
```

The following code illustrates the if/then/else structure:

example02.m

```
function s=example02(a,b)
    if (a<b)
        s = a*2+b*3;
    else
        s = a*2-b*3;
    end
endfunction
```

The following code illustrates a function with 2 inputs and 2 outputs, as well as a for loop:

example03.m

```
function [s1,s2]=example03(a,b)
    s1 = 0;
    for i=1:1:10
        s1=s1+a;
    end
    s2 = s1+b*4;
endfunction
```

3.3.1 EBNF

Here is the complete EBNF syntax supported by the parser.

```
entry           ::= { pramga } , function ;
affect          ::= ident , [ tab_ind ], '=' , expr_lvl1 , [ ';' ] ;
body            ::= { instruction } ;
cmp_op          ::= "==" | "!=" | '<' | '>' | "<=" | ">=" ;
expr_lvl1       ::= expr_lvl2 , { ( '+' | ".+" ), expr_lvl2 } ;
expr_lvl2       ::= expr_lvl3 , { ( '-' | ".-" ), expr_lvl3 } ;
expr_lvl3       ::= expr_lvl4 , { ( '*' | ".*" ), expr_lvl4 } ;
expr_lvl4       ::= expr_lvl5 , { ( '/' | "./" ), expr_lvl5 } ;
expr_lvl5       ::= func_var , { ( "**" | '^' ), func_var } | '(' expr_lvl1 ')' ;
func_param      ::= func_var , { ',' , func_var } ;
func_var        ::= ( [ '-' ], ( ident, [ '(', func_param, ')' ] | number )
                    | '[', [ [ '-' ], number, { ',', [ '-' ], number } ] ) ;
function        ::= function_begin , body , function_end ;
function_begin  ::= "function" , ( '[', ident, { ',', ident } ']' | ident ),
                    "=", ident, '(' , param , ')' ;
function_end    ::= ( "end function" , ident , ';' | "endfunction" [ ';' ] );
ifthenelse      ::= "if" , '(', logexpr, ')' , [ "then" ] , body ,
                    { "elseif", logexpr, body }, [ "else" , body ] ,
                    [ "end" | "endif" ] ;
ident           ::= char , { digit | char | '_' } ;
instruction     ::= affect | loop | ifthenelse | op_switch | not_instruction_cmd ;
logexpr         ::= logterm, [ log_op, logexpr ] ;
logterm         ::= logfactor, [ cmp_op, logterm ] ;
logfactor       ::= [ '!' ], ( func_var | '(', logexpr, ')' ) ;
log_op          ::= "or" | "and" ;
loop            ::= loop_begin , body , [ "end" | "endfor" ] ;
loop_begin      ::= 'for' , ident , '=' , func_var , ":" ,
                    func_var , [ ':' , func_var ] ;
not_instruction_cmd
                ::= ( "printf" | "error" ), '(', any_string, ')', [ ';' ] ;
number          ::= digit, { digit }, [ '.', digit, { digit } ] ;
```

```
op_switch       ::= "switch", func_var, { "case",
                    ( func_var | '[', func_var, { ',', func_var } ),
                    [ ',' ], body }, [ "otherwise", [ ',' ], body ], "end" ;
param           ::= ident , { ',' , ident } ;
pragma          ::= "%m2m" , ident , ':' , ident ;
tab_ind         ::= '(', number, ')' ;

char            ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                    'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                    'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                    'v' | 'w' | 'x' | 'y' | 'z' |
                    'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                    'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                    'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                    'V' | 'W' | 'X' | 'Y' | 'Z' ;
digit           ::= '0' | '1' | '2' | '3' | '4' |
                    '5' | '6' | '7' | '8' | '9' ;
```

3.4 Views

Different views are proposed to the user, the main one being the editor in which the mathematical code has to be written. From this code, a flat/tree view allows to visualize the internal representation of the function, and a dynamic view presents the hardware structure that can be generated.

3.4.1 Editor

A text editor allows to edit the mathematical code. It highlights the text following the Octave syntax.
The editor executes a live parsing every time the user modifies the text. The error messages are displayed at the bottom of the main view, and allow to rapidly verify the code. Moreover, if the dynamic view is shown, then the content of this view is dynamically modified if the code corresponds to a correct syntax.
Using the **Edit** menu, the following actions can be processed on the text:

- Execute Undo/Redo actions
- Cut the selected text and put it into the clipboard
- Copy the selected text and put it into the clipboard
- Paste the content of the clipboard
- Delete the selected text
- Select the entire text
- Toggle the comments on the selected text
- Comment and uncomment the selected text

3.4.2 Dynamic view

The dynamic view corresponding to the source code can be shown by selecting the menu **View→Dynamic view**. It then contains a graphical description of the hardware that will be generated. This view is dynamically regenerated every time the source code changes, i.e. every time the code is valid.
Through this view, moving the pointer on the graphical objects displays their properties in the Properties view.
The following code will create the view presented in figure 2.

test.m

```
function s=test(a,b,c)
   if (a>b)
      s=a-b+c;
   else
      s=b-a+c;
   end
endfunction
```

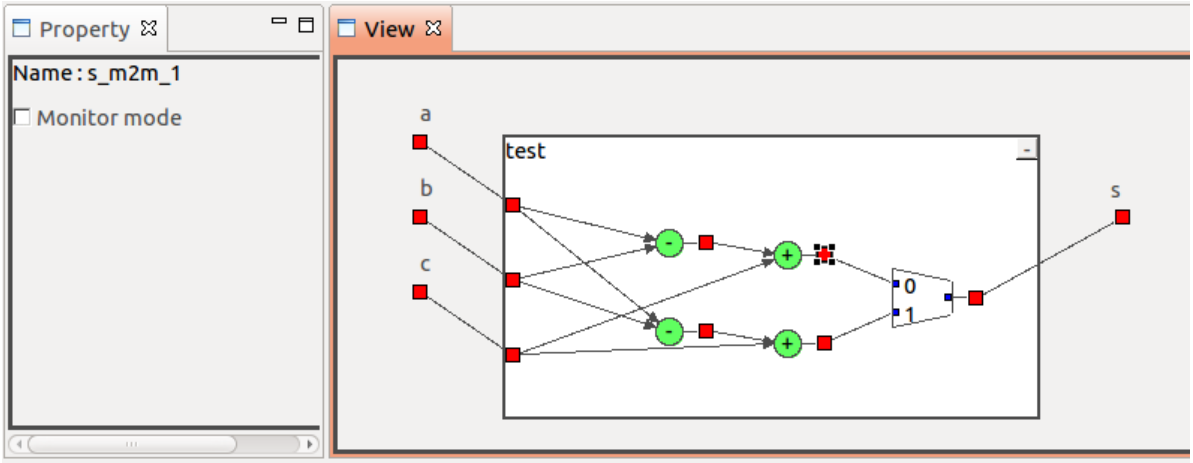


Figure 2: Dynamic view showing the hardware structure

Different properties are displayed and can potentially be changed, depending on the kind of object:

View outside function The type of floating point operations can be chosen: 32 or 64 bits.

Red square A red square represents an internal signal. It can be monitored or not. If checked, then the state of this signal is verified during simulation.

Operator An operator is a basic block that performs computation. Different implementations are proposed and can be chosen by the user. When selecting a block,

it is also possible to apply this change to all operators executing the same computation.

Multiplexer A multiplexer is used when if/then/else statements are present in the source code. Putting the pointer on a multiplexer highlights in blue the signals necessary for the condition calculation. It is also possible to monitor the condition of the multiplexer.

For increment In a "for loop" block, the increment is represented by a rectangle.

3.4.3 Flat/tree view

Math2mat offers the possibility to view the state of the internal structure. This structure is created during the code parsing, and is then manipulated in order to prepare the VHDL code generation. A standard user should note really need to open this view, letting this to developers. The tree view corresponds to the state of the structure after parsing, while the flat one represents its state before code generation.
These view requires "dot" software to be installed (cf. section 2, page 5). Math2mat generates a dot file called `fname_flat.dot` (or `fname_tree.dot`) that is then processed by dot to create a png file called `fname_flat.png` (or `fname_tree.png`) . This last file is then open by Math2mat. The *.dot and *.png files can be found in the `/src_m2m` folder. Figure 3 illustrates the schematics corresponding to the flat view of the following code:

adder.m

```
function s=adder(a,b)
    s=a+b;
endfunction
```

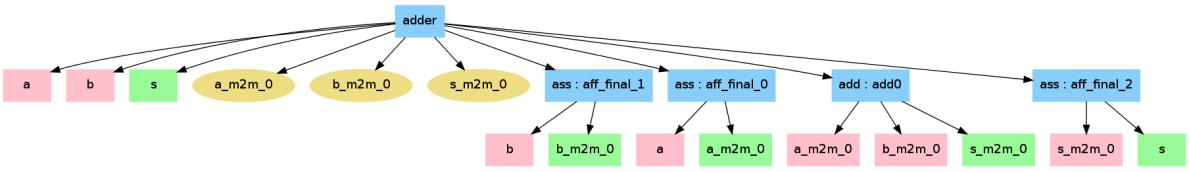


Figure 3: Internal structure corresponding to the operation $s = a + b$;

As the schematics generated can be very big, it is often easier to open them using a traditional image viewer instead of using Math2mat.

3.4.4 Messages view

At the bottom of the graphical interface, a message view corresponding to a console displays useful information in order to let the user know what happens.
This information depends on the type of processing the software is doing:

- **Editing**. Compilation errors or warnings
- **Octave code generation**. Messages about what files are created
- **Simulation**. Messages retrieved from QuestaSim
- **Ubidule testing**. Messages showing what happens during the synthesis, placement/routing processes, and during the test on the real hardware
- **Always**. Potential internal errors

3.5 Code generation

Code generation can be executed by selecting the menu **Tools→Generate VHDL**. The VHDL files are put in `/src_VHDL`. The description of the files structure and implementation can be found in another document.

3.6 Verification

Math2mat being able to generate VHDL code, checking that this code really does what it should allows to better trust the tool. For this purpose, simulation can be automatically launched in order to verify the generated system functionality. The testbench is generated in the folder `src_SV`, and is composed of SystemVerilog files. QuestaSim is required for the simulation of these SystemVerilog files, and is launched by Math2mat. The **Configure...→External Tools** menu allows to specifiy where the vsim executable is located, in case it is not accessible through the common path search. As Octave is also required to generate reference date files, it should also been reached by the common path or located through the same menu.
The verification is done in two phases:

1. Firstly reference data are generated by Octave;
2. Secondly SystemVerilog takes advantage of the references to simulate and verify the system behavior.

Simulation settings can be chosen through the menu **Configure→Simulation**. It opens a tab in which the settings can be entered. Figure 4 illustrates this view.
The following settings can be chosen:

- The calculation precision defines how error calculations are detected. It corresponds to the number of bits of the mantissa that can differ between the expected and the real results;
- The number of samples to test. One sample corresponds to a single call to the function;

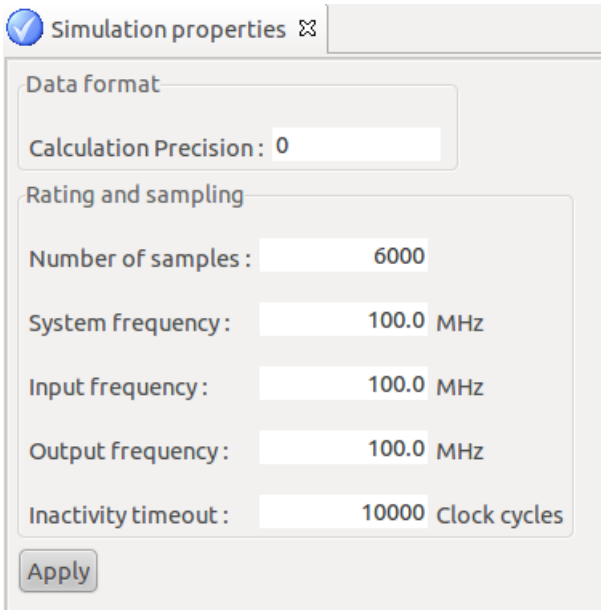


Figure 4: Simulation settings tab

- The system frequency is the frequency at which the system should work;
- The input frequency corresponds to the rate at which input data should be applied to the system;
- The output frequency corresponds to the rate at which output data should be retrieved;
- The period of inactivity allows to automatically end the simulation if no activity is detected on the output for a certain number of clock cycles.

The three frequencies are used in order to more or less stress the system. For each input of the system (each input variable), a valid input is applied at a random rate with probability $\frac{inputfrequency}{systemfrequency}$. For each output of the system (each output variable), a ready signal is applied at a random rate with probability $\frac{outputfrequency}{systemfrequency}$.
The new settings are set when pressing on the Apply button, and will also be stored in the project file.

3.6.1 Reference generation

Octave is used to generated files describing the inputs and files in which the reference output values are stored. All these files are stored in `/sim/iofiles`, and have names being `file_input1.dat`, `file_input2.dat`, ... for the input, and `file_output1.dat`, `file_output2.dat` for the ouptut. The number corresponds

to the place of the input/output in the Octave source code. The calculation is performed using the source file edited in the Math2mat editor. However, if internal variables have to be monitored, then the octave file is regenerated in order to create data files for the internal variables. In that case, the menu **Tools→Test regenerated Octave code** allows to validate the regenerated code, ensuring that its functionality is the same as the original file.

3.6.2 Simulation

Simulation is handled by QuestaSim. QuestaSim is launched in batch mode and executes the script that was previously generated: `/sim/simulation.do`. The output of the simulation is displayed in the console view, the end of the message allowing to observe if the simulation went well or not. The last lines look like this:

Successful		**Unsuccessful**	
M2M_WARNING :	0	M2M_WARNING :	0
M2M_ERROR :	0	M2M_ERROR :	1000
M2M_FATAL :	0	M2M_FATAL :	0

If by any unlucky day the simulation does not go well, then QuestaSim can be used with its graphical interface. The user only needs to go to the `/sim` directory and to launch the script `simulation.do`.

3.7 Ubidule tests

Simulation allows to trust the generated files in terms of functionality. There are garanties that the VHDL description is synthesizable, and tests with real hardware can be done. For this purpose, the software can validate the description using a ubidule (cf. figure 5).
This feature requires a ubidule to be set up with a server running. It is therefore not available to all users at any time.
A ubidule embeds an ARM processor running embedded Linux, and a Xilinx Spartan3, as shown on figure 6.
In this context, Math2mat can automatically perform validation on the real hardware by performing the following actions:

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream

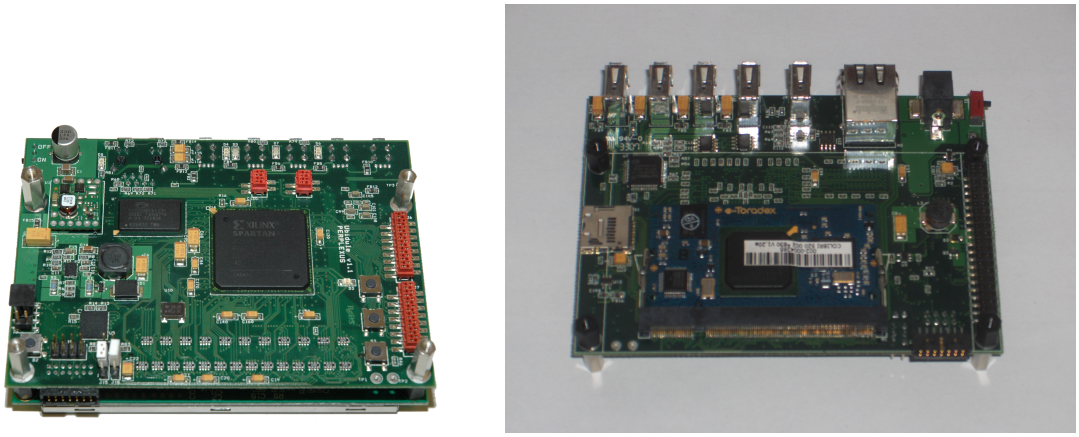


Figure 5: Picture of a ubidule board (top and bottom)

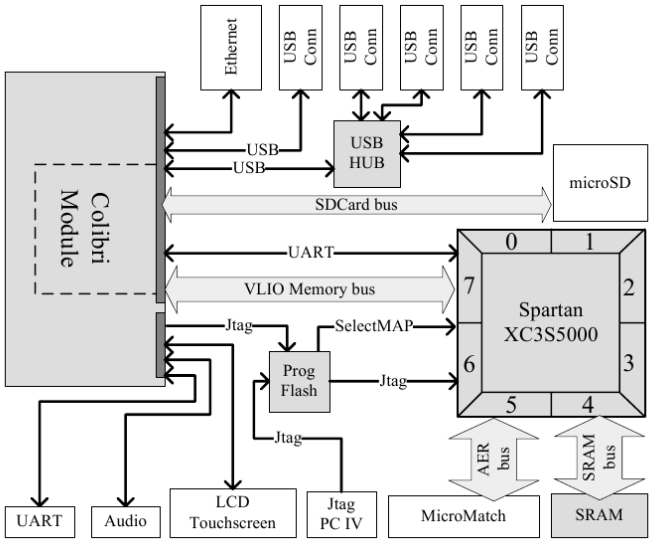


Figure 6: Schematics of a ubidule

5. Connect to a remote server that runs on the ubidule
6. Send the bitstream to the server
7. Send input data
8. Retrieve output data
9. Compare the calculation output to the expected ones

Through the GUI menus, the following actions can be launched:

- The menu **Tools→Ubidule test→Launch entire ubidule process** launches this entire process.

- The menu **Tools→Ubidule test→Launch ubidule files generation** launches steps 1 to 4.
- The menu **Tools→Ubidule test→Launch ubidule verification** launches steps 5 to 9.

The VHDL files for the ubidule implementation are generated in `/src_VHDL/M2M_Test/`, while the scripts are generated in `/ubidule/`.

3.8 External tools

As many different tools are required for the different actions the software can perform, a dialog allows to specify some information about these tools, as shown in figure 7.

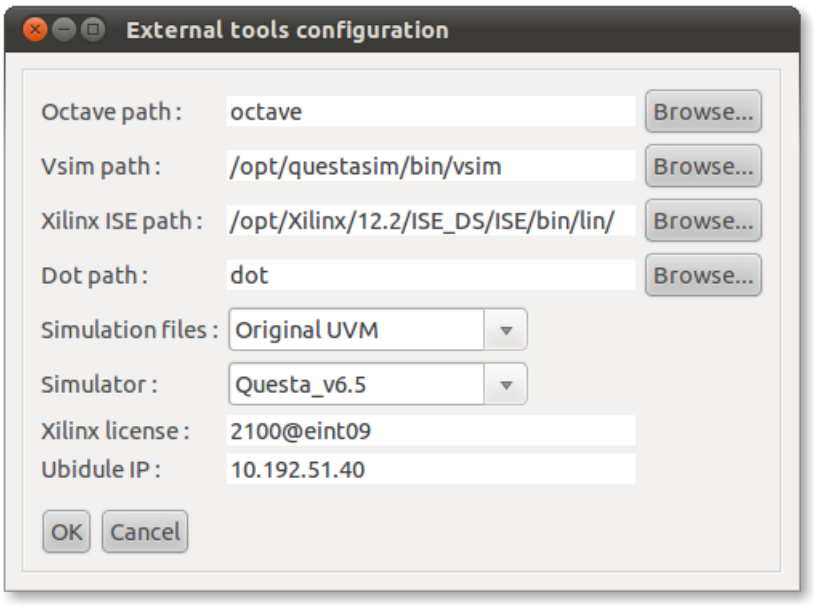


Figure 7: External tools properties dialog

As already mentioned, no external tool is required to simply generate a VHDL description. Octave and QuestaSim needs to be accessible for verification, Xilinx ISE only for ubidule tests, and dot for visualizing the internal structure.
For these four tools, by default their name is proposed by Math2mat. If they are not accessible through the path, then the user can specifiy the location of the executable, as shown in figure 7 for Vsim (the executable for QuestaSim).
⚠ For ISE, only the directory where ISE is installed should be set, not the executable itself.
Two combo boxes allow to specify general settings for the simulation. The simulation files lets the user choose to use OVM, UVM, or modified OVM files. The only changes

during simulation are the way messages are displayed in the console. The prefix will be:

Simulation files	Prefix
OVM	`M2M_ERROR : 0`
Original OVM	`OVM_ERROR : 0`
Original UVM	`UVM_ERROR : 0`

If the user needs to perform simulation using the QuestaSim Gui by executing the script `/sim/simulation.do`, then it is better to use the original options, as the errors will be identified by QuestaSim.
The simulator allows to specify what version of QuestaSim is installed on the computer. Depending on the version (up to 6.5 or from 6.6), different SystemVerilog code is generated in order to attain internal signals in the VHDL hierarchy.
The last two fields are required for the ubidule tests. The license for the Xilinx tools has to be specified, as well as the IP address of the ubidule on which the testing server is running.

4 Tutorial

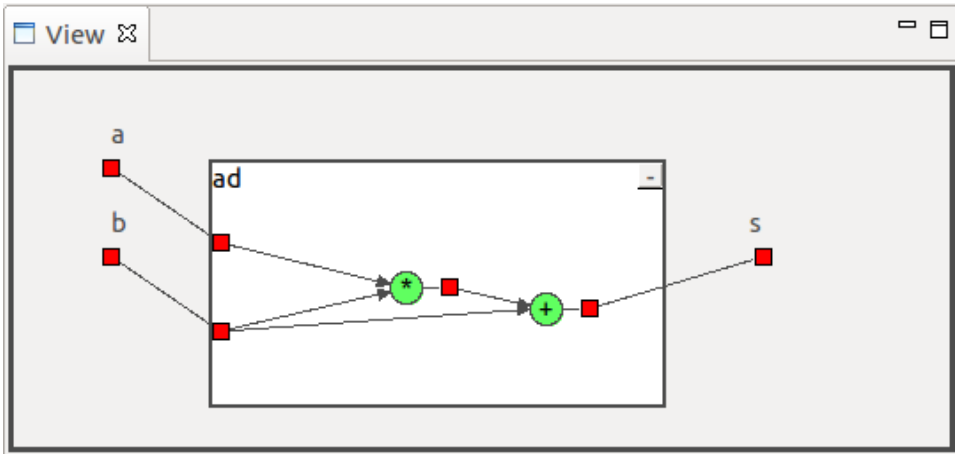
This tutorial shows you the steps required to create a new project, generate the VHDL code, and simulate it. If something goes wrong with the steps, it is probably because Math2mat does not find the external tools. In that case messages should appear, and you'll be able to specify their location with the menu **Configure...**→**External tools**.
So, here are the steps:

1. Select the menu **File**→**New empty project**;
2. In the dialog, choose the name of the project. You should create it in an empty directory, as it will be populated with the directories of Math2mat;
3. The new file contains a function that performs an addition;
4. Open the dynamic view (**View**→**Show dynamic view**) to observe the kind of digital system that will be generated;
5. Feel free to modify the function, but at least you can start with the simple adder;
6. To generate the files and simulate them, select the menu **Tools**→**Run all**, or click on the  icon;
7. And there you are... Too simple, no?
8. Add a little bit of complexity to the function, making it look like this:

<FileName>.m

```
function s=<FileName>(a,b)
   s=a*b+b;
endfunction
```

The dynamic view should now look like this:



9. Move the pointer onto different places of the dynamic view. You can observe the properties of the objects;
10. Click on one of the red squares, and select Monitor mode in the property view. It will then monitor this signal during the next simulation;
11. Click on the icon to generate and simulate the new design.

5 Commands

This section lists all the commands accessible from the different menus. A reference allows to rapidly go to the section in which more details about the command can be found.

5.1 Menu File

File→New empty project Section 3.1, page 7

Create a new empty project. The user is asked to choose a directory and a name for the project. The .m file is automatically created and corresponds to the name of the project. A function template is also written into the file, in order the names to match. This function is

<FileName>.m

```
% Here is a simple function that adds two numbers
% Feel free to modify it

function s=<FileName>(a,b)
   s=a+b;
endfunction
```

Where <FileName> is the name of the project.

Create new project from .m file Section 3.1, page 7

Create a project with an initial source code file copied from an existing one.

Open Section 3.1, page 7

Open a Math2mat project file.

Save Section 3.1, page 7

Save the current project, as well as the .m file

Save as... Section 3.1, page 7

Save, with a new name, the current project.

Exit

Quit the application.

5.2 Menu Edit

Undo Section 3.4.1, page 10

Undo the last action.

Redo Section 3.4.1, page 10

Redo the actions that have been undone.

Cut Section 3.4.1, page 10

Cut the selected text and put it into the clipboard.

Copy Section 3.4.1, page 10

Copy the selected text into the clipboard.

Paste Section 3.4.1, page 10

Paste the content of the clipboard into the text editor.

Delete Section 3.4.1, page 10

Delete the selected text.

Select all Section 3.4.1, page 10

Select the whole text of the current editor.

Toggle comment Section 3.4.1, page 10

Toogle the comments for the selected lines.

Comment Section 3.4.1, page 10

Comment the selected lines.

Uncomment Section 3.4.1, page 10

Uncomment the selected lines.

5.3 View

Show the flat view Section 3.4.3, page 12

Show the flat view of the internal structure in the form of a .png file loaded in the GUI.

Show the tree view Section 3.4.3, page 12

Show the tree view of the internal structure in the form of a .png file loaded in the GUI.

Show the dynamic view Section 3.4.2, page 11

Show the dynmic view corresponding to the mathematical code.

Clear the console

Simply clear the console view.

5.4 Tools

Generate VHDL Section 3.5, page 13

Generate the VHDL files corresponding to the Octave code.

Launch test generation (with Octave) Section 3.6.1, page 14

Generate the input/output data files by launching Octave.

Launch Verification Section 3.6, page 13

Start the verification using QuestaSim. It requires the VHDL and the input/output files to be up to date.

Run all Section 3.6, page 13

Run the entire flow: VHDL generation, input/output files generation, and simulation.

Test regenerated Octave code Section 3.6.1, page 15

This command allows to verify that the regenerated Octave code executes the same functionality as the original one.

Validate Projects Section 3.1.3, page 7

Allow to recursively validate all the projects found in a directory.

Launch ubidule files generation Section 3.7, page 15

Launch the ubidule files generation:

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream

Launch ubidule verification Section 3.7, page 15

Launch the ubidule verification, taking into account that the files have been previously generated:

1. Connect to a remote server that runs on the ubidule
2. Send the bitstream to the server
3. Send input data
4. Retrieve output data
5. Compare the calculation output to the expected ones

Launch entire ubidule process Section 3.7, page 15

Launch the entire ubidule test process:

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream
5. Connect to a remote server that runs on the ubidule
6. Send the bitstream to the server
7. Send input data
8. Retrieve output data
9. Compare the calculation output to the expected ones

5.5 Configure...

External tools Section 3.8, page 17

Configure the external tools properties.

Simulation Section 3.6, page 13

Modify the settings for the simulation.

Optimization Section 3.5, page 13

Modify the settings for the optimization.

5.6 Help

About Math2mat

A standard About dialog, that displays the version number, as well as the way to to contact the developing team and some information about Math2mat.

Key assist...

Allow the user to view the shortcut keys.