Bachelor thesis

Logisim to PyDEVS translator

Author:
Yentl Van Tendeloo

Promotor:
Prof. Hans Vangheluwe

May 21, 2012

Contents

Abstract

This report accompanies the Logisim-to-PyDEVS translator that was written in the context of my bachelor thesis at the University of Antwerp.
This translator will make it possible to translate Logisim[1] circuits to equivalent PyDEVS[5] code. This is a partial continuation of the bachelor thesis of Naomi Christis[2]. The generated models support both XMLTracePlotter[6] and VCD exports in PyDEVS, using the modified simulator.
As a side note, the original verilog translator and intermediate tree parser needed some modification. This was needed because these were programmed using Logisim version 2.6.1 files. Due to an update of Logisim, to version 2.7.1, this parser was no longer completely correct and some new features in Logisim had to be implemented in Verilog too.

Chapter 1

Introduction

1.1 Project description

This project consists of multiple parts, all of which are based on the Logisim to Verilog translator. The different parts are:

1. Update the original Logisim to Verilog translator
   (a) Update to support Logisim version 2.7.1
   (b) Speed up the connection algorithm from $O(n^2)$ to $O(n)$
2. Implement a library of Logisim elements, translated to DEVS
3. Translate Logisim circuit files to equivalent DEVS
4. Trace file export
   (a) VCD file (GTKWave)
   (b) XML file (XMLTracePlotter)

There are multiple reasons why such a translator is needed. The main reason is to allow already existing DEVS models to incorporate Logisim circuits and use them effectively. Other reasons include the possibility to test the Logisim circuit using a pre-generated DEVS circuit, to export a trace to XMLTracePlotter, ...

1.2 Project scope

One rather serious constraint is placed on the input files (apart from being generated by Logisim):
Since the Logisim save file is not according to any standard, it can always change between different Logisim versions. This might effectively break the intermediate tree parser, which has already happened to the original Logisim-to-Verilog translator. This reduces the use of the translator, since it will need continuous development to support newer Logisim versions. The problem is partially solved by checking the version number before translating and showing an error message if necessary. Similarly, it is impossible to translate older circuits, though it is possible to open and save the old circuit with a newer Logisim version. This

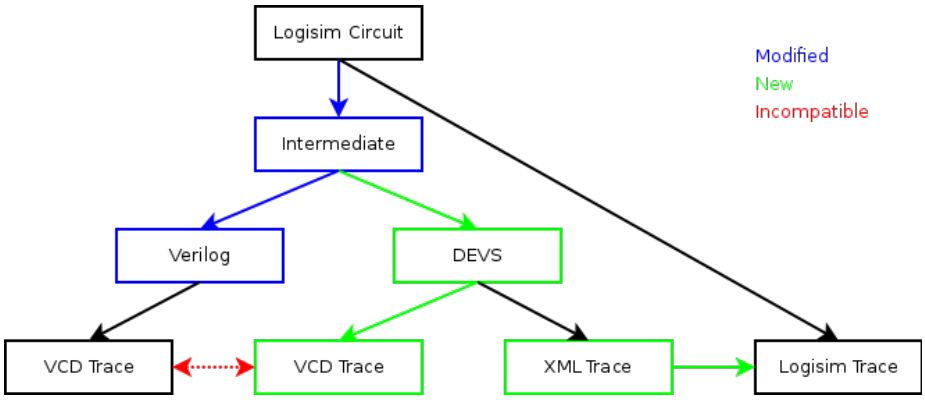


Figure 1.1: The relation between the Logisim to Verilog translator and the Logisim to DEVS translator

would make Logisim responsible for the conversion.
Secondly, a little less serious constraint, is that the Logisim files may include a lot of symbols that are not allowed in Verilog (and PyDEVS). Therefor, the ' ' and '/' characters are filtered out. This will make the translation of 2 different circuits with names only differing in these characters impossible. Since it is only the name, this problem is not really that serious. Besides, it goes against most common sense to name two elements exactly the same, only differing in the use of ' ' and '/'.

1.3 Requirements

The translator itself does not have a lot of requirements. Only Python3 is needed to be able to run the program. The choice for Python3 instead of Python2 is due to the fact that the intermediate tree parser was written in Python3.
The tests however, have some other dependencies. This is mainly due to the fact that the translator, and subsequently the tests too, have been written in Python3, though the simulator, PyDEVS, was written in Python2. This makes it impossible to do a real simulation without Python2 installed on the system. This is a rather logical requirement, since it would not make much sense to generate files to be used in a Python2 program, if no Python2 is installed on the system.
Another requirement for the tests is Java, which is needed for the comparison of VCD files [1] . This uses the open-source compareVCD program, which is written in Java.
To summarize, the requirements are:

- Python v3
- Python v2 (optional)
- Java (optional)

[1] An explanation for the need for Java can be found in section 1.4.2

If any of the optional requirements are not met, the tests that involve these components will be passed and the results will not be taken into consideration. Of course, the tests are ordered in such a way that all testable things are tested before a 'pass' occurs. This means that if a test passes, the basic translation and testing went correctly.

1.4 Testing techniques

1.4.1 Some ideas

A possible way to test the functionality of the final program would have been to translate the same Logisim circuit using both the Verilog translator and the DEVS translator. Afterwards, both can be run using their respective simulator (PyDEVS and iVerilog), using the same test values. The resulting VCD trace files can be compared. There are several reasons why this wouldn't work decently:

1. The Verilog translator doesn't add any delay
   This makes it impossible to generate identical trace files, since DEVS requires a certain delay

2. Naming is different
   The different translations need a difference in processing. E.g. the Splitter can be modeled in Verilog, but not in DEVS. A comparison between these files would thus reveal wires and elements that are present in one VCD file, but not in the other. In normal situations, this would be an error. Should this be allowed, it is no longer a real comparison, but rather guessing. These special cases and their solution in DEVS are mentioned in section 2.4.

3. Collisions might be resolved differently
   When different models have to output at the same moment, a collision occurs. PyDEVS resolves this by using a select() function, specified in the CoupledDEVS model. However, the order in which these are selected matters, because it might make some outputs of other elements unnecessary, incorrect, ... or even cause new models to generate output (when these models have a delay equal to zero). Due to this difference it would be rather difficult to exactly resemble the way Verilog handles this kind of collisions.

Another possibility would be to test the generated DEVS tree and check whether all connections are made the right way, all elements are translated to correct models, ... This kind of testing also has some important drawbacks, that made me consider not to use them:

1. The tests are unclear to maintain and to diagnose
   The tests don't have a clear, logical interpretation. One of the advantages of tests should be that the workings of the system can be easily seen. Also, tests should make it easy to debug the (faulty) code. When using this kind of checks, nothing is known about the system at all. When the same logisim file is redrawn and reparsed, it might even give other results,

since Logisim saves elements in the way they are added. This would cause a circuit file that is drawn in another way to fail the test, even though it looks exactly the same in Logisim.

2. The tests might fail, even though the program still runs correctly
   Since there are several places in the intermediate tree where order doesn't have any meaning, e.g. the connection groups, it doesn't really matter which wire gets checked first or in what order the connections are made. Therefor, any kind of absolute checking (as was done in the tests for the Logisim to Verilog translator) seem useless. Of course, it might be possible to have some more advanced checks, that just search the expected element and only give an error when these are not found or are not correctly bound. However, this would drastically increase the previous disadvantage. As long as the program doesn't lose its correctness, the test should succeed.

1.4.2 Final solution

So after all these different kinds of tests that don't seem all that usefull, a good test is needed. The best way to test, in my opinion, is to translate a circuit and use a testing framework to check if the simulator gives back the expected results. This does cause a large increase in testing time, since a lot of steps have to happen. But this seems the only way to do a decent check to test the final result, and not some (arbitrary ordered) intermediate tree.
To allow this kind of testing in DEVS, a CoupledDEVS model had to be written, that binds to another circuit at input ports 'input$X$', with X the number of the input port, and output ports 'output$X$', with X the number of the output port. To allow maximum flexibility, this framework gets constructed when starting the tests and just takes some lists for every input wire and some lists of expected values of output wires. This also requires a certain time parameter, since this test should wait until the circuit finished processing. This is due to the mandatory delay in DEVS. If the values match, the next value is inserted in the circuit. If the values don't match, the verbose output of the simulator is printed, together with some info about the received data, the expected data and on which wire this happened. This should make it easy to see where the tests went wrong.
The circuits that get tested should closely resemble some kind of unit test. So the basic circuits that are tested, are just circuits containing one element [2] that is immediately connected to input and output pins. This way, only that element is tested and problems can be easily identified.
Of course, most real life circuits don't contain only one element, so different combinations have to be tested. This way the connection between multiple models can get tested and how the different changes to the connections [3] are done in real situations.
Since the project scope is larger than only generating the DEVS models, also the VCD and XML files have to be tested. This is done in two ways, using a regression test and using a comparison to Logisim trace files.
The XML files are less error-prone than the VCD files, since the XML export

[2] Multiple elements are sometimes used, for when an element has many different configurations that should be tested separately

[3] More info about these changes can be found in section 2.4

function was already written and (presumably) well tested. The VCD files on the other hand need some decent testing since these are completely new and thus error-prone. For VCD files, the order in which changes are listed doesn't matter, as long as these changes happen at the same time. This suggests not to use a bit-by-bit comparison, since the result might still be valid. Searching the internet, an open-source program 'CompareVCD'[3] was found, which does just what is needed. Sadly, the program runs under Java, making Java a dependency for the tests.
XML file comparison is rather tricky. Again, a bit-by-bit comparison might not really be what is needed, but might be the only valuable solution. Some of the more elaborate tests (advALU, advTransistor, ...) generate XML files of several megabytes. When using an XML parser in Python to parse these files and compare the trees, the parsing alone takes multiple minutes. Other stand-alone applications also take this long. This is unacceptable, since testing would quickly take half an hour. Due to the fact that XML files should already be well tested when they were integrated in PyDEVS and it would take immensively long to do a complete check, only basic checks for the XML files are done. The binary comparison is done, though this might cause some failed tests even though there is no real error.

1.4.3 Trace files

Logisim recently supports the option to make some kind of logging files. This can be used to generate trace files with Logisim. Since PyDEVS also has an XML trace function, both traces can be made of an identical circuit (that runs completely autonomous thanks to a clock) and compared to each other. The Logisim trace files are very basic and contain nearly no information compared to the XML traces, therefor, the XML traces are stripped down to the same format as Logisim. These two trace files should be binary equal, so the standard UNIX diff can be used.

The logging functionality is (currently) only available through the GUI, making it impossible to fetch these traces automatically. The expected trace files are therefor generated and saved. When a new version of Logisim is released, all trace files should be regenerated.
Currently, all elements are tested in a brute-force manner: all possible combinations are tried, logged and compared. To make it feasible to do so, a maximum of 6 bits is used [4] . Any more bits would cause huge XML traces (several hundreds of megabytes) and would just take extremely long due to the verbosity of XML traces.
The primary advantage of this method is that it allows (semi-)automatic verification of the semantics in Logisim, compared to the semantics in the translator. The only source for semantics that was used in the translator was the Library Reference, so when the trace file comparison would fail, it might indicate a difference between implementation and semantics. Otherwise, it might also indicate an oversight in the Library Reference.
Note though, that these trace comparisons take a very long time, so they are integrated in the tests in a way similar to the 'make' program. When a circuit

[4] A bit actually can have 4 values in logisim: 0, 1, E or x. This causes $4^6 = 2^{12} = 4096$ combinations.

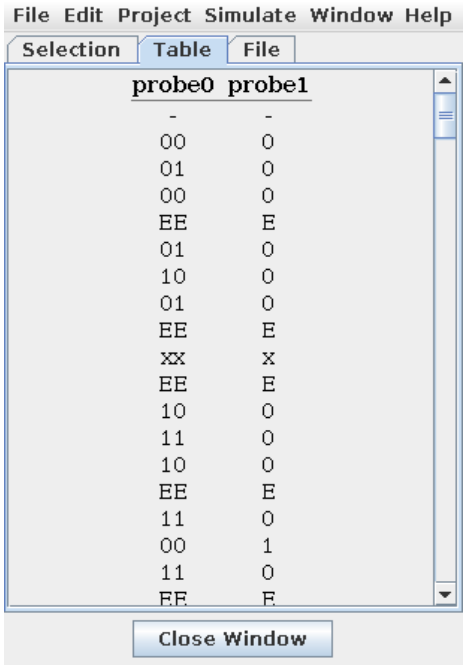


Figure 1.2: An example output of the logging feature of Logisim

is tested, a file will be generated (name.done). When the test finds this file and the timestamp is more recent then the trace file and circuit file, the test will immediately pass. Otherwise, the test will rerun.
When it is desired for these tests to be run on your computer, the .done files will have to be deleted [5].

1.5 Architectural design

The global architecture was already done by the Logisim to Verilog translator, so approximately the same architecture is used. This concerns the intermediate tree, the visit function, the write function, ... However, some smaller design properties have drastically changed, mainly due to the very verbose code that gets generated. In the Verilog tree, all writing and all models are generated in one single Python file, resulting in a lot of lines of code [6] and more difficult debugging since everything is connected to all other things. While the complete design up to the Verilog write visitor uses polymorphism and is rather clear, the write visitor itself doesn't and the classes are (nearly) empty. This was changed in the DEVS write visitor, which uses polymorphism extensively for the generated models. There is still one main class, which does the translation between different classes, but each class is responsible for how the resulting file would look like. Considering that the DEVS code is much more verbose than

[5] located in 'translator/testDEVS/trace/'
[6] About 2100 lines of code at the moment of writing

the Verilog code [7] , it would be insane to put all this code in a single Python file.
Another architectural difference is the generation of the connections. In Verilog, this can be done by using exclusive names, but in DEVS, the connections have to be specifically stated. Therefor, a dual pass system is used. First the connection groups are generated, effectively generating all the connections and the basis that will be used in the DEVS visitor. Afterwards, the next pass will convert all the classes to the DEVS equivalent and use the generated connections to handle all kind of special cases. Since this causes a seperation between connecting and translating, this allows for a much clearer design. The bad part is that it needs several passes over the tree, resulting in a slower conversion. Though translation of a pipelined datapath in Logisim only takes a few seconds on a netbook, this performance decrease will not even be in the order of seconds for far bigger circuits. Also, the algorithmic complexity doesn't change.
For the class diagrams, see chapter 6.

1.6 Planning

Month	Goal
September	Update the Logisim-to-Verilog translator Basic connections
October	Model Gates Model Plexers
November	Fix colliding wires, pull resistors, introduce Null element Refactoring and extend connection algorithm
December	Testing larger circuits and fix found bugs, catch up if late Model Arithmetic elements
January	Exams
February	Model all remaining elements and Memory elements Include working VCD and XML exports Include a configuration file Include some bigger tests
March	Model the Splitter Clean up code Fix bugs Write report Compare Logisim traces to own XML traces and fix found bugs Static bitsize checks Loop checks More configurability
April	Continue writing report Write presentation Implement a GUI Fix bugs
May	Finish report Finish Presentation Fix bugs

[7] The DEVS code for an AND gate is approximately 68 lines of code, while Verilog can do this in a single line

1.7 Usage

Usage is fairly simple, though there are several possible parameters:

- –devs: translate to devs
- –verilog: translate to verilog
- –config: supply a configuration file to use (default: config.ini)

For example, to translate the file 'circuit.circ' to DEVS using 'config.ini' [8] :

```
$ ./translate.py --devs circuit.circ
```

Note that the location of PyDEVS has to be specified in the configuration, as this needs to be imported. There are also some extra configuration options that can be used to fine-tune the generated code, see section 2.6.
Tests are even simpler, just run the command

```
$ ./test.py
```

Again, a configuration file is present for the testing, this should NOT be changed, since it contains some assumptions for the regression tests. Though the location of the generated models and PyDEVS might need changing.

[8] The command './translate.py' might be replaced with 'python3 translate.py', both have the same meaning (under Linux) due to the use of a sha-bang line

Chapter 2

Translator

2.1 Modifications

Some modifications to the intermediate tree parsing were needed due to the new version of Logisim. This is because the defaults were changed between logisim versions. These changes include the following:

- Added element Transistor
- Added element Transmission gate
- Added element Ground and Power
- Changed and added appearance of splitters
- Most plexers now have an optional enable line and a disabled signal
- Plexers can have multiple locations for the select and enable line
- Most basic gates now have an option to change the outputsignal

Most of these changes can be done in the parser itself, though some new elements and features are introduced that need changes to the Verilog part of the translator.
Additionally, a small check is included that first checks the version of the Logisim save file, to determine the correct defaults. Also, the version is checked against a list of compatible versions. This would at least prevent useless translation time when an incompatible version is used.

2.2 Intermediate tree

The parsing of the Logisim circuit file to an intermediate tree is already implemented in the Logisim to Verilog translator by Naomi Christis. This intermediate tree is used without any major modifications [1] , so no effect on the Verilog part of the translator should happen.

[1] The required changes are done, including several bugfixes

2.3 PyDEVS connector

2.3.1 Basic connections

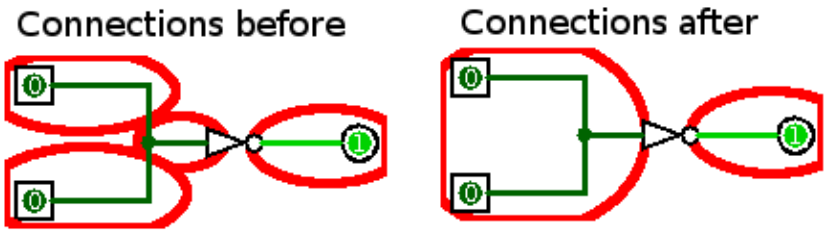


Figure 2.1: The different connection groups, before and after fixing colliding wires

The basic connection algorithm just searches all elements in the intermediate tree and uses introspection to find all connected wires and their names. All elements on the same wire are put in a list. All these lists are separate groups of elements that are connected to each other. All of these lists are then put in the connections list, which is the list that is always used to find connections.
Due to the way colliding wires are parsed in the intermediate tree, the colliding parts are put in different groups. This is not the best way, so some of these groups should be merged together.

2.4 Translation problems

2.4.1 Splitters

The Splitter is a standard Logisim element, but has a strange behaviour in some cases, making it too difficult to translate it to a Model in DEVS. This closely resembles the problems that were encountered when building the Logisim to Verilog translator. The intermediate tree contains a lot of information about the splitter, but constructs some unnecessary inputs and outputs, making loops (see chapter 3.1.2) possible. This suggests that the simplest approach, just model it as a normal element, would be a recipe for disaster.
Besides all these problems, the splitter would have to take into account the different pull directions of each wire separately, prevent collisions, etc. However, a wire that contains multi-bit signals is actually just a bus, containing many different wires. In this point of view, a splitter is just an artifact of this nice representation in Logisim. This suggests the solution: removing the splitters and shredding these busses.
Though this solution seems rather simple, it would cause many esthetical problems when modelling all multi-bit wires like separate wires, e.g. a bit adder with two 32-bit inputs would have 64 different input wires and 32 output wires. This would make it impossible to do decent debugging of a circuit using a waveviewer and furthermore cause a lot of transitions in the simulator, causing the simulation to slow down immensly. Therefor, these busses will only be shredded when a splitter is in the connection group. This would still cause some elements to

have a strange interface, suggesting the addition of some extra helper models: the BitShredder and BitMerger (not to be confused with the Merger, which mergers different signals on one line and allowing collisions).
The BitShredder will take a bus of k bits as input and shred it in k 1-bit wires in the normal order (thus ignoring remapping of the splitter). The BitMerger will do just the opposite and take k 1-bit wires and put it on one k-bit bus. Since these elements are uni-directional, they have to be placed right before or after an input/output, thus making this a non-issue.
Now the real problem arises: connecting all these 1-bit wires in a correct fashion. This can be done by constructing a list of all BitShredders in a connection group and for every bit of these connections, follow the path and eventually hook up these wires to the BitMergers. This seems like a very inefficient way to connect the BitShredder and BitMerger, but this will actually pay of in the resulting model. This is because a lot of reuse can be done: no special bitshredders with many different output combinations, no special bitmergers with special input combinations, ... Another very important advantage is that the signal will just flow through other intermediate splitters, allowing these splitters to be completely removed. This way, a large network of multiple splitters for one wire will just be collapsed to a BitShredder/BitMerger combination, reducing the complexity of the model. This also has the advantage that all collisions and pull resistors can be solved on a bit level, making it very clear what happens.
An extra problem will arise when using the bit-by-bit method and following the flow immediately, since a loop can often occur. Section 3.1.2 continues about this problem and how it is solved.

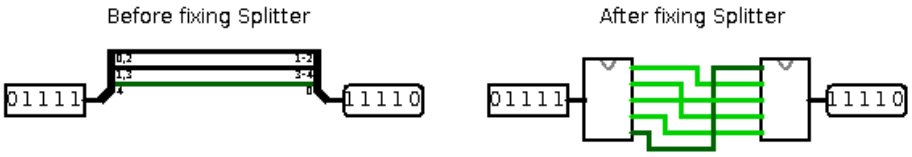


Figure 2.2: A visual representation about the changes for a Splitter

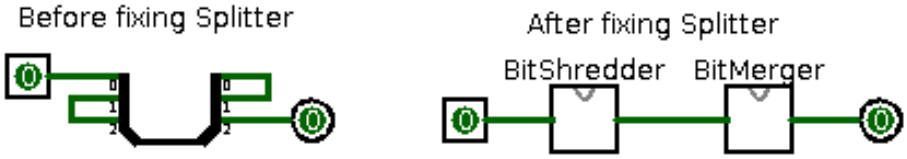


Figure 2.3: A more difficult circuit, illustrating a possible problem when using a model instead of the above algorithm. Note the very clear and efficient solution

2.4.2 Colliding wires

Colliding wires are a problem in DEVS, because a DEVS model has a single input for a certain wire. If multiple elements are connected to this port, the model will just take the last event as the correct one and discard the previous.

This is not what happens in Logisim and is again an artifact from DEVS, because it replaces continuous signals with single events. To solve this, a special element is introduced, namely the Merger element.
To detect this kind of collisions, a check is needed if multiple outputs are connected to each other. To accomplish this, some groups are constructed of elements that are connected to each other. All data that needs to be kept is the element, the name of the port and whether or not it is an input. However, the intermediate tree already does some processing for colliding wires, by adding an element CollidingWire. This is not really what is needed for this approach, so both ends of the colliding wire should just be merged to each other. This way, a decent group of elements is constructed.
Now a simple check for the number of outputs in such a group will suffice to find collisions. If only one output is defined or multiple inputs, no problem occurs. However, when multiple outputs occur, there is a possibility of a collision.
When a collision is possible, the group gets split in two parts. One part receives all inputs, and the other receives all outputs. Now all that is needed is a connection between these two groups. This is done by the Merger, as it will be constructed such that each output element will get an inputline in the Merger, and for all input elements, only a single output is defined. This last part might seem strange, however, no collision can occur since there will only be one output in this group.
Like all other elements, the Merger will have a state that contains the values of all input wires. The Merger element can then be used as a sort of cache for the events. When an event comes in, it will compare the enclosed signal bit-by-bit with all other signals and if necessary replace it with a 'E' signal. Notice that the constructed groups will come in handy in further parts of the translating. Therefor, this construction is done in a separate 'preprocessing' step of the translator.

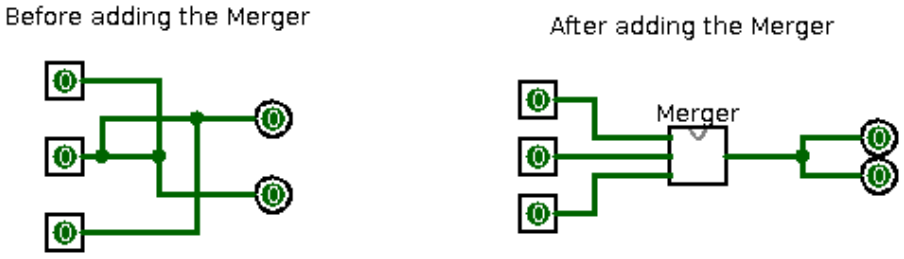


Figure 2.4: A visual representation about the changes for a Merger

2.4.3 Pull resistors

The Pull Resistor is, contrary to the Merger, also an element in Logisim. But since this element does not really have a normal input/output port combination and it isn't even represented as an element in the intermediate tree. This means that it will have to be represented with a new model. The intermediate tree does however contain a list of wires, which have the direction given by the pull resistor. Since the previous step already generated groups of connections, this

can now be used to add the Pull Resistor. Note however, that the problem of multiple outputs on one wire no longer exists, because this was solved by the fixing of the Merger. Thus resulting in an easy division between input and output elements. The insertion of the Pull Resistor happens analogous to the insertion of the Merger.
Because all connected wires have the same direction, it doesn't really matter which wire is chosen to find the direction. This does mean though, that the direction should be taken into account when drawing wires, e.g. when adding the Merger element. To fix this, the input wires of the merger don't have a direction, while the output wire does have the direction. Otherwise there would be a collision if one wire has a pull direction to 0, but is floating, and the other wire does have the 1 signal. The Merger would detect a 0 and a 1 and mark this bit as 'E'. However, in Logisim, the floating value would just be set to 1 and the pull direction wouldn't matter.

Figure 2.5: A visual representation about the changes for a Pull Resistor

2.4.4 The Null element

Sometimes, a connection between two elements just won't work in PyDEVS like it works in Logisim. Therefor, to fool PyDEVS, an extra intermediate element is introduced. This is called the Null element. This element does absolutely nothing, it only copies the input signal to the output port. All of this happens with a delay of 0, as to guarantee no difference in behaviour.

Coupled Input/Output connections

In Logisim it is possible to make an immediate connection between the input elements of the circuit and the output elements of the circuit. This obviously doesn't have any productive use [2]. But since it might be done, it has to be supported. However, the real problem lies with PyDEVS, which doesn't support this kind of immediate connections (again, for obvious reasons). To model these connections, there should happen an extra pass over the connection groups, that checks if an input is connected immediately to (multiple) outputs and if so, add a Null model between them.

[2]It might actually have an esthetical purpose: when connecting multiple elements to each other, that share a same wire for input, it can be drawn in logisim with straight lines instead of multiple curling wires.

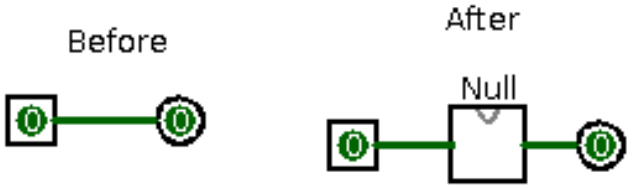


Figure 2.6: A direct connection not allowed by PyDEVS and its solution

Direct feedback loops

It's also possible to make a direct feedback loop in Logisim. However, PyDEVS doesn't allow this. Since a complete translator is needed, it is necessary to translate these files. However, since a direct feedback loop is probably not what the user intended, a warning will be displayed about a loop, but the translation will continue, since there are (semi) valid uses of this kind of connections. Since PyDEVS doesn't allow this, an intermediate element is introduced to fool PyDEVS. This element is, again, the Null element.

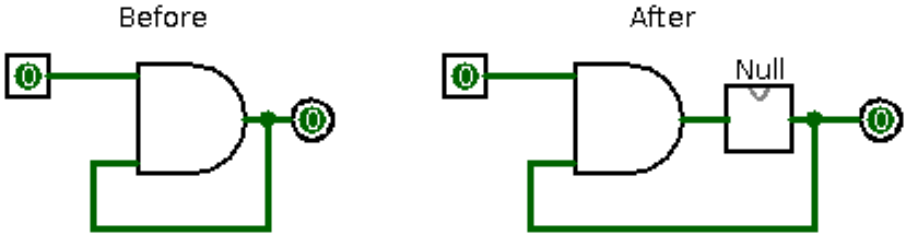


Figure 2.7: A loop not allowed by PyDEVS and its solution

2.4.5 Putting it together

It is possible to construct a circuit that does all these things at once. This has to be taken into account and thus it is necessary to define a correct order of fixing these problems. The order in which the problems were introduced is the one followed in the translator and is a correct one. The order can be explained according to the following constraints:

1. Splitters should be removed first
   This is because otherwise the colliding wires between BitShredder and BitMerger and pull directions of these wires will not be done correctly or efficiently. This does place a lot of constraints on the connection algorithm, because there has to be a possibility for multiple wires to be connected to a single BitMerger input. Also the Pull direction has to be taken into account.

2. Merging has to happen before Pulling
   This has already been mentioned when discussing the Pull Resistor solution. If Pulling happens before Merging, the Merger could have problems

when a floating signal gets pulled, while another wire has a fixed signal. This does however put a constraint on the working of the Merger, which should take into account the pull direction of the resulting wires.

3. Adding the Null element should happen last
   This isn't really necessary, but will result in slightly more performant code. It might be possible that a collision or a pull resistor appears, thus already adding an element between these problematic elements. When this step is done before all others, this might add a Null element, even when there are colliding wires and an extra element will be added nonetheless, this would add useless complexity to the model and should be avoided.

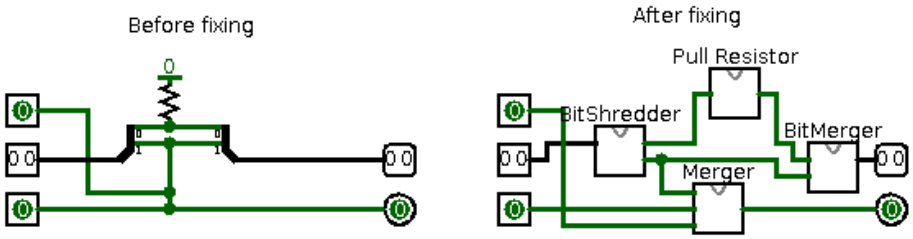


Figure 2.8: A visual representation about the changes for all of the above

2.5 PyDEVS code generation

The generated models need to comply with the semantics provided by Logisim. This suggests the use of the Logisim Library Reference, to find out what happens with different configurations, without having to find it out by constructing that particular circuit and constructing that particular situation. However, this library reference is often far from complete, since there often is no mention of what should happen when an input contains an error value. This did lead to manual checking of these situations. Note also that this could lead to problems when new versions of Logisim are released, because these semantics are undefined and therefor might change between releases. This problem with semantics is solved with the version number checking in the intermediate parser, which requires a code revision when a new version is released.

2.5.1 Basic assumptions

As DEVS requires a certain amount of delay, some concept of time needs to be introduced to the simulation of Logisim. Because, in the real world, all elements have a certain amount of delay [3] , it is possible to simulate this delay as the delay in DEVS. The different states that are constructed for the DEVS models are therefor an 'active' and an 'inactive' state. 'active' means that the output is pending, while 'inactive' means that the element is ready to process new input. If an element gets a new input while it is active, the delay will start all over again and no output for the previous input will ever be calculated. It is then

[3]This delay is called the propagation delay or gate delay

assumed that the input was not stable yet. The concept of how this delay is interpreted is analogous to the MIPS-32 implementation in DEVS by [7].

2.5.2 Colliding events

Due to the above explanation, it follows logically that the time at which elements output is crucial to other elements. Some elements might output at exactly the same instant [4] , causing a collision. The select() function in the CoupledDEVS model tries to resolve these collisions. This collision resolution might influence the result of the system, which is certainly not what is desired [5] , since in real life these collisions might have been resolved simultaneously.
In real life, some variation to this delay is natural, making this kind of simultaneous events another artifact of simulation. To resolve this, an option is provided to use random delays. Though for testing purposes, a fixed delay is more interesting, so it can be compared to known trace files. Because the fixed delay is used for testing, it is very important to have deterministic output, as it would otherwise fail the test, while both files are correct...
The select() function that gets generated is a rather intelligent one. It will run over all elements that are colliding with each other and pick the first one that doesn't influence any other element that is also waiting to output its value. The order in which these elements are chosen doesn't matter, since they won't have any influence on the other elements and as soon as collision resolution is over, it is impossible to view which element did output before the other. Of course, it is possible to have a loop of elements, all of which want to output a value. Because this problem doesn't have a decent solution, the first element in the list is returned, hoping to solve the problem.

2.5.3 Wiring

Most Wiring elements are not real elements that should be translated. They are already mixed in in the intermediate tree and are almost completely handled in the intermediate tree. This does however cause some problems when converting this intermediate tree to a DEVS connection group, as discussed in section 2.4. The BitExtender is just a normal element and is handled like one.

Transistors and Transmission Gates

This element is new in Logisim 2.7.1, so changes to the intermediate tree were needed. This was a good opportunity to learn about how the intermediate tree works and how it can be used. The elements themself can be easily implemented because they are nicely documented in the Library Reference. Because these elements were completely new, there is an advanced test for these elements separately, so all different types and directions get tested. The Transmission gate is basically a Transistor with an extra input, so it is always treated like another type of Transistor (type C).

[4] Equality is up to an ϵ, defined in PyDEVS

[5] Actually, a bad resolution could even lead to complete blocking of an element, when an element keeps sending new inputs to the other element, and the first element is always chosen to resolve the collision, the second element will always start over

Power and Ground

This element is also new in Logisim 2.7.1, but is actually just a constant with a fixed value. This simply suggests changing the intermediate parser to set it as a constant. Only the bitsize has to be determined to know the exact value of the constant.

Probe

The probe element doesn't really do anything useful, but it doesn't have a bitsize. This makes it necessary to not take this element into account when checking for bitsizes. Since this element doesn't have an output, nothing can be done. Printing the content of the wire would be rather dirty, since it would mess up the normal PyDEVS output. However, the element gets generated, though it will be empty. The user might choose what to do with it, either in the Probe.py file itself, or in the generated model.

2.5.4 Gates

Contrary to Verilog, DEVS (actually Python) doesn't have a standard way of doing most of these gates, like AND, OR, NOT, ... [6] so they have to be manually implemented. These elements are like the core blocks of every circuit so are very nicely documented, making it easy to write these blocks.
Note however, that since Logisim 2.7.1, it is possible to change the default output format of most basic gates. This makes it possible to change a 0 signal to floating, or a 1 signal to floating and such. This adds to the complexity of the models, since they need to have another pass to change this.

2.5.5 Plexers

The plexers are rather simple, but quickly become very large. This is due to the fact that there are no hardcoded names of wires in the code. To allow for maximum flexibility and simulation speed, the same code might get duplicated multiple times. This does not really impact the simulation speed in a bad way, since it will remove a function call. For maintainability, no negative impact is experienced, since the translator just contains that code once. If there were a bug in the Plexers, it would be best to change this in the translator instead of the generated model. Otherwise the bug will persist in future plexers.

2.5.6 Arithmetic

Most arithmetic elements are just normal elements and have a very easy implementation, so they are modeled that way. Though some elements do have a certain difficulty to them, namely the basic arithmetic elements (Adder, Subtractor, Multiplier and Divider).

[6] Actually, Python does have some bitwise operators, but these don't handle the different error and floating values like Logisim

Basic arithmetic

These elements have a high grade of complexity, due to functionality given by the Logisim developer. e.g. providing partial operations if a floating or error value is detected. Again, Python doesn't have the kind of nice bit addition like Verilog has, and certainly not with the partial operations feature.
However, a conversion from binary to decimal and decimal to binary does exist in python, suggesting the conversion to decimal, decimal operation and finally reconversion to binary. However, this conversion to decimal doesn't allow values other than 0 and 1, causing a lot of problems with the partial operations. All these problems caused the code to be rather verbose. The behaviour of the partial operation also differs between the different operations, making it hard to reuse a lot of code.

2.5.7 Memory

The memory elements are special in the sense that they have a common part, namely the way they get triggered. This provides a good opportunity to make a superclass that handles the triggering and just sets a variable (self.triggered in this case). This also reduces the complexity in all Memory models. Thanks to this decision, all Memory models are fairly easy to implement. The internal representation of the memory can vary depending on the element. This allows for a lot of performance gain. e.g. ROM memory has a complete list of all adressess and their value, but RAM memory can be optimized by using a dictionary. This way, only used values are stored, while accesses keep $O(1)$. ROM memory on the other hand, can be optimized to only initialize the defined values and all access to higher addresses are just translated to 0.

ROM

The ROM needs to have its content defined by an extra file, this is to make large ROM memories easier to modify. This is analogous to the implementation in the Logisim-to-Verilog translator.
This can easily by optimized, by saving the values in decimal format. Since the data is originally saved and sent as strings, saving as a string would give an immense overhead due to the representation as a character. When translated to an integer, it can be saved optimally in memory, though it will require a lot of conversions. Another advantage is the early detection of files that contain non-binary characters, since the parsing is already done at initialization.

Random

The Random element is a special kind of element, in the way that it depends on randomness. The library reference tells that, when a non-zero seed is given, this seed will be used according to a certain formula. This is implemented and should work exactly like in Logisim. However, when a seed equal to zero is given, the time will be used to initialize the seed. The problem is that there is no info about 'what' from the time is used (e.g. unixtime, milliseconds, time till startup, ...). This problem could still be resolved by looking in the source-code of Logisim, however, it is likely unimportant. When using a seed that is unknown at the moment the circuit is run, it doesn't really matter which seed

is chosen, as long as it has something to do with time. Therefor, the PyDEVS implementation just takes a seed depending on time, but this might give other results then the Logisim Random element at the exact same time. But since the user will never notice this, it is safe to do so.

2.5.8 Input/Output

This category of elements does not really contain anything that can be done decently in a modelling language. Note however that Python is a complete programming language, making it possible to offer this kind of elements and really hooking up some kind of input/output device. This would lead too far and thus this category is unsupported, apart from the joystick, which is simulated with two different input ports. This behaviour is the same as the Logisim to Verilog translator. All other elements will display an error about an undefined element. Generating all these files as empty files would take too long for the user to make them, so it would be best that these elements are defined when needed.

2.5.9 Imported circuits

Importing other subcircuits is relatively easy, since these are just modeled like other elements, but use the CoupledDEVS superclass instead of the AtomicDEVS superclass. There is a minor problem when custom appearances are selected for the circuit, since this might change the location of the ports and break the connection algorithm. This problem is not easily solved, since the way these appearances are saved is non-trivial due to the visual nature of Logisim. To solve it, another pass over the file is needed to check whether there are custom circuits. If the circuit has a custom appearance, the list of appearances is read and these coordinates are saved. When this circuit is encountered as a subcircuit, the name of that subcircuit is looked up and the different coordinates are calculated on a subcircuit base, taking into account the Anchor, the Location and the Ports. Eventually the correct values are found and saved in the element itself. When using this element to make the connections, it is checked whether this has a remapping defined and if so, the remapping is used to search for the pins, instead of just calculating the coordinate of the pin.

2.6 Configuration

The translator uses a lot of parameters that should be user-tweakable. At first a command-line interface was implemented, however, the interface got way too big and it took too much typing to generate a useful model. Also a lot of defaults might not be suited for every purpose, so an ini file was chosen to implement the configuration. The main advantage is that a set of defaults can be generated, saving a lot of typing. Apart from that, ini files are more easy to use and document then command-line parsing. What follows in this section is an enumeration of all possible ini variables and their influence.
However, to make it easier to use the program, a GUI has been integrated to set all these variables in a more controlled way.

2.6.1 Main

keyword	description	type
destination	The location that will hold the generated models, should always end with a '/'	string
importPath	The location of the PyDEVS module on the current system	string
experiment	Should an experiment be generated for this circuit?	boolean
verbosity	How much output should be generated?	integer
allowCollision	Should collisions be allowed? If not, colliding wires will be broken automatically	boolean
testing	Specifies whether this is a test run or not. Should NOT be manually toggled	boolean
interactive	Should an interactive version be made, allowing the user to inject data at run-time	boolean

2.6.2 Delay

keyword	description	type
base	The base delay of all elements (except these not present in Logisim)	float
clockScale	How much longer should a clock tick take, relative to the base delay	integer
random	Should the delays have some kind of (uniform) randomness to them	boolean
variation	If the elements should be random, how much should the variation be? The random values will be between $[base, base + variation]$	float
interactive	The delay between consecutive requests for input data for the interactive circuit	integer

2.6.3 Experiment

keyword	description	type
circuit	Specifies the main circuit, for the experiment to call. If 'None' or an unknown circuit is specified, the one saved in the circ file is used	string
end_time	How long should the experiment take, in simulation time	integer
verbose	Should the experiment itself be verbose	boolean
XMLtrace	Should the experiment output a XML trace	boolean
VCDtrace	Should the experiment output a VCD trace	boolean

2.6.4 Speed

keyword	description	type
dynamicChecks	Should run-time checks happen	Boolean
staticChecks	Should optional translate-time checks happen	Boolean
stopOnStaticFail	Should translation stop when the optional translate-time checks fail	Boolean

2.6.5 GUI

The GUI is implemented in Tk, which is included by default in Python 3. The main goal was to facilitate future integration with Logisim, as Logisim would otherwise have to handle the configurations. Since Logisim is rather graphical itself, it would not show a good integration if it were to require manually changing configuration files. Besides, a GUI also makes it easier for inexperienced users to do a translation. The configuration files can even be used for batch scripts, while the GUI can be used for standalone cases.
Since the GUI doesn't really add any functionality, it is just a basic form which will create a config file depending on the selected options and afterwards it will just call the translator with this config file. This might not seem the most optimal solution at first, though it should be noted that the configuration system can be kept unchanged by doing it this way, thus minimizing the impact of the GUI on the program and reducing coupling of the components.
To invoke the GUI, the translator should be run without any command line options, though the circ-file is allowed.

```
$ ./translate.py
$ ./translate.py circuit.circ
```

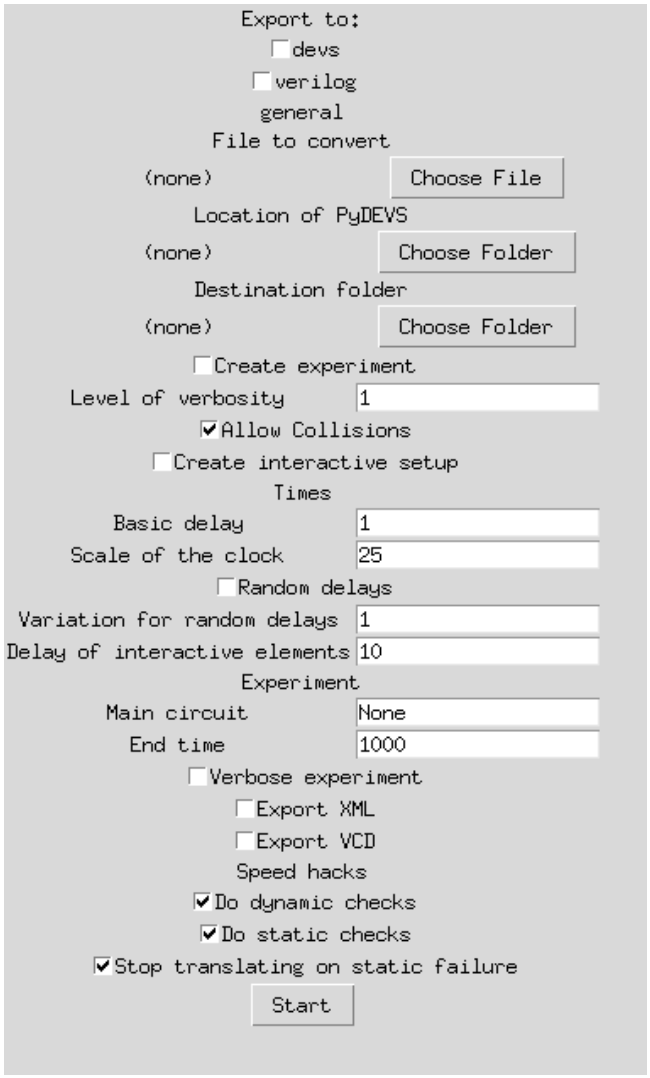


Figure 2.9: The GUI to the translator

Chapter 3

Checks

3.1 Translate-time checks

A lot of things that define a correctly functioning DEVS model can already be checked at translate-time. To prevent most problems in the resulting models (and possibly save time, as the error is identified a lot earlier), some checks are added to the translator.

3.1.1 Loops

The problem that loops could induce are very serious. It might be that a value gets passed infinitely, effectively halting the simulation. In DEVS, this is not a real problem thanks to the decoupling of 'receiving an event' and 'sending an event'. This way it is possible to prevent this kind of loops, on the condition that there is a non-zero delay in the elements.
This is because the delay will halt the loop momentarily and allow the simulation to progress. This means the simulation will eventually progress and finally the simulation will halt. An extra protection that has been added to every generated model is the cancellation of repeated signals. If a model receives the exact same signal on the exact same input port as the previous event, it will just discard this new event and continue as if nothing happened. Since elements do not save the value they sent on their outputs, the outputting model cannot check this. Most elements map different inputs to the same output, e.g. an OR port will output 1 if one of its input signals is 1, but it doesn't matter which of the inputs is 1. So it might be possible to have changing input signals, but a constant output signal.
When ignoring repeated input signals, there is another crucial advantage to simulation of Logisim circuits. Every physical element has some period of time that the input should stay stable to be able to output the desired value. When altering signals happen while the element is still in this time, it should forget these input signals and restart, since it couldn't output these anyway. However, when the incoming signals are identical, this should not happen, since in the real world nothing happened. This is just an artifact of DEVS, where an event is fired once, instead of a constant signal. Therefor, the best practice is to ignore the incoming value. Note that the PyDEVS simulator will still call the

timeAdvance() function, making it necessary for this function to calculate the time that was remaining.

3.1.2 Splitter loops

Contrary to the previous section, splitters are translated in such a way that loops might cause problems due to their delay equal to zero. To prevent this kind of problems, a list of visited elements and their ports is remembered. This way recursion can stop when the same element is visited.
Due to the construction of the intermediate tree, multiple splitters connected to each other, where it would have been possible to do it with only one splitter, might cause this check to go off. This doesn't really pose a problem, since only the suspected loop is broken, this doesn't break the correctness of the translator.

3.1.3 Bitsize

The translate-time check for bitsize might partially be a redundant test, since the same test always happens on run-time when a new signal is received (see further). This check might stop translating early if an error is detected, but it is quite possible that the user wants to translate a 'broken' circuit. This might be because some elements are limited in bitsize by Logisim (most of the time 32 bits), but if the user wants more bits, it would be rather difficult. If the user just constructs a circuit with a wrong bitsize at several places, the translator could translate these files and at the end, the user could simply change the bitsizes of the element.
It is rather difficult to make a circuit with non-matching bitsizes without knowing, as Logisim will immediately warn about the situation, making it a basic assumption that the user knows what is going on. Though the inattentive user might save it and try to translate it, resulting in an error at run-time. Since a translate-time check isn't that hard to implement, it is an optional feature. The user can specify in the configuration what should happen when bitsizes don't match and even if it should be checked or not (see section 2.6. When translating a gigantic circuit, checking bitsizes of all wires and elements might take some time, while the user might already know that it is perfectly safe since Logisim didn't notify him.

3.2 Run-time checks

Not all problems can be identified at translate-time. These checks have an influence on the incoming signal, since this isn't known at translate-time. One could argue that these tests are not really needed, since these assertions are always valid if the circuit was valid in Logisim. Though the project description clearly describes the possibility of coupling other DEVS models, that aren't known at translate-time. Also, if there would be some kind of bug with a wrong output signal, that is caused by an invalid incoming signal, it is way easier to raise an error and clearly explain what went wrong. Otherwise, the user would have to resort to manual checking of the signals, which could be quite tedious if all wires had a lot of bits and only one bit was missing on one wire.
However, these kind of checks might get expensive when continuously running

them, certainly when the user knows everything is safe. These dynamic checks are kind of like the preconditions of the model. Even though it is always safer to use preconditions in functions, there might be good reasons to turn them off. Most of the time when a wrong bitsize is received, the model itself will crash due to out-of-range indexing etc, so the probability of false outcomes is nearly zero. This suggested the configuration option to turn off dynamic checking, though it is highly recommended to leave it on for general purpose.

3.2.1 Bitsize

The most interesting and important run-time check is checking if the bitsize matches. This is a normal test, taking into account the above reason for run-time checks. Another important reason might be the possibility of logisim to save 'broken' circuits, in the sense that the wires might have different bitsizes. Since Logisim allows this, there is no guarantee that the supplied circuit files contain a consistent circuit.
One might argue why this should still be a run-time test, since it already gets checked at translate-time. It is important to note that the generated models will often cooperate with other models, not necessarily generated by this translator. This makes bitsize checks between these models impossible and might result in other errors, hiding the real cause of the problem.

3.2.2 Binary values

Another important check is to check if the signal is actually of the format that is expected. This means that the signal should comply to the following constraints:

1. The first character is a 'b'
2. All other characters should be either (character) '0', '1', 'E' or 'x'

This kind of checks is necessary since DEVS allows much more events to happen. To prevent this kind of problems, a check is recommended. The addition of a 'header bit' is to represent that the following is a binary signal [1].

[1] This notation complies to most representations of bitstrings, such as Python and VCD

Chapter 4

Export methods

4.1 XMLTracePlotter

All that had to be done for a decent XMLTracePlotter output is an implementation of the toXML() function in the model. For optimal flexibility, all variables that are saved in the state are exported in the XML file.

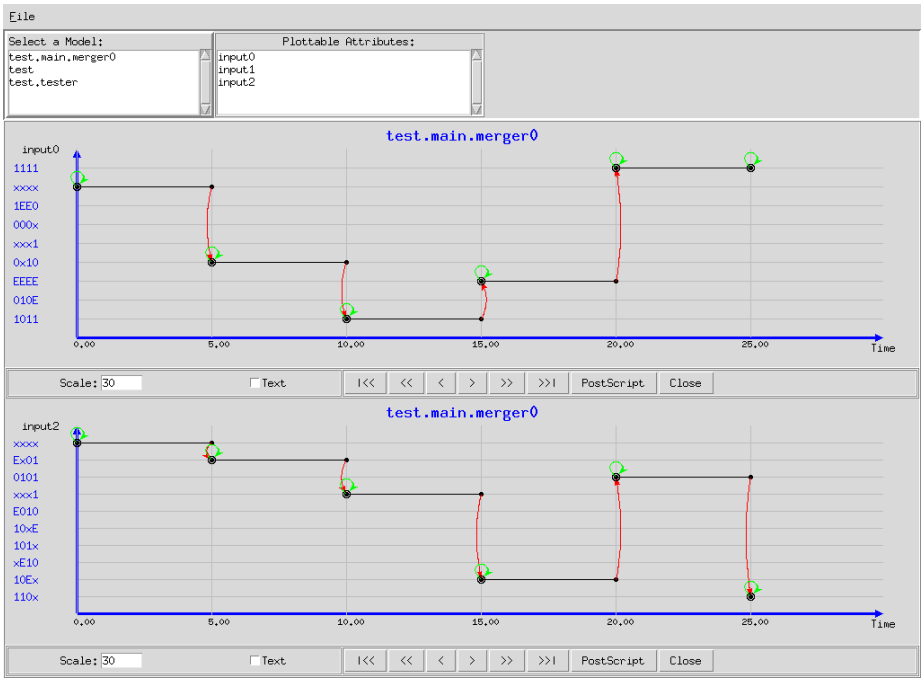


Figure 4.1: An example output in XMLTracePlotter

4.2 VCD

The export to VCD trace files happens analogous to the way XML files get exported. The only real problems that were encountered are mentioned in the following subsections.

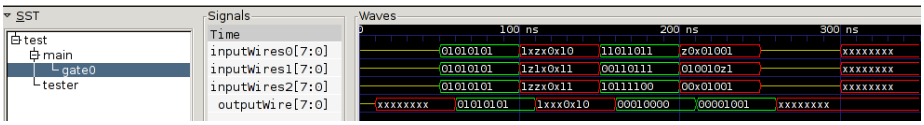


Figure 4.2: An example output in GTKWave

4.2.1 Data representation

The main models have been constructed according to the data representation of Logisim. However, VCD files require another. The main difference is in the error and floating signals. Therefore, a simple replacement is needed when saving the VCD files.

4.2.2 End time

Every VCD file requires an end time. This is the time that is written as the final line in the file. However, if the end time of the experiment file would be used, this might cause some ugly output in GTKWave[4] if the end time was just taken 'large enough'. Therefore, the final time is chosen as 10 further then the last DEVS-output. Most of the time, this would generate proper traces. Though 10 might sometimes be rather small in big traces, though this should not really be a problem.

4.2.3 Scale

A problem with VCD outputs is that a scale has to be specified, while DEVS does not really have a connection with real time. Therefor, a scale of '1 nanosecond' has been chosen. If this scale is inappropriate, it is rather easy to change this in the PyDEVS code or in the exported VCD file, just because there is no relation between this scale and the DEVS simulation time.

Chapter 5

Results

5.1 Comparison to previous work

As mentioned before, the same assumptions are made as in [7] concerning the different states, transition functions, delays and such. However, this bachelor thesis goes a little further, since it will translate any Logisim circuit to an equivalent DEVS model. This caused some problems that weren't present in [7], like feedback loops, colliding wires, pull resistors and in general 'strange' things a user might generate in Logisim that aren't supported in PyDEVS [1] , simply because it would be illogical to do so. However, in the assumption that the user knows what he is doing, these circuits should also be translated.
Also, the semantics of the models had to be exactly the same as Logisim, with the same functionality and behaviour. This was not necessary in [7], since they generated their own models with only the functionality that was required for the MIPS datapath, defining their own semantics and thus avoiding some hard parts like the splitters (in the general case).
Furthermore, the focus is different. In [7], the focus lies on making students understand a MIPS datapath using DEVS, including a very graphical environment. The focus in this thesis lies on translating between different representations of the same circuit, making it possible to integrate a Logisim circuit in any other DEVS model. So while the structure of the DEVS models is the same, what is done with these models differs.

5.2 Testing

The tests that were used were rather exhaustive. Tests include the standard input-expected output, regression testing of the generated VCD files and XML files and trace comparison to Logisim itself. This last one uses the brute force approach: it will just input all possible combinations and check the output. This causes the elements to be of rather small bitsizes though, since increasing the bitsize with 1 automatically quadruples the time required (and memory required, since the XML export list is kept in memory until the end).

[1] These problems and their solution are mentioned in 2.4

5.2.1 Translator

The comparison to the logisim trace files was an interesting test, since a lot of bugs were found. These bugs were mainly due to the fact that the Library Reference just wasn't deep enough and my assumptions of what should happen were different from the assumptions that the author of Logism had. Of course, some bugs of my own were discovered too, mainly due to the fact that the trace files is a brute force method, causing every line to be executed. Most of the found issues were rather small or just some case that was simply forgotten.

5.2.2 PyDEVS

When running the different tests, sometimes the comparison of XML files failed, while it passed a few seconds ago. This seemed rather strange, since all the files that were used were exactly the same. After comparing where it went wrong, it seemed that in some runs, the order of the models was different. Since the simulator should use a canonical form and the generated models didn't change, there had to be an error somewhere. When diving into the PyDEVS code, there was a sort to find the first model that can transition. However, a lot of times were exactly the same each time, because a constant delay of 1 was used in the tests. When printing the list of 'what' was sorted, this was a list of lists containing both the time found with the timeAdvance function and the corresponding event. However, when a draw appeared in the timeAdvance, which happened rather frequently, the second element was used for comparison. Since this was just an instantiation of a class, without any compare function defined, Python would just compare memory addresses. Since this can differ between different runs, this was the cause of the problem. When found, this problem was easily fixed by introducing a lambda function to check for the full model name instead of the address. This migh slow down the sorting a bit, since a string comparison has to happen and the getModelFullName function will be called a lot, however, the output will now always be canonical.
The line that caused the problems was:

```
cDEVS.eventList.sort()
```

The solution to this problem was:

```
cDEVS.eventList.sort(key=lambda i: (i[0], i[1].getModelFullName())
```

5.2.3 Logisim

The trace file comparison caused a lot of differences between the generated models and the Logisim elements behaviour, some bugs on behalf of the translator that weren't caught in the normal tests, but also a lot of ambiguities in the Library Reference that were misinterpreted when writing these models or that weren't mentioned. Furthermore, some elements exhibit rather illogical behaviour. An example might be the Counter element. The Library Reference states:

> In addition to the output Q, the component also includes a single-bit output carry. This is 1 whenever the counter is at its maximum and the load and count inputs indicate that the component should

increment on the next step - or when the counter is at 0 and the load and count inputs indicate to decrement at the next step.

However, when the Counter is at its maximum value and both load and count are 0, this would cause the element to stay at its current value and thus, logically speaking, the carry out should be 0. However, Logisim does actually set the Carry Out to 1 in this situation. This does not really violate what is being said in the Library Reference, but certainly goes against common sense. Note that when going in the reverse direction, so counting down, the carry out bit is being done correctly, making it extra strange.

Some other ambiguities exist for the Counter:

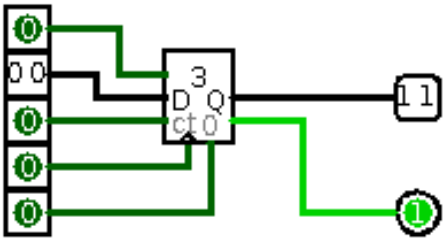


Figure 5.1: The strange output of the Counter element, both the Load and Count input wires indicate that the Carry out should be 0

1. What happens when the counter has a maximum value but a value greater than that one is loaded
2. What happens when an error/floating bit is present in the value to be loaded
3. What happens when an error bit is given as input
4. What happens when decreasing under 0 when a maximum value is specified

Most of these questions aren't only for the Counter element, but for other elements too. This is mainly due to the fact that the Library Reference just isn't thorough enough to be used as a complete definition of the semantics of the elements.

5.3 Profiling

Due to the long time required to run the traces, some basic profiling has been done on the code, using the Python module 'cProfile'. The tested circuit is the RAM trace circuit. The results indicate that most of the time, the simulator itself is responsible for the slow behaviour. A (stripped) version of the profile outputs follows, only the parts that take longer than 1 second in total are shown:

```
102592842 function calls (94325134 primitive calls) in 601.646 seconds

   Ordered by: standard name

   ncalls     tottime  cumtime  filename:lineno(function)
   115661     3.153    10.023   BitMergerb2.py:57(extTransition)
    32774     1.137    4.930    Counterb14i5tri...:67(extTransition)
  1406976     2.521    2.521    DEVS.py:107(type)
 16301534    25.275   25.275    DEVS.py:155(getModelName)
 14656494    86.372   109.065   DEVS.py:164(getModelFullName)
   425265    14.593   16.302    DEVS.py:220(poke)
  1017297    17.900   21.573    DEVS.py:244(peek)
   205630     4.520    8.008    DEVS.py:374(getSubModel)
  1671887     3.592    3.592    DEVS.py:519(type)
    32770     1.347    2.808    Shredderb14.py:68(extTransition)
   749978     9.563    23.169   copy.py:145(deepcopy)
   733591     6.072    7.878    copy.py:267(_keep_alive)
   205630     2.309    10.879   experiment.py:12(terminateFunction)
 11405239    16.131   16.131    infinity.py:54(__cmp__)
    39568     7.408    14.822   main.py:153(select)
   578776    13.941    84.100   simulator.py:163(receive)
   249360   204.430   587.818   simulator.py:318(receive)
  7284503    23.791   132.857   simulator.py:355(<lambda>)
   828136     6.777    589.682  simulator.py:521(send)
        1     1.066    601.627  simulator.py:531(simulate)
   723837     1.051    1.123    {abs}
    48991     1.671    1.671    {dir}
  1954353     2.606    2.606    {getattr}
  2069677     2.591    2.591    {id}
  1677316     2.678    2.678    {isinstance}
  1964356     2.516    2.516    {len}
  7876268    10.158   10.158    {method 'append' of 'list' objects}
  1483569     2.076    2.076    {method 'get' of 'dict' objects}
 14188839    20.277   20.277    {method 'keys' of 'dict' objects}
   227501    35.666   173.450   {method 'sort' of 'list' objects}
  7371970    48.000   58.490    {min}
   730683     1.843    1.843    {range}
```

The most resource-hungry function seems to be the 'receive' function, taking about 200 seconds (or 33%). This shouldn't be too surprising, since about 25% of the simulator code is this function.
Since the sort function uses a lot of different parts now that is being canonicalized, the functions 'sort', 'getModelFullName', 'infinity.__cmp__' and 'lambda' are the most important part of it, taking (in total) 173 seconds out of the total 600 seconds. Meaning that 29% of the time, the simulator is sorting the events. Notice that the generated models are only responsible for a few seconds of the total time, this will of course depend on which models are generated and which input is given.
Other circuits were also tested, yielding comparable results. On average, both the 'sort' and 'receive' functions are responsible for about 30% of the time each. Another test that was conducted is the difference between a constant delay and a variable delay. Normally, it would be expected that the variable delay would be somewhat faster, since it doesn't need to do the collision resolution. This test was conducted with the AND port. First of all, the constant delay profiling was done, revealing a total time in the select function of about 4 seconds (8

	Constant	Random
Total select()	4.0s	1.7s
Cummulative select()	8.0s	3.4s
Total time	221s	216s

Table 5.1: Difference between constant and random delays

	Constant	Random
Total time	690s	663s

Table 5.2: Difference between constant and random delays with verbose output

seconds cummulative). This would mean that the maximum time we could save would be around 8 seconds. The total time of the experiment was 221 seconds. The test with the random delay was a little faster, taking a total of 216 seconds, thus 5 seconds (or about 2%) faster. This profiling revealed that only 1.5 seconds were spent in the select function in total (3 seconds cummulative). Both tests were conducted in (nearly) the same circumstances, with no other resource-hungry processes running. This means that a random delay should not be chosen to speed things up, though it might be useful to do so when the verbose setting is set to True, since a collision would then cause a lot of extra (slow) output to the terminal [2] . For completeness, this comparison was also done. The results are not surprising: This means that, when using verbose output, using a random delay causes another 2% of speed-up, resulting in a total of 4%. This still isn't that interesting, though it might be when huge circuits would be translated. However, in these cases, the output would be way too fast to actually read it and probably a pipe would be used to e.g. a file. So when choosing between random and constant delays, the performance aspect shouldn't really be taken into account.
Note though, that the total time is drastically increased, since python now only used about 40% of the CPU, since the terminal and X required a lot of resources to keep up with the output.

5.4 My opinion

I am very happy with the result. All Logisim circuits (that I could think of) can be converted to DEVS without a problem. These models can even be used interactively, making it possible to inject data at run-time. This prevents the coding overhead of hardcoding all data that should be tested.
Most coding was done very soon, which allowed for a lot of testing and optimizations. I learned a lot about DEVS, which was a completely new concept when I started this project. Besides DEVS, I also learned to use Python and mainly using it in larger projects. Another thing that was new for me, was expanding an already existing project instead of writing everything from scratch. This gave me some insight in why decent coding, comments, documentation, patterns, version control, ... is really necessary in larger projects.
Sadly, as also mentioned in [2], the Logisim save file isn't according to any stan-

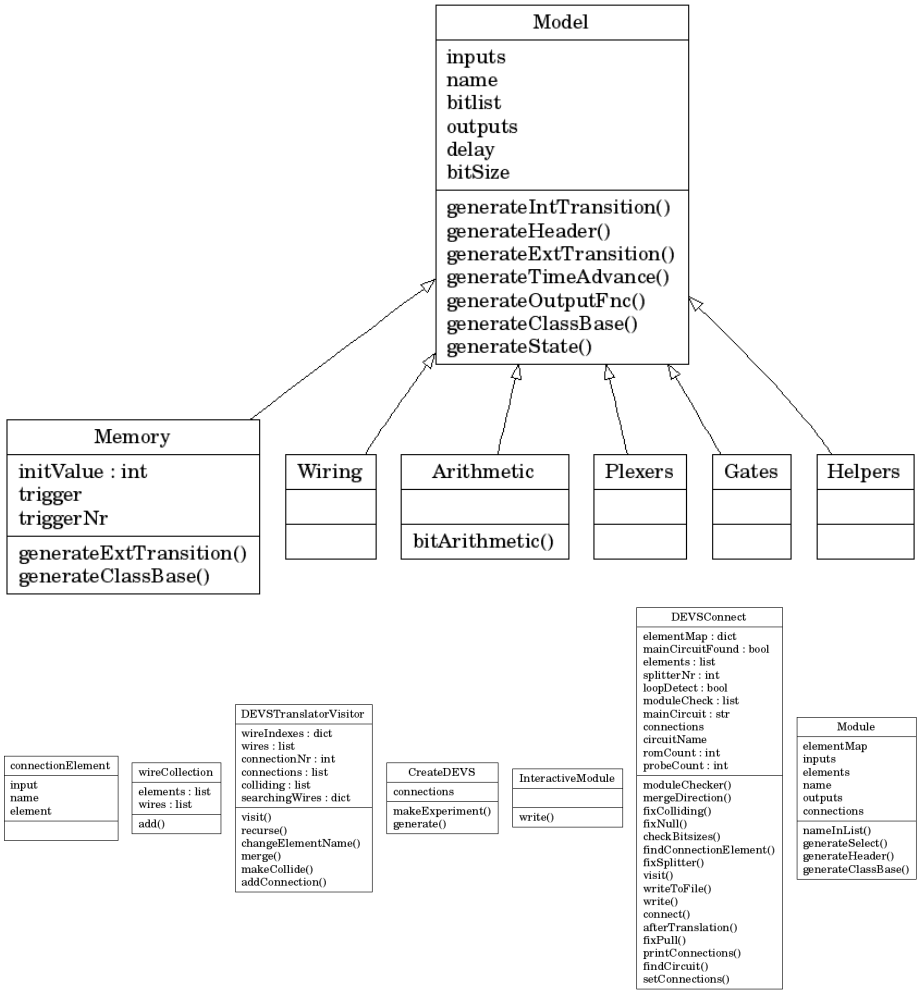
[2] This also reduces readability

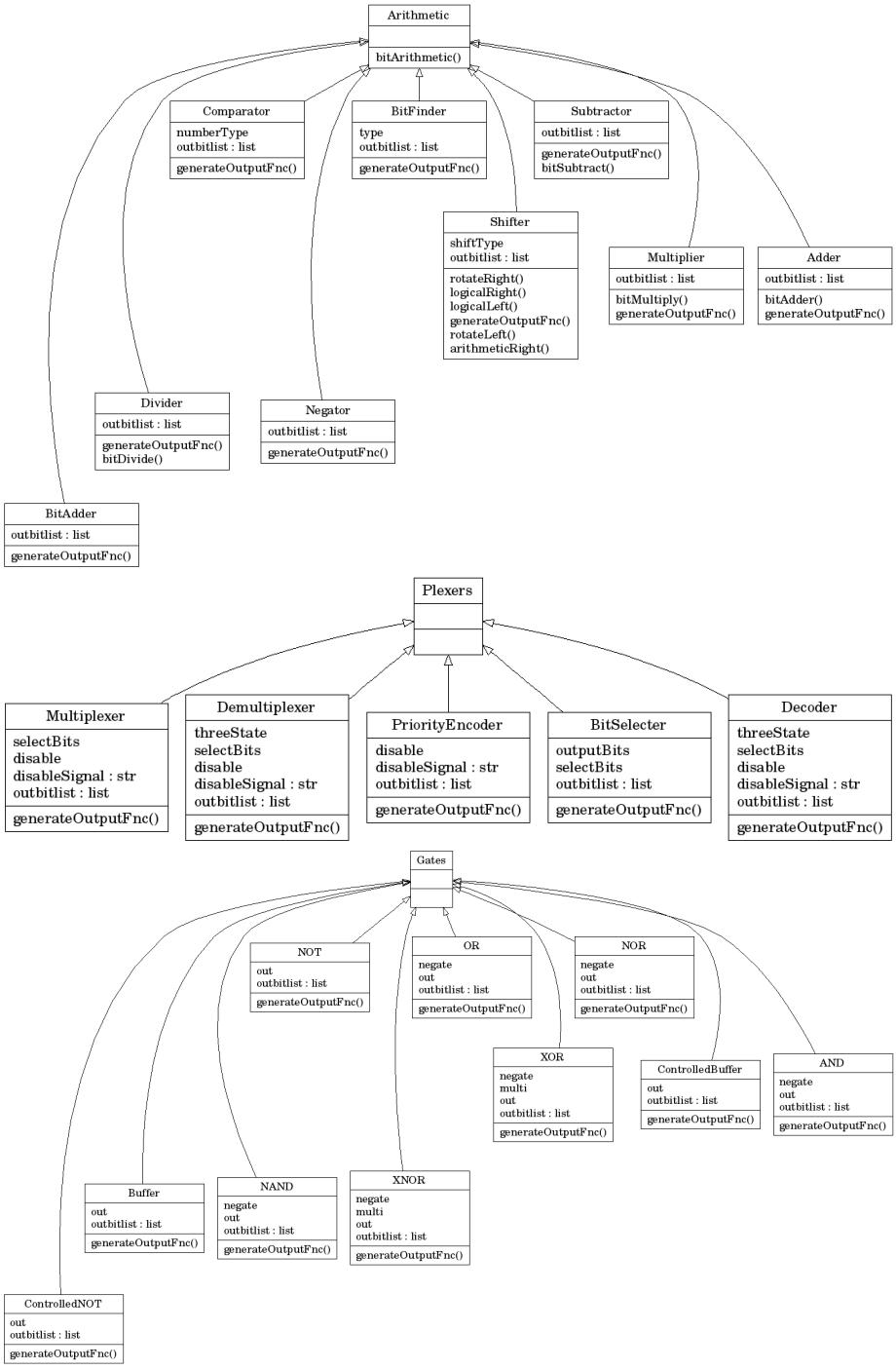
dard, neither is the semantics of the Logisim elements. So when a new Logisim version is released, it might be necessary to rewrite some parts of the parser and the semantics of the generated models. It might also be possible that extra elements are introduced, as happened in the transition between version 2.6.1 and 2.7.1.

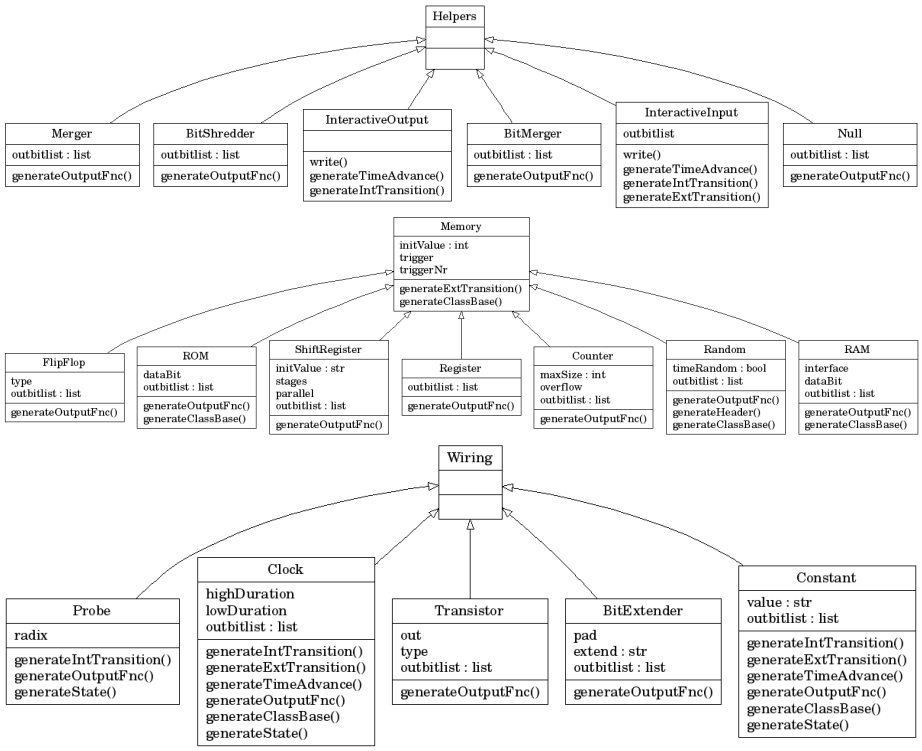
Chapter 6

Class diagrams

For all class diagrams concerning the intermediate tree, please refer to [2]. Only the DEVS-specific class diagrams are new, so only these are included here.




Arithmetic
bitArithmetic()
Comparator
numberType
outbitlist : list
generateOutputFnc()
BitFinder
type
outbitlist : list
generateOutputFnc()
Subtractor
outbitlist : list
generateOutputFnc()
bitSubtract()
Shifter
shiftType
outbitlist : list
rotateRight()
logicalRight()
logicalLeft()
generateOutputFnc()
rotateLeft()
arithmeticRight()
Multiplier
outbitlist : list
bitMultiply()
generateOutputFnc()
Adder
outbitlist : list
bitAdder()
generateOutputFnc()
Divider
outbitlist : list
generateOutputFnc()
bitDivide()
Negator
outbitlist : list
generateOutputFnc()
BitAdder
outbitlist : list
generateOutputFnc()
Plexers
Multiplexer
selectBits
disable
disableSignal : str
outbitlist : list
generateOutputFnc()
Demultiplexer
threeState
selectBits
disable
disableSignal : str
outbitlist : list
generateOutputFnc()
PriorityEncoder
disable
disableSignal : str
outbitlist : list
generateOutputFnc()
BitSelecter
outputBits
selectBits
outbitlist : list
generateOutputFnc()
Decoder
threeState
selectBits
disable
disableSignal : str
outbitlist : list
generateOutputFnc()
Gates
NOT
out
outbitlist : list
generateOutputFnc()
OR
negate
out
outbitlist : list
generateOutputFnc()
NOR
negate
out
outbitlist : list
generateOutputFnc()
XOR
negate
multi
out
outbitlist : list
generateOutputFnc()
ControlledBuffer
out
outbitlist : list
generateOutputFnc()
AND
negate
out
outbitlist : list
generateOutputFnc()
Buffer
out
outbitlist : list
generateOutputFnc()
NAND
negate
out
outbitlist : list
generateOutputFnc()
XNOR
negate
multi
out
outbitlist : list
generateOutputFnc()
ControlledNOT
out
outbitlist : list
generateOutputFnc()


Helpers
Merger
outbitlist : list
generateOutputFnc()
BitShredder
outbitlist : list
generateOutputFnc()
InteractiveOutput
write()
generateTimeAdvance()
generateIntTransition()
BitMerger
outbitlist : list
generateOutputFnc()
InteractiveInput
outbitlist
write()
generateTimeAdvance()
generateIntTransition()
generateExtTransition()
Null
outbitlist : list
generateOutputFnc()
Memory
initValue : int
trigger
triggerNr
generateExtTransition()
generateClassBase()
FlipFlop
type
outbitlist : list
generateOutputFnc()
ROM
dataBit
outbitlist : list
generateOutputFnc()
generateClassBase()
ShiftRegister
initValue : str
stages
parallel
outbitlist : list
generateOutputFnc()
Register
outbitlist : list
generateOutputFnc()
Counter
maxSize : int
overflow
outbitlist : list
generateOutputFnc()
Random
timeRandom : bool
outbitlist : list
generateOutputFnc()
generateHeader()
generateClassBase()
RAM
interface
dataBit
outbitlist : list
generateOutputFnc()
generateClassBase()
Wiring
Probe
radix
generateIntTransition()
generateOutputFnc()
generateState()
Clock
highDuration
lowDuration
outbitlist : list
generateIntTransition()
generateExtTransition()
generateTimeAdvance()
generateOutputFnc()
generateClassBase()
generateState()
Transistor
out
type
outbitlist : list
generateOutputFnc()
BitExtender
pad
extend : str
outbitlist : list
generateOutputFnc()
Constant
value : str
outbitlist : list
generateIntTransition()
generateExtTransition()
generateTimeAdvance()
generateOutputFnc()
generateClassBase()
generateState()

Bibliography

[1] Carl Burch. Logisim v2.7.1. http://ozark.hendrix.edu/~burch/logisim/index.html, March 2011.

[2] Naomi Christis. Logisim-to-verilog translator. http://msdl.cs.mcgill.ca/people/naomi/, June 2011.

[3] CompareVCD. Comparevcd. http://sourceforge.net/projects/comparevcd/, November 2011.

[4] GTKWave. Gtkwave. http://gtkwave.sourceforge.net/, February 2012.

[5] Hans Vangheluwe Jean-Sébastien Bolduc. Pydevs. http://msdl.cs.mcgill.ca/projects/projects/DEVS/, July 2002.

[6] Bill Song. Xmltraceplotter. http://msdl.cs.mcgill.ca/people/bill/0_research.html, December 2005.

[7] Hessam S. Sarjoughian Yu Chen. A component-based simulator for mips-32 processors. http://sim.sagepub.com/content/86/5-6/271/, September 2009.

List of Figures