Regressiontest - Automated Testing

To verify the continous work will be correct, a Junit regressiontest was implemented.

In these test a model (SCL) and the corresponding ESO file is needed. The test will generate a VHDL component from the model and a testbench from the ESO file and test them against each other. This will be done for all models that are stored in the models repository (if it is completely programmed 🙂).

What is needed

The main thing that is needed is the ISE Suite. It contains a compiler to compile the SCL-VHDL file and a simulator (ISIM) to test the SCL-VHDL-Model.

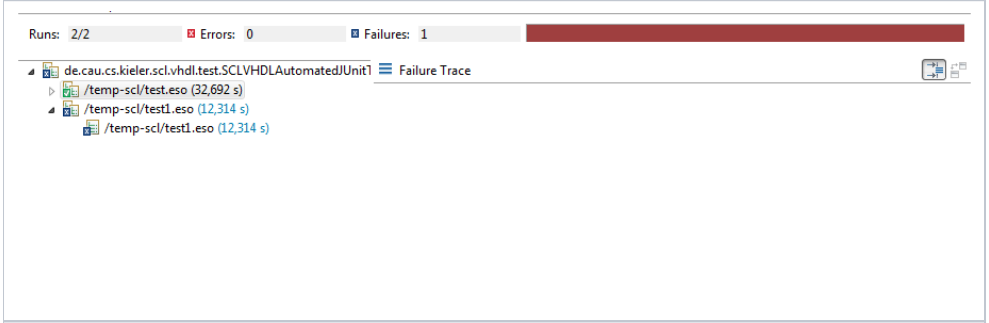
A very big advantage is that the ISE compiler and simulator can be controlled by a shell command. As we want to control and start this test with eclipse, this interface will be used. So Eclipse is additional needed. The whole test is a JUnit test.

Result

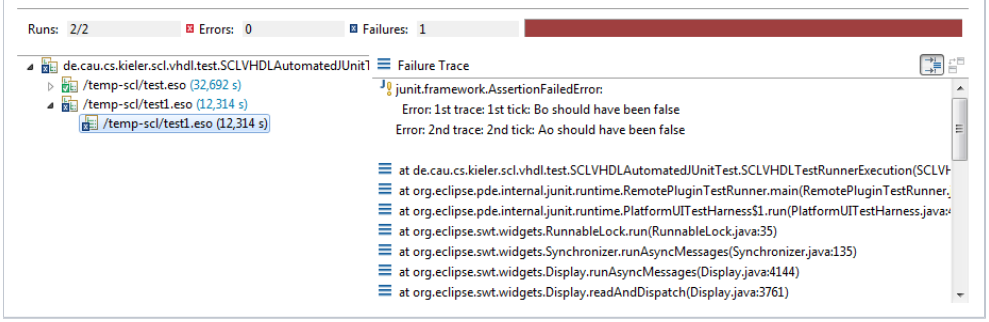
This JUnit test, test all models in the appopriate repository. For every test a JUnit error trace will be generated. By this way it is traceable which models testing fails or work.

Example: JUnit failure trace. Two models were tested *test.scl* and *test1.scl*

JUnit Test Without Failure (test.scl)



JUnit Test With Failure (test1.scl)



A very good thing is that the JUnit error trace contains the missed assertions. So it is exactly traceable where the failure appeared.

Technical View

Now a little more technical 😉

ISE Compiler

The ISE compiler is accessable through a shell command. To get the compiler to work, we must give the compiler a set of parameters.

A compiler command looks like this:

```
fuse -intstyle ise -incremental -o tb_test_isim_beh -prj test.prj

fuse: the ISE compiler
instyle: compile message level
incremental: build files incremental
o: object file (runnable exe)
prj: project file
```

The object file is an executeable file which is needed for simulation.

The project file (prj) contains all vhdl that are needed for the current compile process. In our test case these are all file that describe the model and the testbench file.

ISE Compiler

The ISE simulator is accessable through a shell command. To get the simulator to work, we must give the simulator a set of parameters.

A simulation command looks like this:

```
tb_test_isim_beh.exe -intstyle ise -tclbatch tes.cmd -log out.log -sdfnowarn

instyle: compile message level
tclbatch: a file which contains simulation information
log: specifies the log file
sdfnowarn: supress warnings
```

prj File

The project file contain all relevant vhd file which are needed for a succesfull compile. In this case the testbench file is also included because it should be tested afterwards.

Here an example of the project file is shown.

```
vhdl work "abo.vhd"
vhdl work "abo_tb.vhd"
```

The abo.vhd corresponds to the SCL model and contains the component that behaves like the model. abo_tb.vhd is the generated testbench from core ESO file.

cmd File

The command file contains simulation information. In this case we only need the time that the simulation must take.

Here an example of a command file:

```
run 1000 ns;
quit
```

batch file

To run everything automatically, a batch file was generated, which executes the compiling and simulation process.

Here an example of such a batch file:

```
ise_path="/C/Xilinx/14.5/ISE_DS/ISE/"
project="test.prj"
toplevelEntity="test_tb"
simulation_tcl="test.cmd"

export PLATFORM=nt
export XILINX=$ise_path
export PATH=$PATH:$XILINX/bin/$PLATFORM
export LD_LIBRARY_PATH=$XILINX/lib/$PLATFORM

binary="tb_test_isim_beh"
compile_params="-intstyle ise -incremental -o "$binary" -prj "$project
sim_params="-intstyle ise -tclbatch "$simulation_tcl" -log out.log -sdfnowarn"
tmp_out="sim_out.txt"

fuse $compile_params $toplevelEntity
"./"$binary".exe" $sim_params
echo -e out.log | cat out.log | grep 'Error:' | sed 's/at.*ps: //' >> $tmp_out
```

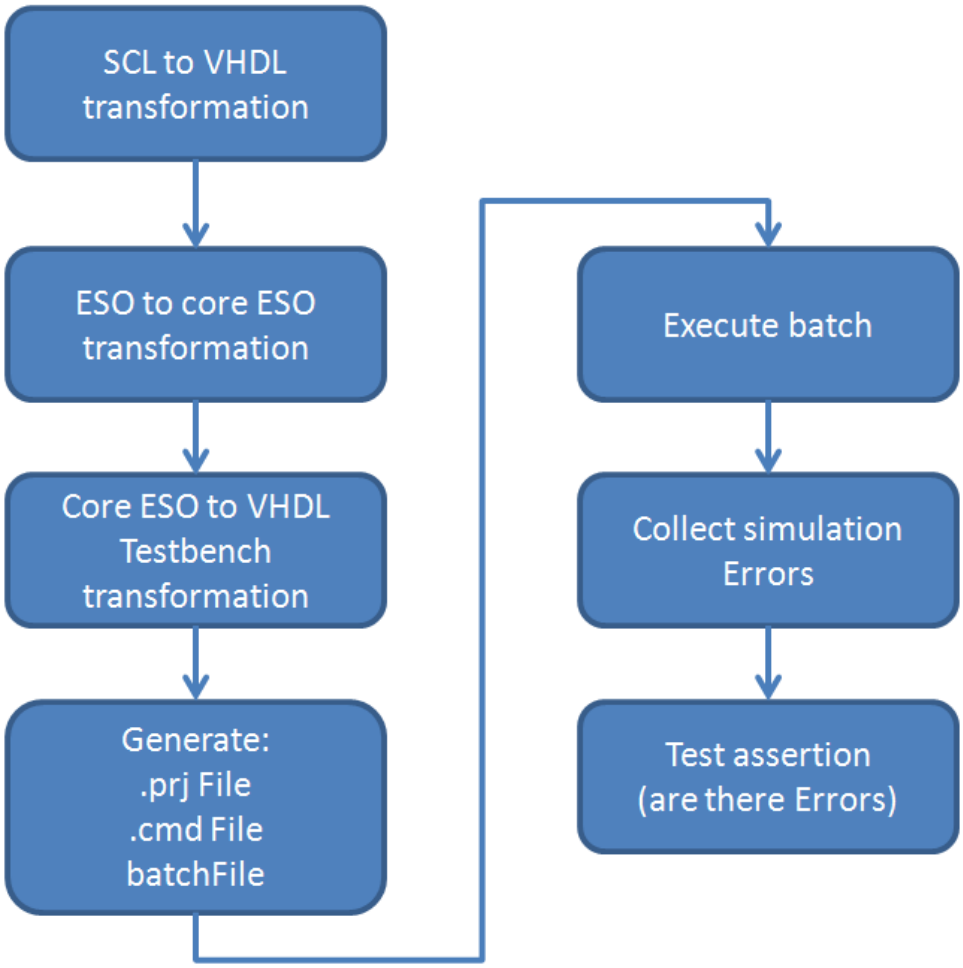
This file would not be explained in detail. In the first block the needed files will be assigned. In the second block the ISE Path will be set. In the third part the compile and simulation parameter are set and in the last part the compilation and simulation is executed.

Some small hints:

- toplevelEntiy specifies the toplevel entity, which should be used, in our case the testbench entity.
- sim_out.txt is the log file which will be used later, to fill failure information into the JUnit failure trace.
- last line: this line takes the simulation log file and performs a string operation which that pipes only the errors to the sim_out.txt

The proper Test

And how does the simulation works? Here is a little control flow diagram which expresses what will be done with each model that will be tested.



Some additional detailed information:

For every model which should be tested, a folder with its model name is created. The generated files will be saved to this folder. These folders won't be deleted after the test, they will be deleted befor a new test take place. So it is possible to look at these folders for more information if there are any errors.