TECHNICAL REPORT

An Introduction to the Esterel Studio Compiler for Constructing Reactive Systems

PROJECT: DECS

DESIGN-METHODS FOR EMBEDDED CONTROL SYSTEMS

Martin Zauner

The project DECS is funded by the FHplus programme of the Austrian Federal Ministries BMVIT and BMBWK and managed by the Austrian Research Agency FFG under grant 811414.

Revision History

Autor	Date	Last Changes
Zauner	2006-01-12	First draft
Zauner	2006-02-08	Document changes for clarification after correction by Roland Höller

Table of Contents

1 Introduction

The purpose of this document is to quickly become familiar with the reactive system design environment Esterel Studio by using the synchronous programming language Esterel. Originally Esterel development has been done by Jean-Paul Marmorat and Jean-Paul Rigault back in 1984. They invented an original information notation to solve their problems by expressing control algorithms in an ordinary and intuitive way. Later Gérard Berry, now Chief Scientist of Esterel Technologies, created Esterel's first formal semantics. Established on this semantics the first generated Esterel tool was used for/during experiments by companies such as AT&T Bell Labs, Dassault Aviation, etcetera. More about Esterel history can be found on the official Esterel-Technologies [E02] website.

2 Development Environment

To be able to construct and run a program designed and generated by Esterel, one out of two available software packages must be chosen. An short overview of available software is given underneath.

Name	***Version***	***Comments***
Esterel.exe	v5_92	Most recent free Esterel Compiler
Xesterel.bat	v5_92	Graphical user interface for free Esterel Compiler
Xes.bat	v5_92	Graphical simulation tool for free Esterel Compiler
Esterel Studio	v5.2.1e	Integrated Development Environment (commercialized)

Basically, two Esterel compiler version are available. Readily available is the free downloadable Esterel Version v5_92. This compiler version can be obtain either via the official Esterel website [E02] or by the academic website [E01]. Additionally to the free compiler version an extensive commercialized Esterel tool set (Esterel Studio, Compiler Verification Kit, etc.) is available from the Esterel-Technologies Company [E02].
Also included in the free compiler version is an graphical user interface for the compiler (xesterel) plus a graphical simulation tool (xes) constructed on the free development platform Tcl/TK. More information about Tcl/TK can be found in [T01].
After download and installation of the compiler version v5_92 a full functional (none restricted) synchronous programming development environment is ready to use.

2.1 Installation

The installation procedure for the free Esterel version compared to the commercial Esterel Studio is distinctive. Within this document the commercialized Esterel Studio Version is recommended because of its simple installation procedure, ordinary use and the available support by the manufacturer. Nevertheless, the free compiler version 5.92 can be utilized to get familiar with synchronous programming of reactive systems without paying any frees. Esterel compiler setup is discussed followed by the commercial Esterel Studio version. The installation steps described underneath are only given for the Windows OS, no installation comments about accomplishment on Linux are supplied by now.

Free Esterel Compiler

The free available software package downloaded (see [B02]) contains a Self-Extractor WinZip file. After double click the executable file the user will be prompted to choose a temporarily location where to unpack the compressed file for further installation. The executable suggests *c:\Temp\Esterel*. In case the folder *c:/Temp* does not exist the WinZip programme will create one. If another folder for unpacking the self-extractor file is chosen, no *c:/Temp* folder will be created. But in order to be able to use the compiler, Esterel compulsorily needs a folder named *c:/Temp* to store all the intermediate compiled files which are needed during the compile and build process. Hence, as long as a *c:/Temp* folder is available on the computer system no problems will arise during compilation procedure.

Environment Variables

Translating Esterel soures files into a sequential programming language such as C or even into a Berkely Logic Interchange Format (BLIF) from which a conversion to a hardware description language like VHDL or Verilog is possible (additional converters are needed), all compulsory environment variables must be set accordingly. Following variable is mandatory for the functional correctness of the compiler:

- %ESTEREL%

As soon as the graphical debugger *xes* or the graphical user interface *xesterel* shall be used, the variables underneath are mandatory as well.

- %TCL_LIBRARY%
- %TK_LIBRARY%

In general the program folder for Esterel contains a folder named *c:/…/Esterel/bin* where a lot of various batch files are accumulated. These files are associated with the

different executables, which are obligatory used while generating C or BLIF files. Within these batch files the above mentioned environment variables are set. Hence, as long as these batch files are utilized, Esterel and Tcl/TK environment variables will be set correctly.

In addition to the Esterel compiler, which is responsible to generate source code out of the synchronous programming language, the utilized compiler (e.g *gcc* with respect to Linux or the pentant *cl.exe* for Windows) and appropriate linker must be visible in order to instantiate these programs efficiently. Due to the fact that Esterel program compilation is executed in the context of a DOS shell, correct environment variables for the e.g. C compiler and linker must be set. For the Microsoft Visual Studio compiler *cl.exe* and linker *link.exe* following variables are obligatory:

- %VSINSTALLDIR%
- %PATH%
- %INCLUDE%
- %LIB%

Correct setting of the above mentioned environment variables are achieved by calling the batch file vsvars32.bat located within the Microsoft development software. E.g. as an example for the Microsoft .NET development suite the file can be found in the directory ..\Microsoft Visual Studio .Net 2003\Common7\Tools\. In order to use the Esterel compiler from the DOS box the batch file vsvars32.bat must be called in advanced before the compiler can be executed. Make sure you always call the batch file as soon as a new DOS box is in use.

Esterel Studio Suite

A much more convenient way concerning the installation procedure as well as Esterel project management can be found in the commercialized version of Esterel known as “Esterel Studio”. This version is not free but as a academic institution, special programs are offered to obtain Esterel Studio including SCADE (a graphical version of LUSTRE) for one year free. After the expiration date a license renewal is possible.
Before Esterel Studio can be installed, make sure a Microsoft Development Studio is properly set up. At least Microsoft Visual Studio 6 plus Service Pack 3 must be available in order to use Esterel Studio Simulator.
To start the installing procedure just insert the CD and run es_setup.exe. Be aware in case of a previous installation of the free Esterel compiler that the environment variable ESTEREL will be overwritten due to the Esterel Studio setup and therefore can only be used for the new development environment. Changing it back can be accom-

plished only by hand and can cause some disturbances (in case of wrong settings) within correct behaviour of the Esterel compilers.

Environment Variables

Basically a Licences Server for Esterel Studio and SCADE is running within the Technikum Wien Campus. Using the Esterel Studio the License Server must be reachable. This can be achieved be setting the local environment variable ESTERELD_LICENSE_FILE to <PortNumber>@<Server>.

1.1 <PortNumber> is the free TCP port number specified in the license file.
1.2 <Server> is the name of the computer where the FLEXnet[1] License Server is installed.

Inside Technikum Wien the <PortNumber> is set to 29030 and the <Server> name is given as “embsyslic”. Hence the variable ESTERELD_LICENSE_FILE must have the value 29030@embsyslic. Furthermore the server name “embsyslic” must be specified inside the “hosts” file locate in Microsoft Windows OS under the path C:\WINDOWS\system32\drivers\etc\. Following line must be added to this file:

- 10.128.202.250 embsyslic embsyslic

After successful setup the development suite “Esterel Studio” can be utilized.

3 Esterel Studio

This chapter gives an overview of how the Esterel workspace is divided and in what way the integrated development environment can be used to design, verify and generate code for Reactive Systems. The basic integrated menus and tools are explained in the light of the up/down counter example in the remainder of this document. For a comprehensive list of menus and tools about Esterel Studio refer to the User Manual [E03]. The command line version for the free Esterel compiler is not part of this document. This information can be obtained be reading literature [B03].

3.1 Workspace

Esterel Studio can be started by click on the appropriate link (Start > Program > Esterel Studio > Esterel Studio). The integrated development environment after start up

[1] Esterel Technologies products uses the FLEXnet license manager from Macrovision Cooperation for product license administration.

is portrait in Figure 1. Generally there are three different main windows, as described in Figure 1, which are identified and highlighted in yellow.

- In the ***Tree Window*** project file, modules and Safe State Machine are listed. Use this window for project management.
- The ***Document Window*** basically serves as a editor to create, view and edit Esterel source code, Safe State Machine objects and textual documents.
- Status, error, output etc. messages are displayed inside the ***Console Window*** to inform the user with various statements.

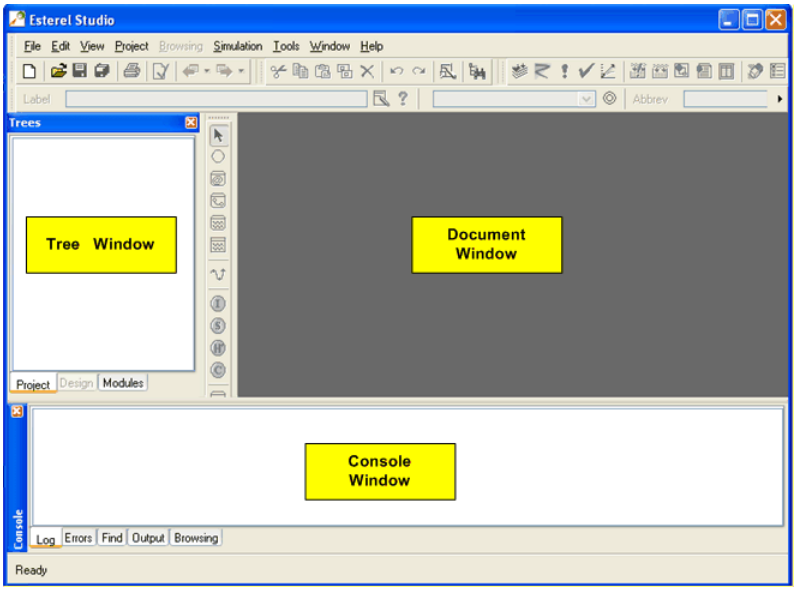


Figure 1: Esterel Studio Workspace

Extended information about the workspace and its features are presented during the given up/down counter example described in the following.

3.2 Project Creation

Before any new project can be created some specification must be made about what actually should be modelled, and later on source code should be generated either for software or hardware with the help of the synchronous programming language. Therefore a short verbal specification about the following project is given.

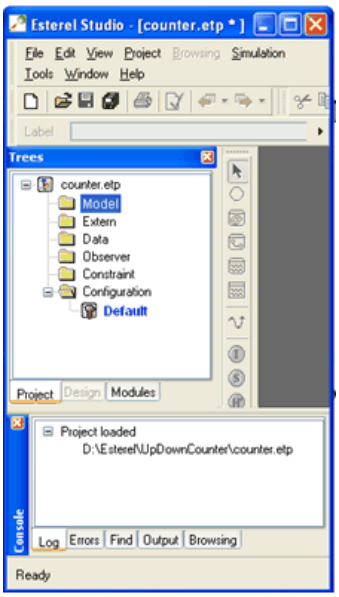
As a simple project an up/down counter should be designed and implemented in software on the evaluation board Phytec KC167CR. Depending on the environment the up/down counter should increase/decrease its value by one every second and emit the counter value to its environment. The range of the counter is given by the formula $2^N - 1$ whereby N is 8 bit wide and unsigned. Consequently the counter will wrap around at a value of 255 followed by 0 while increasing, and again wrap around at a value of 0 followed by 255 during decrease mode. The initial counter value is set to 0. Counter should count up when the up/down signal is true (1).

Interface

Input Signal	***UD***	(up/down signal corresponding to the counter direction)
Input Signal	***S***	(timer interval in seconds)
Output Signal	***Count***	(counter value to display)

Project

Before programming a new project must be specified within Esterel Studio. This can be attained by click on the menu bar to *File > New Project*. Choose a location for your project and name it **counter**. In Figure 2 the created project is visualized.



In the Tree Window the just created counter example is shown. There are 6 folders generated whereas meanwhile only 2 folders will be used. These are the *Model* and *Configuration* folders.

Right click on the *Model* folder will bring up a context menu to add a new file to the folder. A pop-up window appear. Now choose data type *Esterel Code (*.strl)* and store it in a suitable folder and under the file name *counter*. Right after storing, the file will be visible in the **Tree Window** under the folder *Model*. In the **Document Window** a empty file is shown. This is the Esterel source code document. Additional under *Configuration* folder an already defined configuration file (Default) is available.

Figure 2: Project Tree

In the **Console Window** a log message is displayed that the project "*counter.etp*" is loaded. Furthermore the default configuration file must be altered in order to simulate and compile the source code. In the first step no verification are applied on the model. Next, double click on the configuration file *Default* and the following window, as depicted in Figure 3, will be opened.

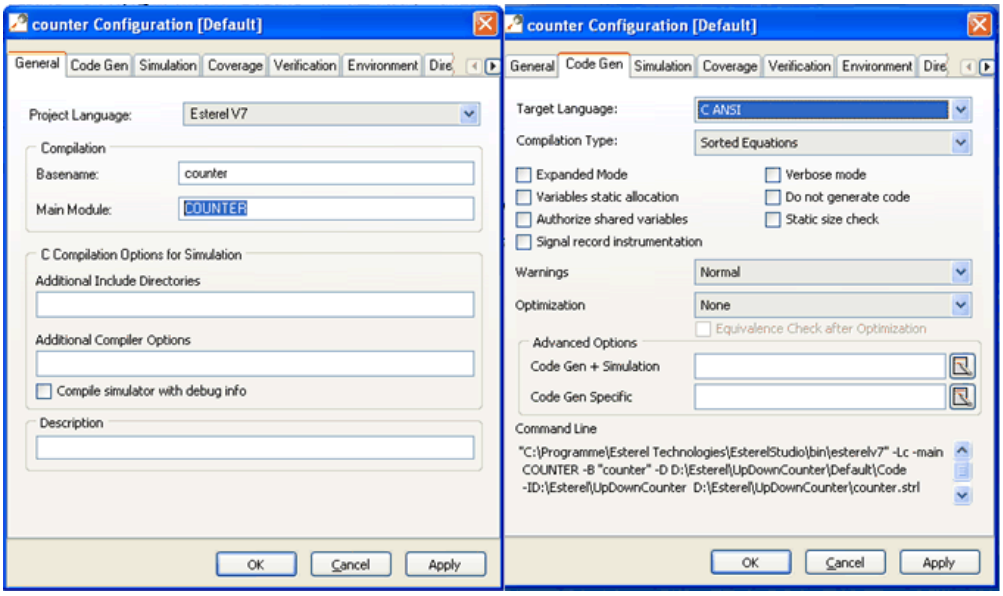


Figure 3: Configuration Properties

Make sure in the register card **General** *Compilation > Main Module* the Esterel main module name is entered. In this case the module name is called **COUNTER** (not necessarily case sensitive). In the second register card **Code Gen** select **C ANSI** as a target language. All other registers are untouched at the moment. The next step is writing the source code for the counter model and try to check the model in relation to internal correctness and completeness.

Figure 4 shows the Esterel source code as well as some errors and warning during first model check.

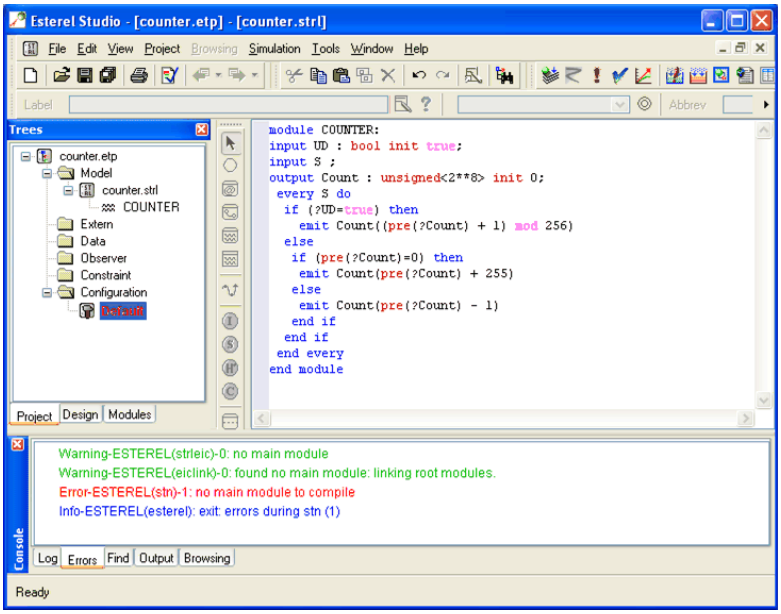


Figure 4: Counter Source Code

The reason why warnings and one error occurred is, because no main module name, as depicted in Figure 3, is provided. This is done on purpose to illustrated what will

happen if such a main module name is missing. Even when main is specified in front of the keyword *module* within source code, no warnings and errors will be detected while checking the model, but simulation compilation will fail with the following message: *Compilation stopped: main module required*. Setting a source file as main module can also be achieved by right click on the desired model → *Set As Main Module*.

3.3 Model Generation

The subsequent chapter deals with the source code description, source compilation as well as source code debugging issues.

Source Description

A textual description of the project plus interface is previously given in chapter 3.2. Now, through the use of Esterel Language a much more formal description (given in Appendix B) should be achieved and explained underneath. For a better understanding about fulfilling the achievable requirements the counter is illustrated in Figure 5 as a schematics drawing.

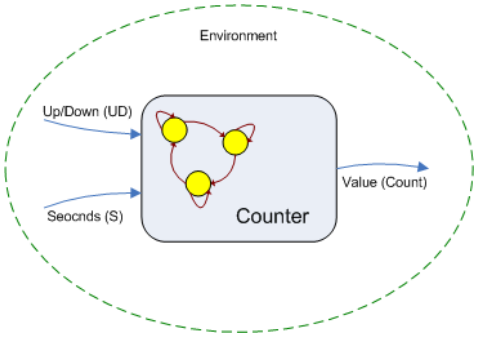


Figure 5: Counter schematics

Two inputs ***UD***, ***S*** and one output signal ***Count*** coming/going from/to the environment are specified. If and only if the value of signal ***UD*** is true and the signal ***S*** is present, the counter will count up. In case the value of signal ***UD*** is false and the signal ***S*** is present, then the counter will count down. In each case as soon as the signal ***S*** is available the output signal ***Count*** is emitted to the environment. When signal ***S*** is absent ***Count*** will not be emitted to the environment. In the beginning ***UD*** is initialized with true (means count up) and ***Count*** is initialized with zero. The declaration of ***Count*** with unsigned<2**8> equals 2^8, which signifies that only 256 positive values are allowed. Therefore unsigned<2**8> presents all the natural numbers for 0 to 255, not the set of number one can write with 256 bits, which is unsigned<2**256>.

The every statement, a simple delay expression is a Boolean test. The delay expression then elapses at the first instant in the strict future where the test is true. The emit statement is an emission of the valued signal ***Count***. Depending on ***UD*** or the valued signal ***Count*** itself, the instantaneous ***Count*** value is emitted to the environment.

The pre(?***Count***) data expression reads the value at the previous instant of ***Count***. The previous value is defined at the first instant that follows the value definition. If the signal has an initial value, it is also the initial value of pre(?***Count***). More details about Esterel language definition can be found in [E04].

Simulation Compilation

Before the source code written in Esterel will be compiled, a check of the generated model should be performed in order to satisfy all rules (like static cycles, etc.) associated with the synchronous programming language Esterel. An overview, where the most important properties of Synchronous Programming of Reactive Systems are discussed, can be found in [H01].
In Esterel such a check can be activated by pressing the button circled in red in Figure 6.

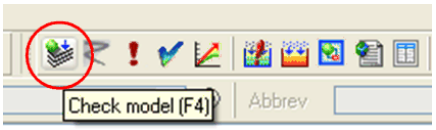


Figure 6: Toolbar "Check Model Funktion"

In case of an error, a message is printed in the **Console Window** otherwise, in the *Log* register the message "*Syntax analysis successful*" can be obtained. After a successful model check the project can be compiled to a so-called "Standalone Simulation"

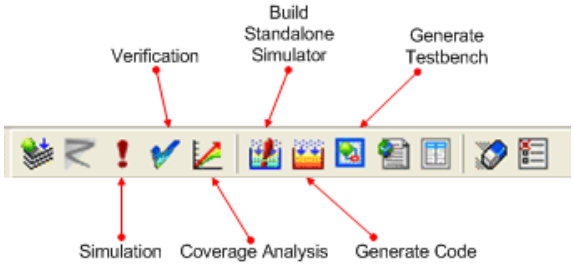


Figure 7: Toolbar Description

For that reason the red button "exclamation mark" illustrated in Figure 7 must be pressed in order to build and start the simulation. There are buttons available for coverage analysis, source code as well as test bench generation and a button for model verification. Additionally an automatic report generation is included. Advance, and more precise information can be obtained from [E03].

Simulation

Starting a project simulation at the first time without previously building the stand-alone simulator, the simulation will run through the complete compilation procedure first before starting the simulator. Past successful compilation the simulator looks like exemplified in Figure 8.

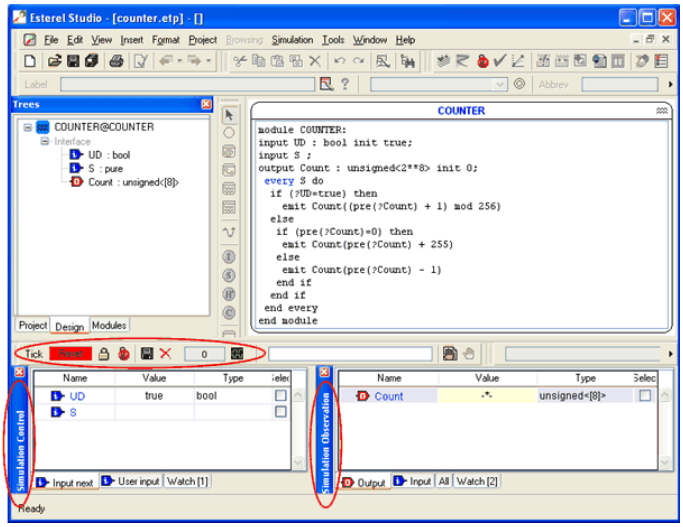


Figure 8: Esterel Simulator

The environment interface is clarified within the **Tree Window**. The input ***UD***, ***S*** and the output ***Count*** is shown. The bottom of Figure 8 reveals two additional windows (surrounded in red) named **Simulation Control** and **Simulation Observation**. In the control section all inputs and within observation section all outputs are listed. The tabs in the control and the observation panel can be used to sort signals. Simulation is started through the *Tick* button. Signal colors can be either red or blue. Blue means the signal is absent and can thus not be emitted in the next *Tick*, whereas red indicates that a signal is present and will be emitted with the next *Tick*. This corresponds to the *Pure* and likewise to the *Valued Signal* type. Additionally to the *Pure Signal* a value must be specified while using *Valued Signal* types. An example is given in Figure 8 which is the ***UD*** signal. This signal carries a *Boolean* value, therefore *TRUE* or *FALSE* can be specified. To change the status of any signal, left mouse click on the signal or right mouse context menu → *Emit*. Should the signal be emitted over a period of *Ticks* the user can specify this by using the context menu → *Keep Input*.

Furthermore the input and output signals can be visualized as a timing diagram by selecting *Simulation* → *View Session Waveform*.

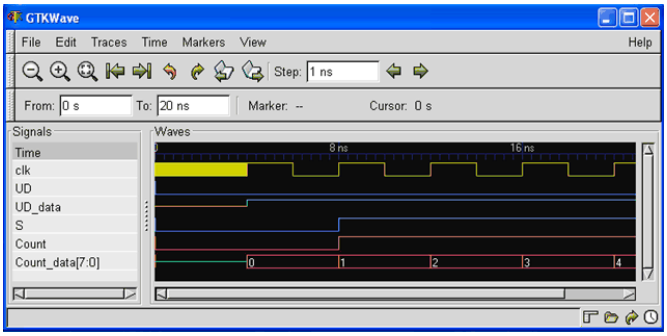


Figure 9: Session Waveform

The waveform window is presented in Figure 9. All input and output signals can be listed and altered in many ways. As the user progresses through the simulation elements of the Esterel language are highlighted, both in the **Tree**- and **Document Window**. Breakpoints, transitions, halt points and signals, the user wants to locate, are highlighted in following colors.

Color	*Description*
Red	Activated halt point for a state or emitted signal for a transition
Green	Transition active (i.e. trigger is present and condition is true)
Pink	Signal with an unknown status (emitted or not emitted)
Gray	Breakpoint
Orange	Signal highlighted with View Bindings
Yellow	Suspended

Table 1: Simulation Color Definition

Such a highlighted example is pictured in Figure 10. In the **Tree**, **Document** and **Simulation Control Window** some elements are highlighted in orange to symbolize *View Bindings*. ***Count*** signal is emitted and therefore colored in red within **Simulation Observation Window**. The input signal ***S*** is also colored in red and additionally underlined which indicates that the signal keeps its value for further *Ticks*.

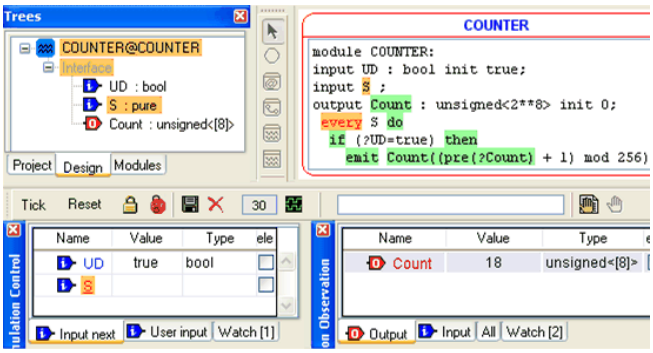


Figure 10: Simulation Highlight

All elements colored in green inside the **Document Window** signifies, in case of the counter example, that the signal ***S*** is present (active) and the condition of ***UD*** is true. As a result, the condition of signal ***Count*** is evaluated to true, hence ***Count*** is emitted.

Source Code and State Pair Coverage

Before Source Code Coverage can be put into practice a few modifications must be done in the **Tree Window** under *Configuration*. Thus the default configuration file (see Figure 2) can be duplicated after simulation is stopped in order to switch between various configurations without doing a lot of modifications and consequently impede miss configurations. Enabling one specific configuration file is done by right click on the file → Set Active.

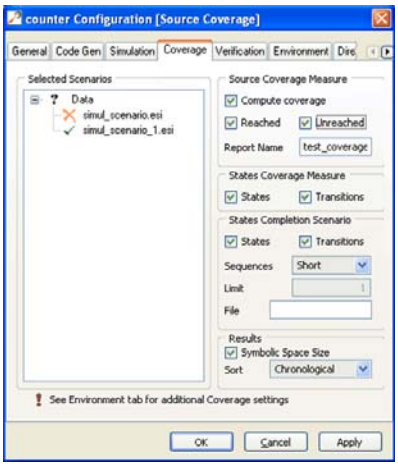


Figure 11: Source Code and State Pair Coverage Configuration

Source Code Coverage is used to display and obtain statistics about what part of the source code is reachable and finally reached during computation. As shown in coverage configuration (see Figure 11) the appropriate checkboxes under *Source Coverage Measure* must be activated in order to perform coverage analysis. Additionally, turn on corresponding simulation scenario by activating the recorded simulation file under the region *Selected Scenarios* in Figure 11. If no scenario file is available, a model simulation must be performed (see chapter Simulation) in order to save the simulation as a scenario file. This is simple done by just saving the performed simulation to a file.

For a better visibility to highlight reached, unattained or dead source code following text coloring are used throughout Source Code Coverage.

- *Grey*: Dead code – not reachable
- *Blue*: Reachable code that has not yet been reached
- *Purple*: Code successfully reached at completion of computation

Starting Source Code Coverage while performing e.g. a recorded simulation is done by pressing the *Start Source Coverage* button in the Simulation Coverage toolbar or through the main menu *Simulation* → *Start Source Coverage*. An example source coverage analysis is presented in Figure 12.

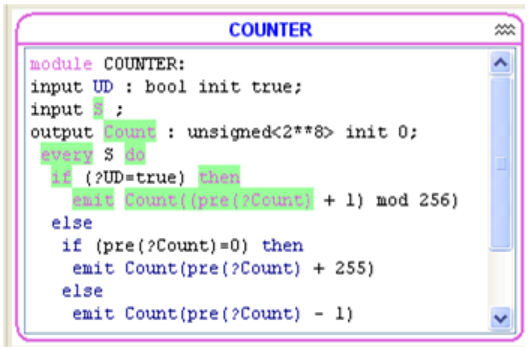


Figure 12: Highlighted Coverage Source Code

Here in Figure 12 it is obvious that some portion of the source code (*purple*) has already been reached while other parts of the source code (*blue*) are not yet been reached. Finally, after the simulation has ended the source coverage must be stopped. Consequently, the Source Code Coverage Report can be saved to a location provided by the user. A possible coverage report for the introduced model in this paper is presented in Appendix A. More detailed source coverage information can be extracted form [E03].

In addition, State Pair Coverage can be applied in order to ensure that all possible states (***RSS*** – ***R***eachable ***S***tate ***S***et) and transitions (***RPS*** – ***R***eachable ***P***airs of states ***S***et) have been attained. The importance of State Pair Coverage is underlined by the following statement

> *When you measure and complete coverage, you test the global state machine (GSM) of your system, or a 'flattened' version of your system where each state is a combination of all states of the system components. That is, you test all possible combinations of states and transitions, not just the ones in a single automaton or a collection of sequential components in a Safe State Machine. A model is considered to be completely tested when the set of simulation scenarios used to test the model (the Global State Machine) have reached all states in the Reachable State Set (RSS) or the Reachable Pairs Set (RPS).*

which was taken from the Esterel User Manual given in [E03] on page 220. Subsequently, for a better understanding the term RSS and RPS are explained The given definitions are extracted from [E03] as well.

Reachable State Set:

> *An Esterel model execution uses binary registers grouped in a state bit vector to encode the system global state. After every tick (reaction of the system), the global state is reevaluated based on the inputs and the events, and the current state bit vector modified accordingly. The set of possible bitvector configurations is called the Reachable States Set (RSS). Each bit vector configuration in the RSS represents a unique state of the model.*

Reachable Pairs of State Set:

> *State enumeration does not test a model completely because it only tests a static view of the model. To test a model completely, its transitions - the capability to go from one state to another - must also be tested.*
>
> *A transition is defined by a start state, an end state, and an event that makes the transition possible. One can decompose the transition into the pair of states (start, end) and the event. The theoretical transition number is infinite because no limitation to the event expression capability exists; however, the possible pair of states cannot be greater than N2 where N is the number of states. The set of possible pair of states is called the Reachable Pairs Set (RPS).*

To be able to perform such State Pair Coverage, configuration must be set accordingly to Figure 11. More configuration details of State Pair Coverage is presented in [E03]. Not to forget, a previous recorded scenario file (this is done either by storing a simulation to a file or by performing a verification) must be available. The execution is accomplished by pressing *Project* → *Coverage Analysis* from the main menu.

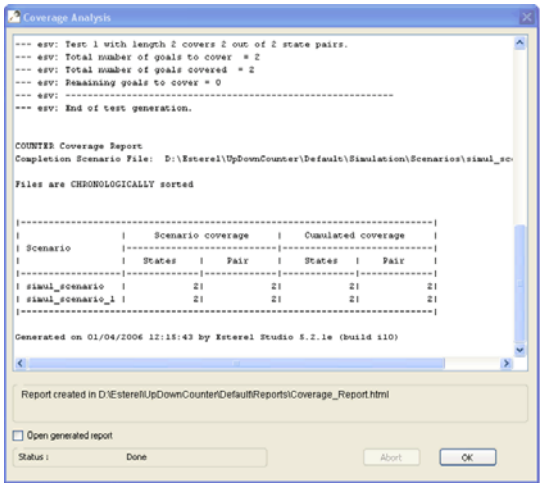


Figure 13: Screen Shot of Coverage Analysis

The result of State Pair Coverage is depicted in Figure 13. Moreover, a Coverage Report is automatically generated which summarizes the Coverage Analysis in some way. An example report can be found in Appendix A.

3.4 Model Verification

While still in the design phase Esterel Studio allows to detect corner-cases, which are probably not covered during Simulation, Source Code and State Pair Coverage. As a result, Esterel Studio has a build in design verifier, which implements formal verification techniques. The reason for such design verifier is that it can be very hard to detect enough input sequences for any reasonable sized design, while simulating, in order to determinate the expected behavior. Esterel-Technologies states the need of formal verification techniques by saying:

> *Formal property verification is an alternative and complementary approach to functional verification of hardware designs. Using mathematical methods, it*

> *can be proved that a given property of the DUV (Device Under Verification) cannot possibly be violated by any input sequence. The method implemented by the Design Verifier is known as* ***Model Checking****. In model checking all behaviors that could potentially lead to the violation of a given property are exhaustively examined. If no such behaviors can be found the property is proved correct. On the other hand, if there is a behaviors that violates the property, the model checking procedure generates a counter-example, that is, an input sequence demonstrating the violation of the property.*

This statement was taken form [E03] on page 196. Descent information about model checking and its advantages are introduced in [CGP01]. The generated counter-example can be used as an input scenario for the simulator to facilitate the investigation of the violation.

Two possibilities are available to form a more precise environment activity if needed when verifying a model. Such possibilities are available as environment constraints file and observer files. These files can be written either in Safe State Machine (.scg) or Esterel (.strl) format. Depending on the file, which is created in the **Tree Window** click right on *Observer* or *Constraint* folder and add a new file as illustrated in Figure 14.

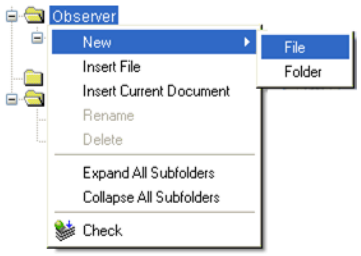


Figure 14: Add Observer File

During verification within the counter example an observer file is used to demonstrate its utilization. The source code is offered in Appendix B. After adding the observer file to the project, it must be made available in the *Active Configuration* within tab *Environment*. Double-click on the displayed observer file and finally click apply to activate the observer for the verification process. How to design an environment constraint or a observer file see literature [E03]. Next step in carry out a verification process, one out of two verification engines must be specified by activating the corresponding option field in *Active Configuration* → *Verification*. Inside the counter example project the full model checking based on SAT-technology (more about SAT see [I01]) is used. Information about the two different verification engines and there appropriate usage is discussed in [E03] on page 203 as well as in [CGP01].

Verification process

Launching a verification process is done by choosing Project Verify or pressing the toolbar verification button shown in Figure 7. After verification computation is com-

pleted the following window given in Figure 15 is shown. ***Note***: In order to get the same screen as shown in Figure 15, extend the Esterel source code as exposed in Appendix B capture 6.2.

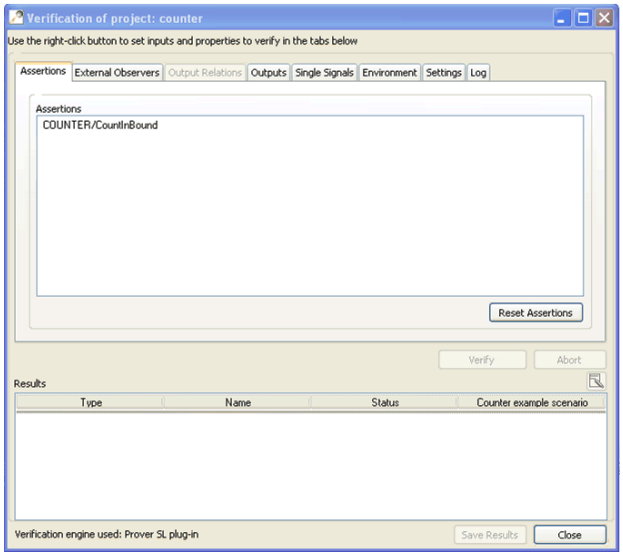


Figure 15: Project Verification

Formulating verification questions depends on the selected properties. Basically, there are six possibilities to select specific properties:

- Asserstions
- Externel Observers
- Output Relations
- Outputs
- Single Signals
- Environment

Assertions

To add an assertion to the counter example copy the source code under caption Counter Source Code including Assertion from Appendix B to the design. Consequently, in the **Verification Window** under the tab *Assertions*, the specified assertion is listed. Check the given assertion with the right click → *Always true?* to verify if these special logic signal is always true. No signal verification can be achieved by right click on the signal → *Not checked*.

External Observer

Within this paper an observer is appended to the system model. To make the output of the observer available to the verification session change to *External Observers* tab right click on the counter observer signal *OverUnderFlow*. Potential questions for observers are shown in Figure 16.

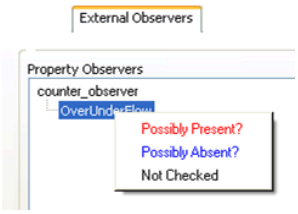


Figure 16: Verify External Observer

Output Relations

The design represented here does not include Output Relations. As soon as Output Relations is part of the design, they cover predicates on output which must always hold. Further information are exposed by [E04] on page 73.

Model Outputs

In order to check the model output if they are possibly present or absent during verification go to the Output tab and select the corresponding output signal. Right click to select the condition of an output's status.

Single Signal Emission

If two and more emissions of the same incarnation of a single valued signal is performed, a run-time error is produced. To detect the presence of such multiple emissions go to the *Single Signals* tab, right click on the signal, choose "*Always Signal Emission?*".

Environment

The environment tab contains two fields labeled as *Model Inputs* and *Environment Constraints*. Use the labeled field *Model Inputs* to directly constrain the offered inputs to either *Always Present* or *Always Absent*. Additionally, if a constraint file exists, this can be seen under the second labeled field named *Environment*. No selection can be made in this panel.

Settings and Log

As described earlier in this paragraph under tab *Settings* one out of two verification engines can be chosen. Within the *Log* tab information produced during verification process are displayed.

At the bottom of the *Log* tab verification results are revealed after clicking the *verify* button. In the *Results* panel the type of property e.g. *Assertion*, *Output*, etc., the property name, as well as the status e.g. *Always True*, *Always Single Emission*, etc. is published. Additionally, in case a formulated question evaluates to e.g. *Possibly absent*, *Potential multiply emission*, etc. a counter example is produce to lead to the possible misbehavior of the model. By double-clicking on the generated counter ex-

ample a simulation can be started in order to follow the path which directly escort the model designer to the possibly fault behavior. In depth information are presented in [E03], chapter Formal Property Verification.

3.5 Code Generation

This chapter deals with the target dependable code generation as well as test bench generation. Only basic issues are covered, detailed information are found in [E03].

Target Code

Past a successful model generation and verification, embedded code generation can be carried out. Thus the corresponding target language must be selected in the *Active Configuration* under tab *Code Gen*. Esterel supports different target languages such as ANSI C, C++, VHDL, Verilog only to mention a few. An illustration about different types of target languages is also given in Figure 17.

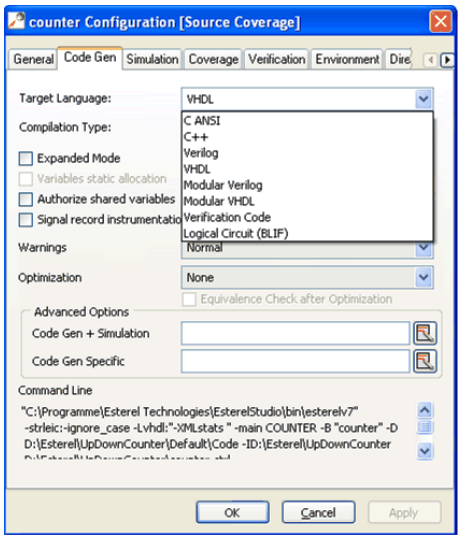


Figure 17: Code Generation

After selecting the desired language, code can be generated by choosing Project Generate Code. A short cut for generating code is also depicted in Figure 7. More useful information about generating embedded software, non-modular hardware and modular hardware is introduced in [E03] beginning with page 259.

Testbench

Esterel Studio can automatically generate executable code which is used for a test bench to deliver a predetermined input sequence into a design, to in return, observe its response. To be able to generate a test bench, an ESI scenario file (see chapter Source Code and State Pair Coverage) must be available as an input file.
A test bench is composed of the following components:

- ***Stimulus generator:*** Generates test inputs to stimulate the design.

- ***Checker:*** Checks what outputs are expected for a valid design.

Figure 18 illustrates the design flow for a test bench and its model usage.

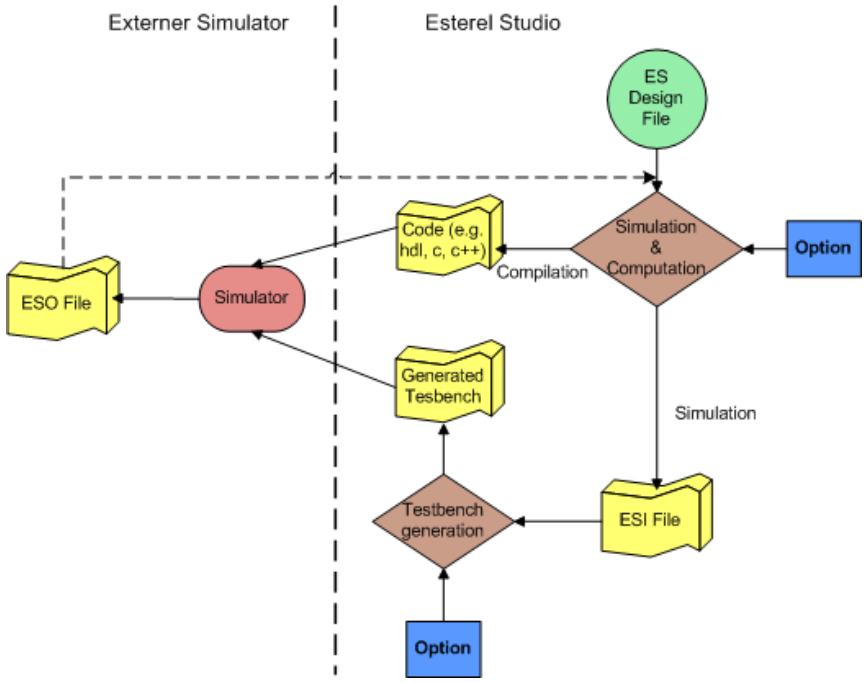


Figure 18: Test Bench Design Flow

As illustrated in Figure 18, simulation and computation as well as test bench generation is done in Esterel, while the generated code and test bench is used as an input for an external e.g. HDL simulator such as ModelSim. The target language specified for the test bench depends on the *Active Configuration* settings configured under the *Code Gen* tag, hence test bench generation for target language such as C or C++ is supported too. For the given counter model, test benches used for ModelSim (VHDL), Visual Studio (ANSI C) and Keil µVision 2 (ANSI C) were created. Figure 19 presents the Esterel simulation waveform outcome (upper part of Figure 19) which is compared with the external simulation waveform produced by ModelSim (lower part of Figure 19). Example test bench outputs generated by Visual Studio and Keil µVision 2 development environment are presented in Appendix C.

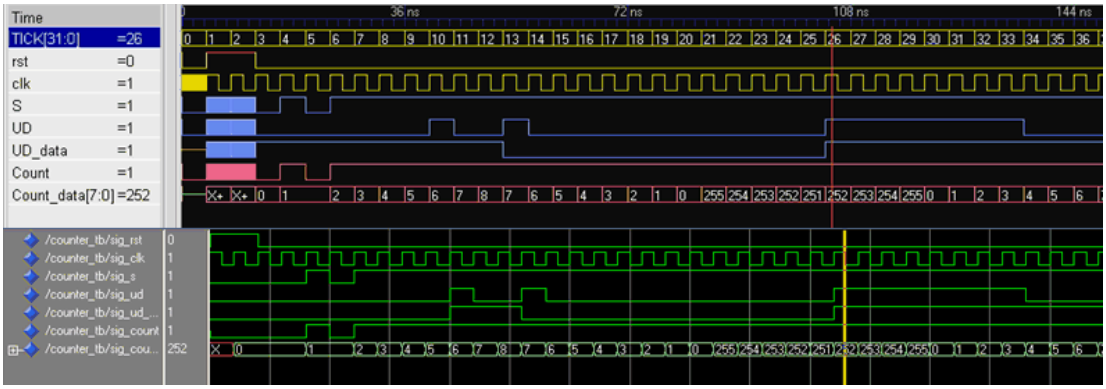


Figure 19: Esterel vs. ModelSim Waveform

ESO Co-execution

During the external simulation session in ModelSim and the ability of Esterel Studio to instrument HDL code and write ESO (Esterel Simulator Output) files, the performed external simulation session can be visualized within the Esterel model whereas the simulation control is still handled by the external simulator. Co-execution is done by activating *Signal Record Instrumentation* within *Active Configuration* → *Code Gen*. After HDL source code and test bench generation, the model must be simulated in the external simulator. Right after the simulator is started an xxxxx.eso file is created where all dynamic inputs and outputs are written to. Before continuing with the external simulation, an Esterel simulation must be started and the just created ESO dump file must be loaded (*Simulation* → *Connect To File*). Past starting the simulator, Esterel monitors the ESO file in order to provided dynamic information on the scenario content like number of step in the scenario and the current step in particular. The ESO Co-execution can be observed (dashed line) in Figure 18 which runs from the *ESO file* to *Simulation & Computation*.

4 References

[B01] Gérard Berry, *The Foundations of Esterel*, Ecole des Mines de Paris and INRIA, Route des Lucioles, 2004 (http://www-sop.inria.fr/esterel.org/home.htm)

[B02] Gérard Berry, *The Esterel v5 Language Primer Version v5_91*, Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, Route des Lucioles, 2004 (http://www.esterel-technologies.com/)

[B03] Gérard Berry, *The Esterel v5_91 System Manual*, Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, Route des Lucioles, 2004

[CGP01] Edmund M Clarke Jr., Orna Grumberg, Doron A. Peled, *Model Cecking*, Lucent Technologies, 1999, ISBN 0-262-03270-8

[E01] Esterel.org, *Academic Esterel Homepage*, Free Download of Esterel program, documentation and examples (http://www-sop.inria.fr/esterel.org/)

[E02] Esterel-Technologies, *Commercialized Esterel Homepage*, Additional Esterel Tools and documentation (http://www.esterel-technologies.com)

[E03] Esterel Studio, *User Manual*, Esterel Technologie SA, 8 rue Blaise Pascal, 78990 Elancourt, France, Revision: ESUM- ET/0128u1-ES5.2 (http://www.esterel-technologies.com)

[E04] Esterel Studio, *Language Reference Manual*, Esterel Technologie SA, 8 rue Blaise Pascal, 78990 Elancourt, France (http://www.esterel-technologies.com)

[H01] Nicolas Halbwachs, *Synchronous Programming of Reactive Systems, A Tutorial and Commented Bibliography*, Verimag, Grenoble, France

[I01] Internet, *The International Conferences on Theory and Applications of Satisfiability Testing (SAT)*, (http://www.satisfiability.org)

[T01] Tcl/TK, *Tcl Developer Xchange*, Tool Command Language (http://www.tcl.tk)

5 Appendix A

5.1 COUNTER Source Code Coverage Report

Generated on 12/29/2005 10:43:07 by Esterel Studio 5.2.1e (build i10)

List of scenarios files:

Scenario file name	Number of ticks played	Total Number of Ticks
simul_scenario_2.esi	9	9

Number of manually played ticks: 0
Total number of ticks played: 9

Reached statements:

Statement key-word	Li-ne	Column	Instance name	Reachability Coun-ter	Reachability Percen-tage	Reachability List of Ticks
module	1	1	COUNTER	1	11	1
UD	2	7	COUNTER	2	22	4, 7
S	3	7	COUNTER	8	88	2, 3, 4, 5, 6, 7, 8 9
Count	4	8	COUNTER	8	88	2, 3, 4, 5, 6, 7, 8 9
every	5	2	COUNTER	9	100	1, 2, 3, 4, 5, 6, 7 8, 9
do	5	10	COUNTER	8	88	2, 3, 4, 5, 6, 7, 8 9
if	6	3	COUNTER	8	88	2, 3, 4, 5, 6, 7, 8 9
then	6	17	COUNTER	5	55	2, 3, 7, 8, 9
emit	7	5	COUNTER	5	55	2, 3, 7, 8, 9
Count((pre(?Count)	7	10	COUNTER	5	55	2, 3, 7, 8, 9
else	8	3	COUNTER	3	33	4, 5, 6
if	9	4	COUNTER	3	33	4, 5, 6
then	9	23	COUNTER	1	11	6
emit	10	5	COUNTER	1	11	6
Count(pre(?Count)	10	10	COUNTER	1	11	6
else	11	4	COUNTER	2	22	4, 5
emit	12	5	COUNTER	2	22	4, 5
Count(pre(?Count)	12	10	COUNTER	2	22	4, 5

Reached SSM objects:

SSM object name	Instance name	Reachability Counter	Reachability Percentage	Reachability List of Ticks
COUNTER	COUNTER	8	88	2, 3, 4, 5, 6, 7, 8 9

SSM object net explanations:

SSM object name	Net Name	Net Explanation
state COUNTER	Zero00_0_0	Auxiliary gate, always false.
state COUNTER	RootRst_0_0	Modular compiling: resets a submodule.
state COUNTER	RootWrst_0_0	Modular compiling: weakly resets a submodule.
state COUNTER	Res_0_0	Triggers resumption of a selection statement.
state COUNTER	Se-lex_Selex_0_0	Or gate that bears selection incompatibility information used for verification and optimi-zation.
state COUNTER	Boot_0_0	Boot register, yields 1 and then 0 forever.
state COUNTER	K1_0_0	Tells whether the program is still active.
state COUNTER	GoAct_0_0	Triggers reset actions for local signals values and booleans.
state COUNTER	ResAct_0_0	Triggers calls to data resumption actions (pre value setting).
state COUNTER	PauseReg_15_0	Pause register, value 1 when the pause statement is active.

5.2 COUNTER Coverage Report

Completion Scenario File:
D:\Esterel\UpDownCounter\Default\Simulation\Scenarios\simul_scenario_3.esi

Files are CHRONOLOGICALLY sorted

Scenario	Scenario coverage		Cumulated coverage	
	States	Transitions	States	Transitions
simul_scenario	2	2	2	2
simul_scenario_1	2	2	2	2

Generated on 01/04/2006 15:38:15 by Esterel Studio 5.2.1e (build i10)

6 Appendix B

6.1 Counter Source Code

```
module COUNTER:
input UD : bool init true;
input S ;
output Count : unsigned<2**8> init 0;
 every S do
  if (?UD=true) then
    emit Count((pre(?Count) + 1) mod 256)
  else
   if (pre(?Count)=0) then
    emit Count(pre(?Count) + 255)
   else
    emit Count(pre(?Count) - 1)
   end if
  end if
 end every
end module
```

6.2 Counter Source Code including Assertion

```
module COUNTER:
input UD : bool init true;
input S ;
```

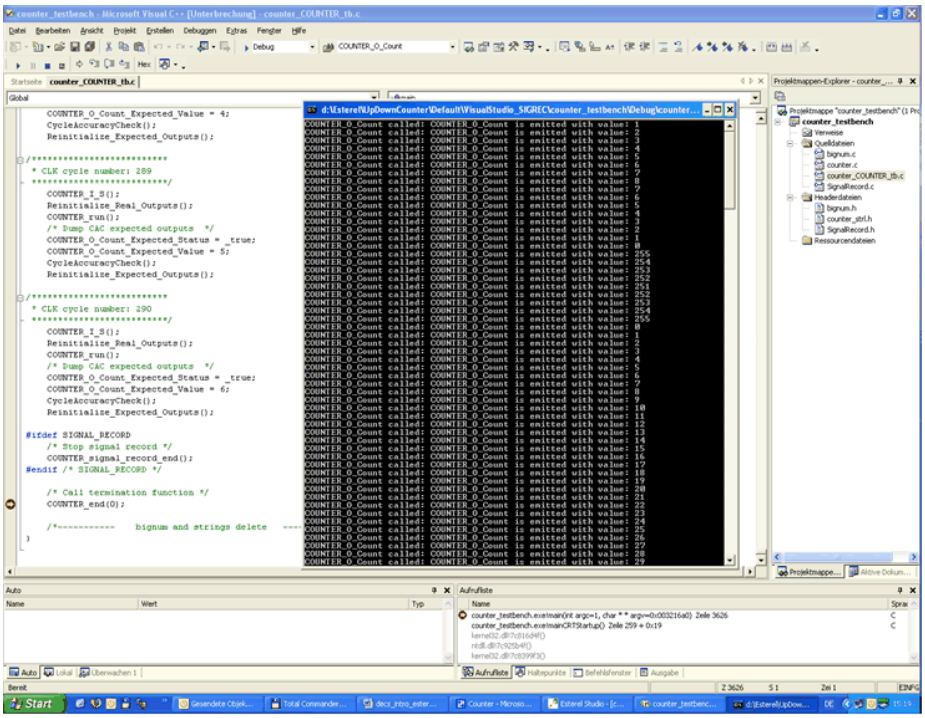
```
output Count : unsigned<2**8> init 0;
 every S do
  if (?UD=true) then
    emit Count((pre(?Count) + 1) mod 256)
  else
   if (pre(?Count)=0) then
    emit Count(pre(?Count) + 255)
   else
    emit Count(pre(?Count) - 1)
   end if
  end if
 end every
||
   // assertion section
   sustain  {
      assert CountInBound = (?Count >= 0 and ?Count <= 255)
   }
end module
```

6.3 Observer Source Code (counter_observer.strl)

```
module COUNTERCONSTRAINT:
input Count : unsigned<2**8> init 0;
input S;
output OverUnderFlow;
  every S do
   if (?Count<0) then
    emit OverUnderFlow
   end if
  end every
end module
```

7 Appendix C

7.1 Visual Studio Test Bench Output



7.2 Keil Vision 2 Test Bench Output