DOCUMENT

document title/ titre du document

Single Event Upsets Simulation Tool Functional Description

prepared by/*préparé par*	Daniel González Gutiérrez
reference/*réference*	TEC-EDM/DGG-SST2
issue/*édition*	1
revision/*révision*	2
date of issue/*date d'édition*	26/07/04
status/*état*	Draft
Document type/*type de document*	Functional Description
Distribution/*distribution*	

European Space Agency
Agence spatiale européenne

APPROVAL

Title *titre*	issue 1 *issue*	revision 2 *revision*

author *auteur*	date 26/07/04 *date*

approved by *approuvé by*	date *date*

CHANGE LOG

reason for change /*raison du changement*	issue/*issue*	revision/*revision*	date/*date*
This file is based on the former User manual of revision 1.1 renamed to Functional Description. Changes from 1.1 to 1.2 are added here and a new User manual is issued.	1	2	26/07/04

CHANGE RECORD

Issue: 1 Revision: 2

reason for change/*raison du changement*	page(s)/*page(s)*	paragraph(s)/*paragraph(s)*

TABLE OF CONTENTS

1 SCOPE 1

2 TERMS AND ACRONYMS 1

3 APPLICABLE DOCUMENTS 1

4 INTRODUCTION 1

5 FUNCTIONAL OVERVIEW 1
5.1 Running steps 2
5.2 Internal Structure 3
5.2.1 Interdependency table 3
5.2.2 Block diagram 4
5.3 SST scripts description 5
5.3.1 SST_gui.tcl 5
5.3.1.1 'Actions' menu button 5
5.3.1.1.1 Gather design information 6
5.3.1.1.2 Generate upsets 6
5.3.1.1.3 Run a simulation introducing the upsets 7
5.3.1.2 'Configuration' menu button 7
5.3.1.3 'Help' menu button 7
5.3.2 SST_startup.tcl 7
5.3.3 SST_list_instances.pl 8
5.3.3.1 all_instances.dat format 9
5.3.4 SST_wires_parser.pl 9
5.3.4.1 Wire files format 10
5.3.5 SST_upset_generator.pl 10
5.3.5.1 Options and switches 11
5.3.5.1.1 Manual option (-m) 11
5.3.5.1.2 Instances option (-i) 12
5.3.5.1.3 Number option (-n) 13
5.3.5.1.4 Time option (-t) 14
5.3.5.1.5 Help option (-h) 15
5.3.6 SST.do 15
5.3.7 SST_bit_flip.pl 16
5.3.8 SST_config.tcl 16
5.3.9 SST_perl_package.pm 16

6 SST DIRECTORY STRUCTURE 16
6.1 Files supplied by the user 16
6.2 Files generated by the tool 16

7 TEST EXAMPLE 17

1 SCOPE

The object of this document is to describe what the Single Event Upsets Simulation Tool is able to do, and how this is achieved.

2 TERMS AND ACRONYMS

DUT	Design Under Test
GUI	Graphical User Interface
HDL	Hardware Description Language
SEU	Single Event Upset
SST	Single Event Upsets Simulation Tool

3 APPLICABLE DOCUMENTS

The table 3.1 shows the documents referred in this document.

[ModelCRef]	Modelsim Command Reference.
[SSTUsrMan]	SST User Manual

Table 3.1 Applicable documents

4 INTRODUCTION

Single Event Effects and in particular Single Event Upsets are of major concern when dealing with electronic designs that will suffer the consequences of a radiation environment. The sooner we know the effects of SEUs on a particular design, the better. This is the main reason for the development of this tool: to be able to emulate SEUs, easily and in a useful controlled manner, while still in the simulation (HDL) stages of the IC design flow.

One of the main concerns about introducing SEUs in HDL simulations, is to be able to do it with independence of the particular design and in a non-intrusive way. The use of *Perl* as the main programming language of the tool, and *Tcl/tk* to interact with the simulator, made this possible. On the other hand, the fact that the bit-flips were to be done while a design was being simulated, created a dependency with the simulation tool (*Modelsim*).

5 FUNCTIONAL OVERVIEW

The SEUs Simulation Tool consists of a set of *Perl* and *Tcl* scripts used to prepare the environment to be able to upset (bit flip) in a controlled and effective manner, any register or internal signal of the design under test, while a simulation is running. The tool allows several degrees of freedom for the user to decide, which signals and when during the simulation the SEUs will be emulated. A test bench for the DUT is needed in order to detect if the upsets modify in any way the behavior of the design.

In order to provide the user with an easy to use environment, a Graphical User Interface has been developed in *Tcl/tk* to invoke the scripts (see figure 5.1). The user will always be able to call the *Tcl* and *Perl* scripts directly from the simulator or from a command line interface without using the GUI, if running the tool in command line mode is preferred.

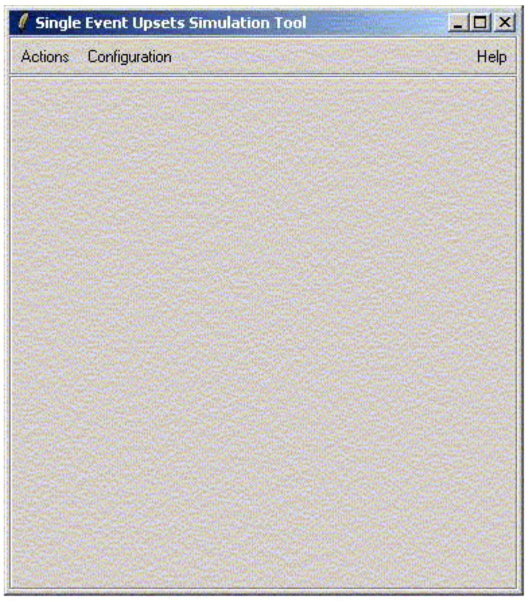


Figure 5.1 SST Graphical User Interface main window

5.1 Running steps

Either if the GUI is being used or the command line mode has been chosen, the use of the tool requires the following four sequential steps (see [SSTUsrMan] for a detailed description of them):

- Load the Test Bench of the DUT in the simulator.
- Gather information about the design.
- Select the wires that are going to be upset and when.
- Run a simulation introducing the upsets.

5.2 Internal Structure

5.2.1 INTERDEPENDENCY TABLE

A table describing the dependencies of the SST scripts can be found hereafter:

	Is called		**Calls**	**Input files/*parameters***	**Output files**
	GUI mode	**command line mode**			
SST_gui.tcl	From a *Modelsim* window	-	SST_startup.tcl SST_config.tcl SST_upset_generator.pl sst.do	*Configuration parameters have to be set up in:* SST_config.tcl	SST_perl_package.pm hierarchy.dat all_instances.dat Wire files all_wires_parser.log sst.do
SST_startup.tcl	Actions/ Gather design information	From a *Modelsim* window.	SST_config.tcl SST_list_instances.pl SST_wires_parser.pl	No input files. *Configuration parameters have to be set up in:* SST_config.tcl	hierarchy.dat *.sig files SST_perl_package.pm
SST_config.tcl	Automatically called by SST_startup.tcl		-	*Configuration parameters have to be set up in this script*	SST_perl_package.pm
SST_list_instances .pl	Automatically called by SST_startup.tcl		SST_perl_package.pm	hierarchy.dat	all_instances.dat
SST_wires_parser .pl	Automatically called by SST_startup.tcl		SST_perl_package.pm	all_instances.dat *.sig files	all_instances.dat Wire files all_wires_parser.log
SST_upset_genera tor.pl	Actions/ Generate Upsets	From a command line interface or a *Modelsim* window.	SST_perl_package.pm	all_instances.dat Wire files *Command line/GUI options and switches*	sst.do
sst.do	Actions/ Run a simulation introducing the upsets	From a *Modelsim* window	SST_bit_flip.pl	*It requires a test bench for the DUT.*	-
SST_bit_flip.pl	Automatically called by sst.do		-	*The value of the signal to be upset*	-
SST_perl_package .pm	Automatically called by: SST_list_instances.pl SST_wires_parser.pl SST_upset_generator.pl		-	-	-

Table 5.1 Scripts interdependency

5.2.2 BLOCK DIAGRAM

A block diagram of the scripts involved in the tool can be found in the following figures:

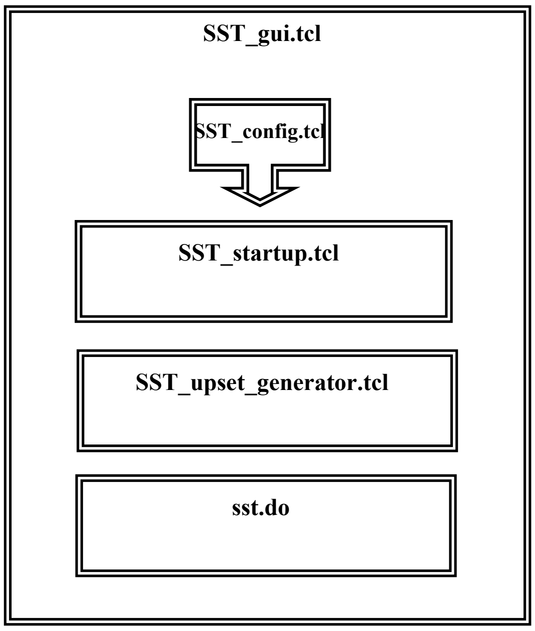


Figure 5.2 SST_gui.tcl block diagram

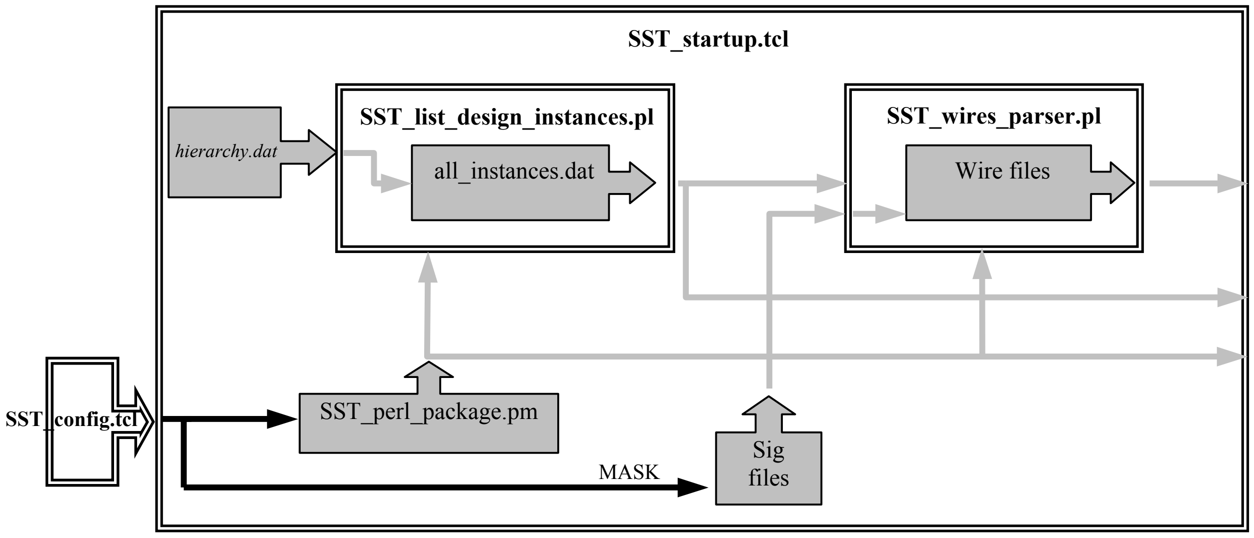


Figure 5.3 SST_startup.tcl block diagram

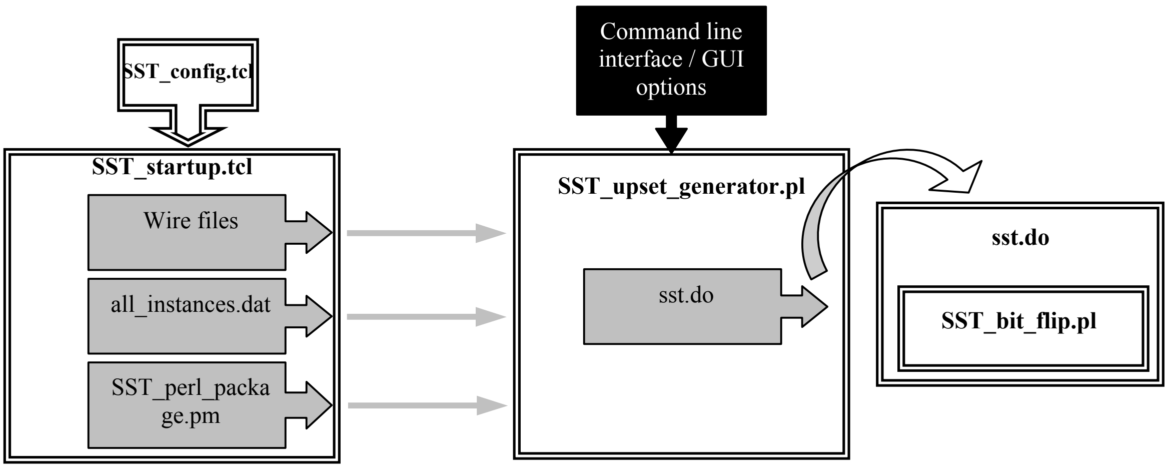


Figure 5.4 Interactions between SST_startup.tcl, SST_upset_generator.pl and sst.do.

5.3 SST scripts description

5.3.1 SST_GUI.TCL

The Graphical User Interface is written in *Tcl/tk* and consists of a menu bar and an interface window. The steps required to run the tool can be easily followed by simply clicking in the menu buttons and by filling the entry widgets that will appear in the interface window.

5.3.1.1 'Actions' menu button

The actions menu button has three different options as can be seen in figure 5.5. Each of these options invokes a different *Tcl* or *Perl* script, which performs the action stated by its option label.

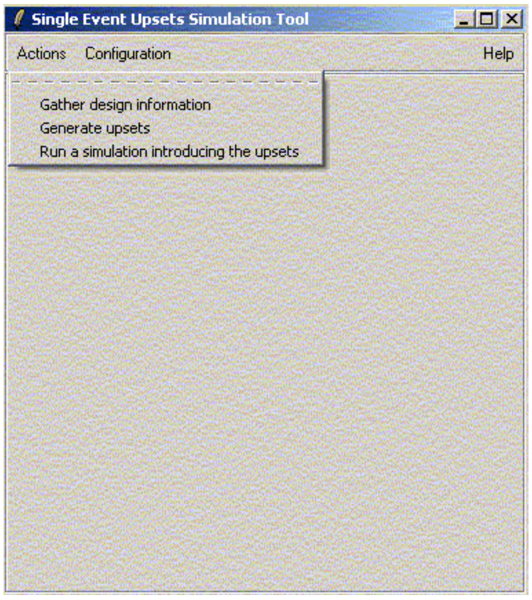

Single Event Upsets Simulation Tool
Actions
Configuration
Help
Gather design information
Generate upsets
Run a simulation introducing the upsets

Figure 5.5 SST GUI 'Actions' menu button.

5.3.1.1.1 Gather design information

This option executes the script *SST_startup.tcl* after reminding the user that a design needs to be loaded.

5.3.1.1.2 Generate upsets

This option executes the script *SST_upset_generator.pl* after the user has filled the entry boxes and check buttons of the interface window (see figure 5.6).

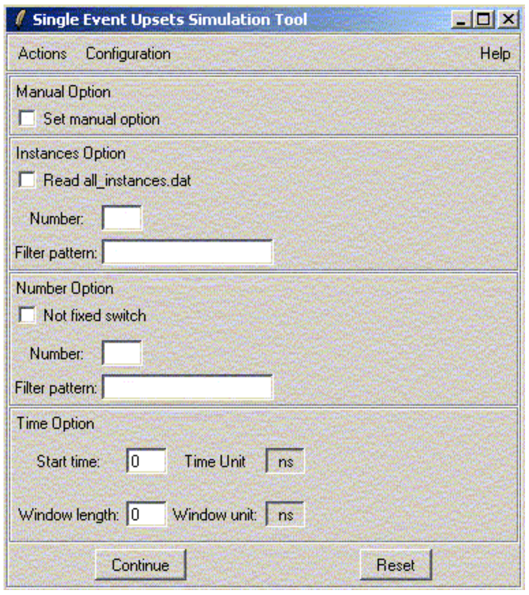


Figure 5.6 SST GUI, 'Generate upsets'.

5.3.1.1.3 Run a simulation introducing the upsets

This option executes the script *sst.do* (generated by *SST_upset_generator.pl*), which runs a simulation introducing the desired upsets.

5.3.1.2 'Configuration' menu button

The configuration menu button opens the file *SST_config.tcl* for editing.

5.3.1.3 'Help' menu button

The help menu button contains links to PDF versions of both this document and [SSTUsrMan].

5.3.2 SST_STARTUP.TCL

This script is written in *Tcl* and consists of a set of simulator commands and two *Perl* script calls. It can be seen as a simulator macro as a whole, which is used to set up the environment for another script: *SST_upset_generator.pl*

In order to upset the value of any wire of a particular design, it is necessary to know the exact location of it, which implies that the hierarchy and the structure of the DUT has been completely analysed and understood. On one hand we need to know the number of modules and the way they are instantiated. On the other hand, we need to specify the type of wire we are interested in (internal signal, input or output of any sub-module, etc) and find out how many wires of this type can be found in every module.

The hierarchy of the DUT (the number of modules and how they are instantiated) is saved into the file *hierarchy.dat* by *SST_startup.tcl*, using the commands reproduced in figure 5.7.

```
view structure
.structure.tree expandall -1
.structure.tree write ./SST/control_files/hierarchy.dat
```

Figure 5.7 Saving the hierarchy of the DUT

However, if that structure file is to be used by the tool, we need to give it a more useful format. This is the task performed by the *Perl* script *SST_list_instances.pl*, called from *SST_startup.tcl*

The selection of the type of wire we want to focus in, has to be done by directly editing the *wire_mask* configuration parameter in *SST_config.tcl*.

The names of the wires that belong to a particular instance can be saved into a file, which name will be the instance name (unless several instances have the same name, in that case a number is appended at the end) and which extension will be .sig, using a command similar to the one used to save the hierarchy of the DUT, as can be seen in figure 5.8.

```
view signals
.signals.tree write instancename.sig
```

Figure 5.8 Saving the wires that belong to a particular module of the DUT

To collect this information from all the modules instantiated in the DUT, a loop through the whole hierarchy has to be performed. As it happened with *hierarchy.dat*, the *.sig files will need to be reformatted by another script (*SST_wires_parser.pl*) in order to be used by the SST afterwards. These newly formatted files will be stored in: ./SST/wire_files and will have the extension hold in the *SST_config.tcl* variable *'$wire_files_ext'*

In order for the *Perl* files to use the configuration parameters defined in *SST_config.tcl*, the script *SST_startup.tcl* creates a *Perl* package called *SST_perl_package.pm* in SST/control files. This package contains a copy of those parameters that will be used by *Perl* scripts.

5.3.3 SST_LIST_INSTANCES.PL

This *Perl* script is called from *SST_startup.tcl*. Its basic function is to reformat the file *hierarchy.dat* into *all_instances.dat*. It also checks the names of all the instances, so the files that will be generated by the script *SST_wires_parser.pl* (which are named after every single instance), could have names that do not conflict with each other or with the tool itself.

Samples of *hierarchy.dat* and *all_instances.dat* can be found in figures 5.9 and 5.10.

```
tb_top: tb_top(behaviour)
        tbinterfacei: interface(beh)
                txfifo: fifo(beh)
                rxfifo: fifo(beh)
                txi: tx(beh)
                rxi: rx(beh)
                statemachinei: statemachine(beh)
        link: link_wrap(rtl)
                li_1: link_interface(structural)
                        clk10gen_1: clk10gen(rtl)
                        state_machine_1: state_machine(rtl)
                                state_counter_1:
state_counter(rtl)
                        receiver_1: receiver(structural)
                                rxcontrol: rx_control(rtl)
                        transmitter_1: transmitter(structural)
                                txctrl: tx_controller(rtl)
                txfifo: generic_fifo(behav)
                rxfifo: generic_fifo(behav)
        common_ctrl1i: common_ctrl(behaviour)
Package std_logic_textio
```

Figure 5.9 hierarchy.*dat* file

5.3.3.1 all_instances.dat format

The file contains 4 columns:

- **Force**: this column is used to state whether the instance has been selected or not to have its wires upset ('No' is the default value written in this column when the file is created).
- **File name**: The file name is based on the instance name. If several instances have the same name, an '-n#' will be appended at the end of the file name, from the second occurrence on, where # holds the number of repetitions. If an instance name has any of the following characters: \/:*?"<>|, they will be replaced by '-' when giving a name for its corresponding file (the instance full path column will keep the original characters).
- **# of wires**: the number of wires of the type selected in *SST_startup.tcl* that can be found in each instance. This field is filled by the script *SST_wires_parser.pl* since *SST_list_instances.pl* has no information about the contents of each instance ('unknown' is the default value written in this column when the file is created).
- **Instance full path.**

```
Force          File Name      # wires        Instance full path
No             tb_top          unknown       /tb_top
No             tbinterfacei    unknown       /tb_top/tbinterfacei
No             txfifo          unknown       /tb_top/tbinterfacei/txfifo
No             rxfifo          unknown       /tb_top/tbinterfacei/rxfifo
No             txi             unknown       /tb_top/tbinterfacei/txi
No             rxi             unknown       /tb_top/tbinterfacei/rxi
No             statemachinei   unknown       /tb_top/tbinterfacei/statemachinei
No             link            unknown       /tb_top/link
No             li_1            unknown       /tb_top/link/li_1
No             clk10gen_1      unknown       /tb_top/link/li_1/clk10gen_1
No             state_machine_1 unknown       /tb_top/link/li_1/state_machine_1
No             state_counter_1 unknown       /tb_top/link/li_1/state_machine_1/state_counter_1
No             receiver_1      unknown       /tb_top/link/li_1/receiver_1
No             rxcontrol       unknown       /tb_top/link/li_1/receiver_1/rxcontrol
No             transmitter_1   unknown       /tb_top/link/li_1/transmitter_1
No             txctrl          unknown       /tb_top/link/li_1/transmitter_1/txctrl
No             txfifo-n1       unknown       /tb_top/link/txfifo
No             rxfifo-n1       unknown       /tb_top/link/rxfifo
No             common_ctrlli   unknown       /tb_top/common_ctrlli
```

Figure 5.10 *all_instances.dat*
Reformatted version of *hierarchy.dat* generated by the script SST_list_instances.pl

5.3.4 SST_WIRES_PARSER.PL

This *Perl* script is also called from *SST_startup.tcl*. It has two separate tasks, the first one is to give a useful list of the wires that can be found in every instance, and the second one is to update the contents of the file *all_instances.dat*, by setting the number of wires every instance has, and by removing from it all those instances that have no wires from the type selected in *SST_config.tcl* (these removed instances will be logged together with a warning message to *all_wires_parser.log*)

As it was stated before, for every single instance found in the DUT, the script *SST_startup.tcl* generates a .sig file. These *.sig files contain all the relevant information of the wires that can be found in each instance, however their format is not very user friendly. The *SST_wires_parser.pl* script gives the *.sig files a handier format as can be seen in figure 5.11 B. The extension name for these reformatted files can be set up in *SST_config.tcl* as desired. They will inherit the file name from the .sig file they come from (which is set in *SST_list_instances.pl*), and they will be referred as wire files throughout this document.

```
clk = U
rst = U
rx_data = XXXXXXXXX
       (8) = U
       (7) = U
       (6) = U
       (5) = U
       (4) = U
       (3) = U
       (2) = U
       (1) = U
       (0) = U
rx_write = U
```
A

```
Force   Name           @Time

No      clk            0ns
No      rst            0ns
No      rx_data        0ns
No      rx_write       0ns
```
B

Figure 5.11 Internal signals of a particular instance.
A. *.sig file
B. Wire file

5.3.4.1 Wire files format

Each file contains 3 columns:
- Force: this column is used to state whether the wire has been selected or not to be upset ('No' is the default value written in this column when the file is created).
- Name.
- @Time: The scheduled time when the wire is going to be upset ('0ns' is the default value).

5.3.5 SST_UPSET_GENERATOR.PL

This *Perl* script is the core of the SEUs Simulation Tool since the selection of what wires are to be upset and when this will happen is done here. The basic idea taken into account while developing this script was to give the user a great flexibility when controlling where and when the upsets can be simulated, so a wide range of tests could be run on the DUT.

Before running *SST_upset_generator.pl*, it is mandatory to have executed the macro *SST_startup.tcl* once (or its equivalent GUI option). The files created by the calls to *Perl* scripts found in this macro are necessary for *SST_upset_generator.pl* to run properly.

The following points have to be considered:
- Number of wires: How many do we want to upset?

- Location of the wires: Is there a particular place (instance, group of wires, etc) on our design that we want to test?
- Time when the upsets have to be done: At what point of the simulation do we want to upset the wires?
- Names of the wires: Is there a particular pattern of characters that we can use to select the wires to be upset?

The selection of wires and the time of the upset is done via some command line interface options and switches, if the command line mode is used, or by filling the entry boxes and check buttons of the interface window, if we are running the GUI. Once these input parameters are interpreted, the script *SST_upset_generator.pl* generates a simulator macro, *sst.do*, which will be used to run a simulation and upset the selected wires at the times specified.

5.3.5.1 Options and switches

5.3.5.1.1 Manual option (-m)

Both *all_instances.dat* and the wire files have to be edited manually by the user of the script, in order to select the desired wires and the upset times. The way to select a particular instance in *all_instances.dat* is by writing a 'Yes' in the *Force* column. The way to select a wire of an instance is by writing a 'Yes' in the *Force* column and a time value on the *@time* column of the corresponding wire file.

This option excludes the rest of the command line interface options.

A

```
C:\test >SST_upset_generator.pl –m
```

B

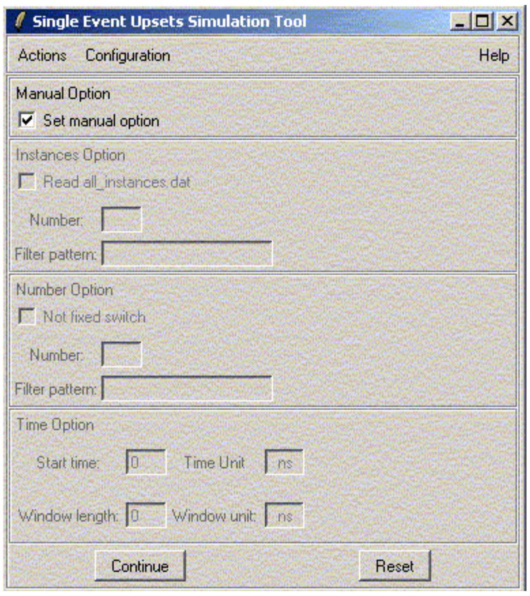

Single Event Upsets Simulation Tool
Actions Configuration Help
Manual Option
[✓] Set manual option
Instances Option
[ ] Read all_instances.dat
Number:
Filter pattern:
Number Option
[ ] Not fixed switch
Number:
Filter pattern:
Time Option
Start time: 0 Time Unit: ns
Window length: 0 Window unit: ns
Continue Reset

Figure 5.12 Example of use of the 'manual' option of the script SST_upset_generator.pl.
A. Command line mode.
B. GUI mode.

5.3.5.1.2 Instances option (-i)

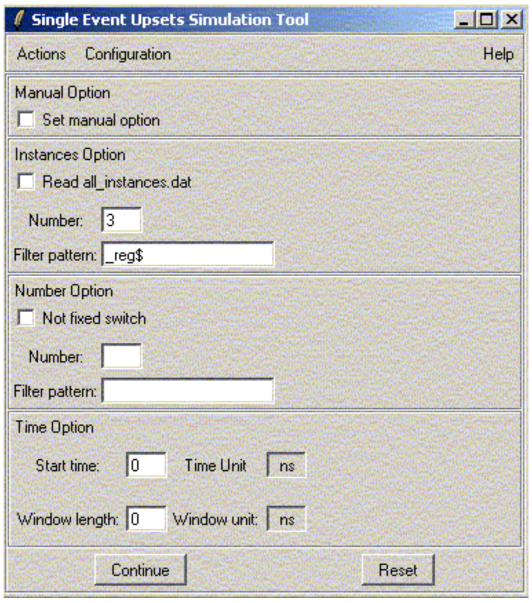
This option is used to specify in what instances, from the ones that can be found in *all_instances.dat*, we want to induce the upsets. It has 3 switches (optional characters or fields are written inside brackets):

- r(ead): the user has selected the instances by editing *all_instances.dat*. The script will just read this file in. Note that this option is not similar to "- m" since that manual option implies that both all_instances.dat and the wire files have been edited.
- f(ilter): the selection of instances will be done by filtering their names using patterns introduced via the command line interface. The patterns should be *Perl* like regular expressions.
  Valid input format for the command line mode: -i( )filter (#) pattern1 (pattern2) (pattern3)
  Where # is the number of filtered instances to be randomly selected. If this number is not specified, all the filtered instances will be selected.
- Number: a number specifies the amount of instances that will be randomly selected.

A

```
C:\test >SST_upset_generator.pl  -ifilter 3 _reg$
```

B



Manual Option
☐ Set manual option

Instances Option
☐ Read all_instances.dat
Number: 3
Filter pattern: _reg$

Number Option
☐ Not fixed switch
Number:
Filter pattern:

Time Option
Start time: 0 Time Unit ns
Window length: 0 Window unit: ns

Figure 5.13 Example of use of the 'instances' option of the script SST_upset_generator.pl

Amongst all the instances whose names end with '_reg', three of them will be randomly selected. The '$'character is used by *Perl* and *Tcl* to anchor the search to the end of a string. If we want to search for the occurrence of a '$' in any instance, '\$' should be written instead.

A. Command line mode.
B. GUI mode.

5.3.5.1.3 Number option (-n)

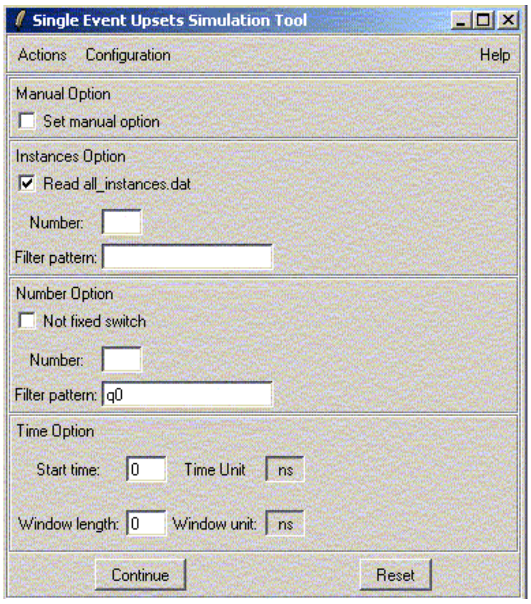
This option is used to set which wires will be upset. It has 3 switches (optional characters or fields are written inside brackets):

- n(ot_fixed): the number of wires is calculated using the parameter weight_in, which is defined in *SST_config.tcl*. This parameter represents the percentage of the wires that will be upset in each selected instance.
- f(ilter): the selection of wires will be done by filtering their names using patterns introduced via the command line interface. The patterns should be *Perl* like regular expressions.
  Valid input format for the command line mode: -i( )filter (#) pattern1 (pattern2) (pattern3)
  Where # is the number of filtered wires to be selected. If this number is not specified, all the filtered wires will be selected.
- Number: a number specifies the amount of wires that will be randomly selected.

A

```
C:\test >SST_upset_generator.pl  -ir -nfilter q0
```

B



Manual Option
☐ Set manual option

Instances Option
☑ Read all_instances.dat
Number:
Filter pattern:

Number Option
☐ Not fixed switch
Number:
Filter pattern: q0

Time Option
Start time: 0   Time Unit: ns
Window length: 0   Window unit: ns

Figure 5.14 Example of use of the 'number' option of the script SST_upset_generator.pl
All the wires (of the instances set to 'Yes' in all_instances.dat) whose names contain the pattern 'q0' will be the ones selected.

A. Command line mode.
B. GUI mode.

5.3.5.1.4 Time option (-t)

This option is used to determine in which time window (within the test bench simulation limits) the upsets will be emulated. Given a starting time value (0 by default) and a time window, the script will randomly set a time value between the limits specified, for each signal to be upset.
Valid input formats for the command line mode (parenthesis can be omitted):

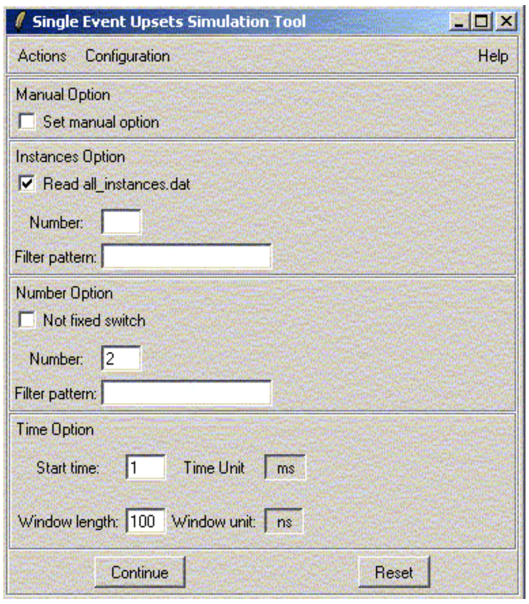
- -t window_length time_unit.
- -t (starting_value window_length) time_unit.
- -t (starting_value time_unit window_length time_unit).

Note that the time window has to be selected between the simulation limits, therefore the simulation length has to be known a priori.

A

```
C:\test >SST_upset_generator.pl -ir -n2 -t 1ms 100ns
```

B


Single Event Upsets Simulation Tool

Manual Option
☐ Set manual option

Instances Option
☑ Read all_instances.dat
Number:
Filter pattern:

Number Option
☐ Not fixed switch
Number: 2
Filter pattern:

Time Option
Start time: 1 Time Unit ms
Window length: 100 Window unit: ns

Figure 5.15 Example of use of the 'time' option of the script SST_upset_generator.pl
Two wires (of the instances set to 'Yes' in all_instances.dat) will be randomly selected to be upset, at random time values between 1ms and 1.0001ms.

A. Command line mode.
B. GUI mode.

5.3.5.1.5 Help option (-h)

This option displays a help message with information about the use of the script and some examples.

5.3.6 SST.DO

This script is written in *Tcl* and consists of a set of simulator commands and a *Perl* function call, which will automatically run a simulation introducing the upsets. The script *SST_upset_generator.pl* generates it automatically.

The way *sst.do* proceeds is as follows (see figure 5.16 for a detailed example):

- The simulation is run up to the first scheduled upset.
- The value of the wire is checked.
- If that value is not undefined, it is upset using the *Perl* script *SST_bit_flip.pl* and the *Modelsim* command *force –deposit*.
- The simulation is run up to when the next upset is scheduled.
- The value of the wire is checked and upset…
- When the last upset is done, wait until the simulation ends.

```
#Macro generated by SST_upset_generator.pl

run 262285 ns
# define signal path
set wire /tb_top/tbinterfacei/txfifo/data_out
# examine current value
set wire_checked_val [exa $wire]
# flip one bit of the wire
set wire_upset_val [exec perl –S SST_bit_flip.pl
$wire_checked_val]
if {$wire_upset_val != "undefined"} {
  echo Forcing $wire to $wire_upset_val @ 262285ns
  force -deposit $wire $wire_upset_val
} else {
  echo Unable to upset $wire. Undef value.
}

run 10 ns
# define signal path
set wire /tb_top/link/li_1/state_machine_1/cntrl
# examine current value
set wire_checked_val [exa $wire]
# flip one bit of the wire
set wire_upset_val [exec perl –S SST_bit_flip.pl
$wire_checked_val]
if {$wire_upset_val != "undefined"} {
  echo Forcing $wire to $wire_upset_val @ 262295ns
  force -deposit $wire $wire_upset_val
} else {
  echo Unable to upset $wire. Undef value.
}
run -all
```

Figure 5.16 Source code of an sst.do file

5.3.7 SST_BIT_FLIP.PL

This small *Perl* script is used to change the value of a wire and emulate the occurrence of a Single Event Upset. If the width of the wire is one bit, it negates it. On the other hand, if the width of the wire is greater than one, it changes the value of one of its bits; the selection of the bit that is going to be upset, in this case, is done in a random manner.

5.3.8 SST_CONFIG.TCL

This file contains global variable definitions for parameters used in different scripts of the tool. The parameters that have to be used by *Perl* scripts will be copied to the package *SST_perl_package.pm*

The values of some of these configuration parameters can be changed for advance users to fine tune the SEUs Simulation Tool. Other configuration parameters such as *wire_mask* and *wire_files_ext* should be checked every time the initialization phase is run, to make sure that the information gathered is the one we are interested in.

Please refer to the source code of this file to know more about each configuration parameter.

5.3.9 SST_PERL_PACKAGE.PM

This *Perl* package contains a copy of those configuration parameters set in *SST_config.tcl*, which have to be used in *Perl* scripts.

6 SST DIRECTORY STRUCTURE

6.1 *Files supplied by the user*

- **./**
  HDL testbench_files

6.2 *Files generated by the tool*

- **./SST/control_files**
  all_instances.dat, *all_wires_parser.log, hierarchy.dat,sst.do*, *SST_perl_package.pm*
- **./SST/wire_files**
  In this folder we can find all the wire files.

7 TEST EXAMPLE

Assuming that the tool is run together with a self-checking test bench for the DUT, a wide range of SEUs tests can be created. The *Tcl* macro of figure 7.1, which is described hereafter, shows an example of what can be done with the SST.

```
###########################################################################################
# Before running this script we need to:
# * Load the design and run SST_startup.tcl
# * Set the variable number_of_ff to the total number of ff we would like to upset (one
#   at a time)
###########################################################################################

1       set number_of_ff 20;
2       for {set i 0} { $i<= $number_of_ff} {incr i} {
3         # execute the script SST_upset_generator
4         # select a particular flip flop in -i
5         exec perl -S SST_upset_generator.pl -ifilter ctm/reg__$i/ff\$ -nfilter 1 q0\$ -t
          247us 15us
6         echo ### Upsetting: /ctm_testbench/ctm0/ctm/structuralgen/sequentialgen__$i/ff/f
7         # run the modelsim macro created by that script
8         do ./SST/control_files/sst.do;
9         # Check status
10        view variables;
11        set status_index [.variables.tree find Status];
12        set status [.variables.tree get2 $status_index];
13        destroy .variables;
14        set keep_running [regexp {true} $status];
15        if {$keep_running} {
16          restart -f;
17          destroy .source;
18        } else {
19          break;
20        }
21      }
```

Figure 7.1 Test example coded in *Tcl* as a simulator macro.

In this test the same simulation is run continuously, upsetting a different (and only one) flip-flop in each iteration. If the simulation fails, the loop is interrupted.

The variable *$number_of_ff* holds the number of flip-flops we would like to upset. As only one flip-flop is being upset each time the simulation is run, this variable also holds the total number of times we run the simulation (line 2).

The flip-flops are the instances of our design whose names end with 'reg__#/ff' (where # is an integer number) therefore we could use the filter switch of the –i option of the script *SST_upset_generator.pl* (line 5), to select a particular flip-flop (the loop variable '*i*' is used as part of the search pattern since it holds an integer number). The outputs of the flip-flops can be selected by using 'q0' as a filter pattern for the –n option of the *SST_upset_generator.pl* script.

Its basic operation per iteration is as follows:

- Execute *SST_upset_generator.pl* filtering the output of a particular flip-flop.

- Execute *sst.do*, which will run the simulation (line 8).
- Continue with a new iteration if the test bench passed, and stop otherwise (line 15)

Note that the character '$' (used by *Perl* to anchor the search to the end of a string) is being preceded by a backslash (line 5) to avoid naming conflicts since *Tcl* also uses the dollar sing to name its local variables *($number_of_ff*, *$i*, *$keep_running*…).