SIVA

User Manual

Malay K. Ganai

SIVA: An Integrated Environment for Efficient State Space Exploration

1 Introduction

SIVA (**SI**mulation **V**erification with **A**ugmentation), first introduced in [1], is a system that provides a robust and scalable alternative to complete verification of synchronous hardware designs. It integrates simulation-based search with symbolic algorithms for coverage-directed state space exploration. It also provides a framework for future research.

State space exploration has a variety of applications including verifying safety properties, analysis of third-party RTL code to understand its behavior, generating tests to satisfy missing coverage, and justifying the reachability of a given state for a combinational equivalence checker. Instead of a complete state space search, SIVA improves on partial state exploration using guided search techniques. The primary goal of the tool is to help designers find bugs in larger quantities and more efficiently compared with other verification methods used in isolation.

2 Overview

The general flow of SIVA is shown in Figure 1. The user inputs to SIVA include the design under verification given in the language BLIF [2] and a set of coverage goals. The user can also provide lighthouses that might help in the verification process. The general concept of lighthouses is described in [1]. As for the internal design representation, SIVA applies a two-input AND/INVERTER graph [3, 4]. The core algorithms of SIVA include a directed forward search engine that is an integration of a SAT-based symbolic simulator [5] and a cycle simulator [6]. The forward search explores the design's state space using guided techniques [7, 5, 8, 6]. It attempts to find a trajectory from an initial state to a set of target states that fail particular properties. The tool uses state *score-boarding* to prioritize the state traversal according to the potential of reaching targets. We maintain a *priority-queue* for storing promising seed states. SIVA also has a backward search engine for target enlargement and automatic lighthouse generation. These enlarged targets are prioritized using a heuristic evaluation that determines their potential to be reached. Lighthouses are prioritized according to their importance for reaching the targets [5].

3 Scope

This document decribes the user-sepcifications of the input language, coverage goals, and lighthouses. Description of each command in SIVA is provided in details. It also provides instructions to use SIVA to reach the coverage goals specified.

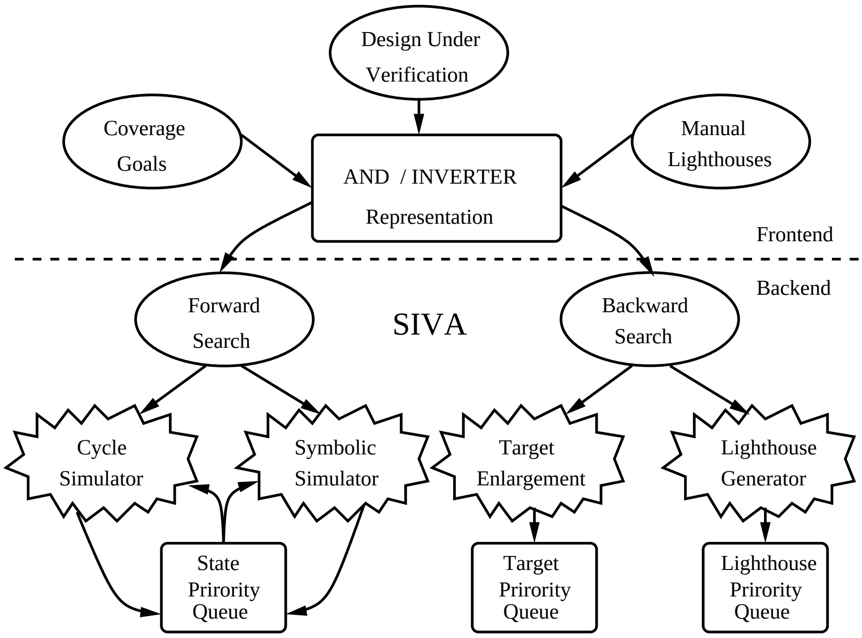

Figure 1: SIVA: An integrated environment for efficient state space exploration.

4 User Specification

4.1 Input Language – a Subset of BLIF

Input language to the tool `SIVA`, a subset of the language BLIF [2], is used to represent a synchronous hardware design with a single initial state.

A subset-BLIF file contains a flattened network of combinational and latch nodes. A latch node has a single initial Boolean value. A combinational node has a two-level, single-output logic function associated with it. Each feedback loop must contain at least one latch. The network is described as follows:

```
.model <decl-model-name>
.inputs <decl-input-list>
.outputs <decl-output-list>
<latch>
<logic-gate>
.end
```

decl-model-name is a string giving the name of the design.

decl-input-list is a white-space separated list of strings giving the formal input terminals for the model being declared. Multiple `.inputs` lines are allowed. This declaration can appear anywhere in the file between `.model` and `.end` lines.

decl-output-list is a white-space separated list of strings giving the formal output terminals for the model being declared. Multiple `.outputs` lines are allowed. This declaration can appear anywhere in the file between `.model` and `.end` lines.

latch is used to define a single bit of state element. It is declared as follows:

```
.latch <input> <output> <init-value>
```

input is the data input to the latch.

output is the output of the latch.

init-value is the initial state of the latch which can be 0 or 1.

This declaration can appear anywhere in the file between `.model` and `.end` lines.

logic-gate defines and declares a combinational gate. It is declared as follows:

```
.names <in-1> ··· <in-n> <out>
<out-cover>
```

out is a unique string for declaring the gate being defined.

in-1, ···, in-n are the naming strings of the inputs to the logic gate being defined.

out-cover is n-input and 1-output PLA description of the logic function of the gate. $\{0, 1, -\}$ is used in the n-wide "input plane" and $\{0, 1\}$ is used in the 1-bit wide "output plane". The ON-set is the specified with 1's, and the OFF-set is specified with 0's in the output plane.

As an example, logic-gate *out* $(= a + \bar{a}c + \bar{b}\bar{c})$ can be defined as:

```
.names a b c out
1-- 1
0-1 1
-00 1
```

A gate f with constant 1 function (similarly a constant 0 function) is defined as:

```
.names f
1
```

This declaration can appear anywhere in the file between `.model` and `.end` lines.

4.2 Coverage Goals and Lighthouses

The coverage goals are network nodes targetted for 0 or 1. Lighthouses are the hints that will help in reaching the coverage goals [1, 6]. Coverage goals and lighthouses are declared as follows:

```
<target>
<lighthouse>
```

target defines and declares the coverage goal. The declaration is given as:

```
<target-name> <target-value>
```

target-name is a string that represents a network node.

target-value is value 0 or 1.

lighthouse defines and declares lighthouses associated with the coverage goal. The declaration is given as:

```
<number>
<lh-1> <value-1>
··· ···
<lh-n> <value-n>
```

number specifes the number of lighthouses.

lh-1, $\cdots$, *lh-n* are strings that represent corresponding network nodes. Number of such string should match *number*.

value-1,$\cdots$,*value-n* are values that equal 0 or 1.

As an example, network node *out* targetted for 1 without any lighthouse can be specified as:

```
out 1
0
```

and with lighthouse $a = 1$ as:

```
out 1
1
a 1
```

The coverage goals are specified in a separate file. Multiple declaration of targets and lighthouses are allowed.

4.3 Debug Nodes

The debug nodes are the network nodes that used to generate debug trace. Debug trace constitutes the binary valuation of the debug nodes from the initial state. The debug node is specifes as:

```
<debug>
```

debug is a string that represent the network node.

Multiple declaration of debug nodes is allowed. It is specified in a separate file.

4.4 Debug Trace

Debug trace is output in the following format:

```
invar( (cntr=<clock>) ->
       (<debug-1> = <value-1>) &
       ... ...
       (<debug-n> = <value-n>));
```

clock is a number denoting the clock step.

debug-1···debug-n are strings denoting the name of the debug nodes.

value-1···value-n is 0/1 value on the corresponding debug nodes.

5 Command Overview

We will discuss the following list of commands available in `SIVA`.

`read_blif`	`write_blif`	`pre_process`
`solve_init`	`build_pi_free_target`	`solve`
symbolic_sim	`write_unresolved_target`	`solve_quit`

5.1 Command "read_blif"

The usage of the command is as follows:

```
read_blif <blif-file>
```

The command `read_blif` is used to read the network described in the subset-BLIF language from the *blif-file*. This command sets all the nodes in the design as potential targets.

Reinvokation of the command discard the previous execution.

5.2 Command "write_blif"

The usage of the command is as follows:

```
write_blif [-o <output>]
```

The command *write_blif* is used to write the internal AND/INVERTER graph representation in BLIF in the specified *output* file. This command should be invoked only after successful read of the design using *read_blif*.

Multiple invokation of the command is allowed.

5.3 Command "pre_process"

The usage of the command is as follows:

```
pre_process[-n <rand-depth>] [-r <num-restart>] [-t <target>]
           [-p <unresolved>]
```

The command `pre_process` is used to reach targets specified in *target* file. If the file does not exist or is not specified, previous unreached targets are used. During the execution phase, cycle-simulation of the design takes place until the specified *rand-depth*. During the simulation run, targets are checked for reachability. This process is repeated *num-restart* times from the initial state. The targets not reached, are written out in the *unresolved* file.

This command should be invoked only after successfully read of the design using *read_blif*. Mulitple invokation of the command is allowed. This command is mainly used to reach easy targets in the design.

5.4 Command "solve_init"

The usage of the command is as follows:

```
solve_init[-t <target>] [-c <group>]
          [-A <ALOG | LOG | NORMAL>] [-S] [-T <threshold>]
          [-n <rand-depth>] [-r <restarts>] [-D <debug>]
```

The command *solve_init* performs necessary utilization for solving the targets specified in *target* file. If the file is not specified or does not exist , then previous unreached targets are used. The debug nodes are specified in the *debug* file.

The options *-c, -A, -S,* and *-T* are used to control the different state prioritization schemes; the details are available in [7]. Using the option *-c*, user specifies maximum number of active *group* he would like to have for activity-induced prioritization [7]. Different normalization schemes for controlling such distribution is specified by using the option *-A*. User can specify the option *-S* to specify partition-induced state prioritization scheme [7]. The partition *threshold* specifies the support overlap threshold for deciding the partition.

Using the option *-n* and *-r* user specifies the simulation depth and number of restarts; simulation of the desing is used to determine the toggle activity of the latches which is used for forming active groups.

After successful invokation of this command, the tool goes into *solve_mode*. In *solve_mode*, the valid commands are: *build_pi_free_target*, *symbolic_sim*, *solve*, *write_unresolved_target*, and *solve_quit*.

5.5 Command "build_pi_free_target"

The usage of the command is as follows:

```
build_pi_free_target [-u <unfold-depth>]
```

This command is used to enlarge targets by *unfold-depth* and quantifying out the input support variables [6]. Command might fail if *unfold_depth* is too large.

Reinvokation of the command is allowed only if it did not succeed earlier. This command can be invoked only in *solve_mode*.

5.6 Command "solve"

The usage of the command is as follows:

```
solve[-n <rand-depth>] [-u <fwd-depth>] [-E <bwd-depth>]
      [-L] [-c <group>] [-B]
```

This command executes forward and backward search engine to reach the targets. From a seed state, cycle-simulation is invoked until *rand-depth* and symbolic simulation is invoked until *fwd-depth*. The targets are enlarged upto *bwd-depth*. The option *-L* is used for generating automatic lighthouses [6, 5]. The *group* number is used to control state pruning. Larger *group* number indicates less state pruning. The option *-B* enables storing of all new states visited during cycle-simulation.

Multiple invokation of this command is allowed. This command can be invoked only in *solve_mode*.

5.7 Command "symbolic_sim"

The usage of the command is as follows:

```
symbolic_sim [-s <start-depth>] [-u <fwd-depth>]
```

The command executes symbolic simulation as described in [5]. From the initial state, the symbolic simulation runs until *fwd-depth*. The reachability for the targets are checked between the *start-depth* and *fwd-depth*.

Multiple invokation of the command is allowed. This command can be invoked only in *solve_mode*.

5.8 Command "write_unresolved_target"

The usage of the command is as follows:

```
write_unresolved_target [-p <file>]
```

The command is used to write the unreached targets in the *file* specified.

Multiple invokation is allowed. This command can be invoked only in *solve_mode*.

5.9 Command "solve_quit"

The usage of the command is as follows:

```
solve_quit
```

This command is used to quit from the *solve mode*. It releases all the resources used in the *solve_mode*.

Multiple invokation is not allowed. This command can be invoked only in *solve_mode*.

6 Starting SIVA

Let the design be specified in *example.blif* and the targets be specified in *tgt*. A typical flow using `SIVA` would be:

```
$ siva
siva> read_blif example.blif
siva> pre_process -t tgt -n 100 -r 5
siva> solve_init -c 10 -S -T 0.01 -n 100 -r 5
siva>> build_pi_free_target -u 1
siva>> solve -n 100 -u 10 -c 2 -L -E 3
siva>> write_unresolved_target -p hard_tgt
siva>> solve_quit
siva> quit
$
```

`SIVA` is started by typing the command `siva` at UNIX prompt. Using the command *read_blif* the design *example.blif* is read. Targets specfied in the *tgt* file, are checked for reachability using the command *pre_process*. Then, using the command *solve_init* we enter into the *solve_mode* where the necessary initialization for solving the earlier unreached targets is done. The *solve_mode* is indicated by ">>". We enlarge targets using the command *build_pi_free_target*. Using the command *solve*, we check reachability for the targets. The unreached targets are then written out in the file *hard_tgt* using the command *write_unresolved_target*. We quit this *solve_mode* using the command *solve_quit*.

For comments and suggestions, please send e-mail to `malay@ece.utexas.edu`.

References

[1] M. K. Ganai, A. Aziz, and A. Kuehlmann. Enhancing Simulation with BDDs and ATPG. In *Proc. of the Design Automation Conf.*, New Orleans, LA, June 1999.

[2] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, October 1992.

[3] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *Proc. of the Design Automation Conf.*, June 1997.

[4] M. K. Ganai and A. Kuehlmann. On-the-Fly Compression of Logical Circuits. In *Proc. Intl. Workshop on Logic Synthesis*, Dana Point, California, June 2000.

[5] M. K. Ganai and A. Aziz. Robust SAT-based Symbolic Simulation and Improved Guided State Space Search. Submitted for publication, 2001.

[6] M. K. Ganai and A. Aziz. Enhancements to Invariant Verification Using SIVA. In *Proc. Intl. Workshop on High Level Design Validation and Test*, San Diego, California, November 1999.

[7] M. K. Ganai and A. Aziz. Rarity-Based Guided State Space Search. In *Great Lakes Symposium on VLSI*, March 2001. To Appear.

[8] P. Yalagandula, V. Singhal, and A. Aziz. Automatic Lighthouse Genertation for Directed State Space Search. In *Design Automation and Testing in Europe*, Paris, France, March 2000.