V3 User Tutorial

LAST UPDATE: 2014/12/05

Cheng-Yin Wu

gro070916@yahoo.com.tw

author.v3@gmail.com

Design Verification Lab (DVLab), NTUGIEE

http://dvlab.ee.ntu.edu.tw

National Taiwan University (NTU), Taiwan

http://www.ntu.edu.tw

ACKNOWLEDGMENT

The author, Cheng-Yin Wu, wants to thank his colleagues in Design Verification Lab (DVLab) and his supervisor Professor Chung-Yang (Ric) Huang for their supports and encouragements during the construction of $\mathcal{V}3$, which turns out to be one of the most valuable contributions of his Ph.D. work. Also, he would like to thank friends and anonymous researchers from all over the world who gave precious comments and interests to $\mathcal{V}3$. Last but not least, he thanks all organizers and participants of Hardware Model Checking Competition (HWMCC) [1] for their contributions to hardware verification.

LICENSE OF THIS FRAMEWORK

Contents

Chapter 1

Read and Write Designs

1.1 Introduction

In this tutorial we demonstrate how users can read designs into $\mathcal{V}3$ and write designs out from $\mathcal{V}3$ by commands. In addition, we illustrate how to print network information or plot networks by $\mathcal{V}3$.

1.2 Prerequisites

Please download the latest $\mathcal{V}3$, and let `V3` be the $\mathcal{V}3$ directory. Make sure $\mathcal{V}3$ has been successfully installed such that an executable named `v3` is located under `V3`. In addition, check if the following files exists under `V3/design/alu`:

- ☐ `alu.v` : Verilog design for Section 1.3.
- ☐ `alu.btor` : BTOR design for Section 1.4.
- ☐ `alu.aig` : AIGER design for Section 1.5.

1.3 Read and Write Verilog Designs

In this tutorial, we adopt a very simple Verilog [15] design `alu`. Please set `V3/design/alu` as the current working directory and find the design `alu.v` under this directory. To begin with, please type `../../v3` to run `v3` program. Without otherwise specified, commands in the following context should be typed and then executed under the *V3 command-line interface*.

1. Type `read rtl alu.v` to read the Verilog design `alu.v` and create a network in $\mathcal{V}3$.

   If $\mathcal{V}3$ encounters any problem in reading the Verilog input, some error messages will be either reported in the standard error or redirected to the file `quteRTL.log`. In such cases $\mathcal{V}3$ does not successfully create the network, and probably users should solve the issues in their designs and try again. Regarding that the academic front-end parser of $\mathcal{V}3$ (literally, *QuteRTL*) is not as

powerful as those in commercial tools, we suggest users to fix potential design errors by third-party tools, for instance, *Synopsys/SpringSoft Verdi/nLint*, and *Cadence LEC*. Also, please refer to the website of *QuteRTL* and ***V3 User Manual*** to take a glance of unsupported Verilog syntax.

It is important that $\mathcal{V}3$ currently permits a design with flip-flops (i.e. registers) triggered by at most one *clock* input. The *clock* signal is absent from a $\mathcal{V}3$ network [1] topology; however, clock information is still retained in the data structure.

2. Type `write rtl v3_alu.v` or `write rtl v3_alu.v -symbol` to write out current network into Verilog file `v3_alu.v`.

   Please note that some signals will be renamed for design output; however, input/output/inout port (including *clock*) names should be reserved. Furthermore, the parameter `-Symbol` reserves as more signal names specified in the original input design as possible.

3. Type `write btor v3_alu.btor` or `write btor v3_alu.btor -symbol` to write out current network into BTOR [8] file `v3_alu.btor`.

   It is important that flip-flops specified in BTOR designs are triggered by a hidden synchronous clock signal, and thereby there is no clock input for flip-flops in BTOR netlists. Parameter `-Symbol` augments names of input/output/inout port signals to the end of lines where they are declared.

4. Use command `print ntk` to retrieve network information. For instance, `print ntk -primary` lists primary input/output/inout ports, while `print ntk -verbose` compiles gates and ports in the network into statistics. (Notice the absence of the *clock* signal in the list of primary inputs.) A sample output of the two commands are shown as follows:

```
v3> print ntk -primary
Primary Inputs = rst, a(8), inst(2)
Primary Outputs = out(16)
Primary Inouts =
v3> print ntk -verbose
=================================================
BV_MUX                                           4
BV_AND                                           1
BV_ADD                                           1
BV_SUB                                           1
BV_MULT                                          1
BV_MERGE                                         4
BV_EQUALITY                                      3
BV_SLICE                                        11
BV_CONST                                         6
-------------------------------------------------
V3_PI                                            3
V3_PO                                            1
V3_FF                                            1
```

[1] Only Verilog designs can specify *clock* signals, and there is no *clock* signal defined for other supported design formats, e.g. AIGER [3] and BTOR [8].

```
================================================
TOTAL                                          36
Floating                                        1
```

5. Run command `plot ntk` to depict network topology. Command `plot ntk -level k -png alu_level.png` plots *k* levels of the network from output ports into PNG (Portable Network Graphics) file `alu_level.png`, while command `plot ntk -depth k -ps alu_depth.ps` depicts *k* cycles of the network from output ports into PS (PostScript) file `alu_depth.ps`. A sample output of the network topology for two cycles is as shown in Figure 1.1.

1.4 Read and Write BTOR Designs

BTOR and Verilog designs are modelled as word-level networks in $\mathcal{V}3$, and thereby commands to operate on networks created from Verilog inputs are usually portable to those constructed from BTOR designs. In this tutorial, we input a BTOR design `alu.btor` that was generated from the above Verilog input `alu.v` by $\mathcal{V}3$. That is, we read the BTOR output of $\mathcal{V}3$ back. [2]

1. Type `read btor alu.btor` or `read btor alu.btor -symbol` to read BTOR design `alu.btor` and then create a network in $\mathcal{V}3$.

   Please notice that the BTOR parser written in $\mathcal{V}3$ does not perform complete rule checking over the BTOR format, i.e. we assume that inputs should conform to the format. Moreover, some operator types in [8] are not supported by $\mathcal{V}3$ currently. Please also refer to ***V3 User Guide*** for our supports to BTOR.

2. Type `write btor v3_alu.btor` or `write btor v3_alu.btor -symbol` to write the network into a BTOR file named `v3_alu.btor`.

   Similarly, the parameter `-Symbol` writes signal names (more precisely, input/output ports and latches because they are the only signals declared as either *var* or *root* in BTOR) if they are specified in the original input design. Otherwise, an auxiliary name will be assigned to a nameless signal if necessary.

3. Type `write rtl v3_alu.v` or `write rtl v3_alu.v -symbol` to dump the network into Verilog.

   The effect of the parameter `-Symbol` is as described in the last step. If the *clock* signal of a sequential network is nameless (i.e. the network is not originated from a Verilog design), $\mathcal{V}3$ creates an auxiliary name *v3_clock* for the *clock* signal. The *clock* signal should always exist in a Verilog file that describes a sequential network.

4. Use commands `print ntk` to reveal network information by text and `plot ntk` to illustrate network topology by figures. Please follow corresponding steps described in Section 1.3 to complete the tutorial in this section.

[2] Readers can also rename the created BTOR output `v3_alu.btor` in Section 1.3 to `alu.btor`.

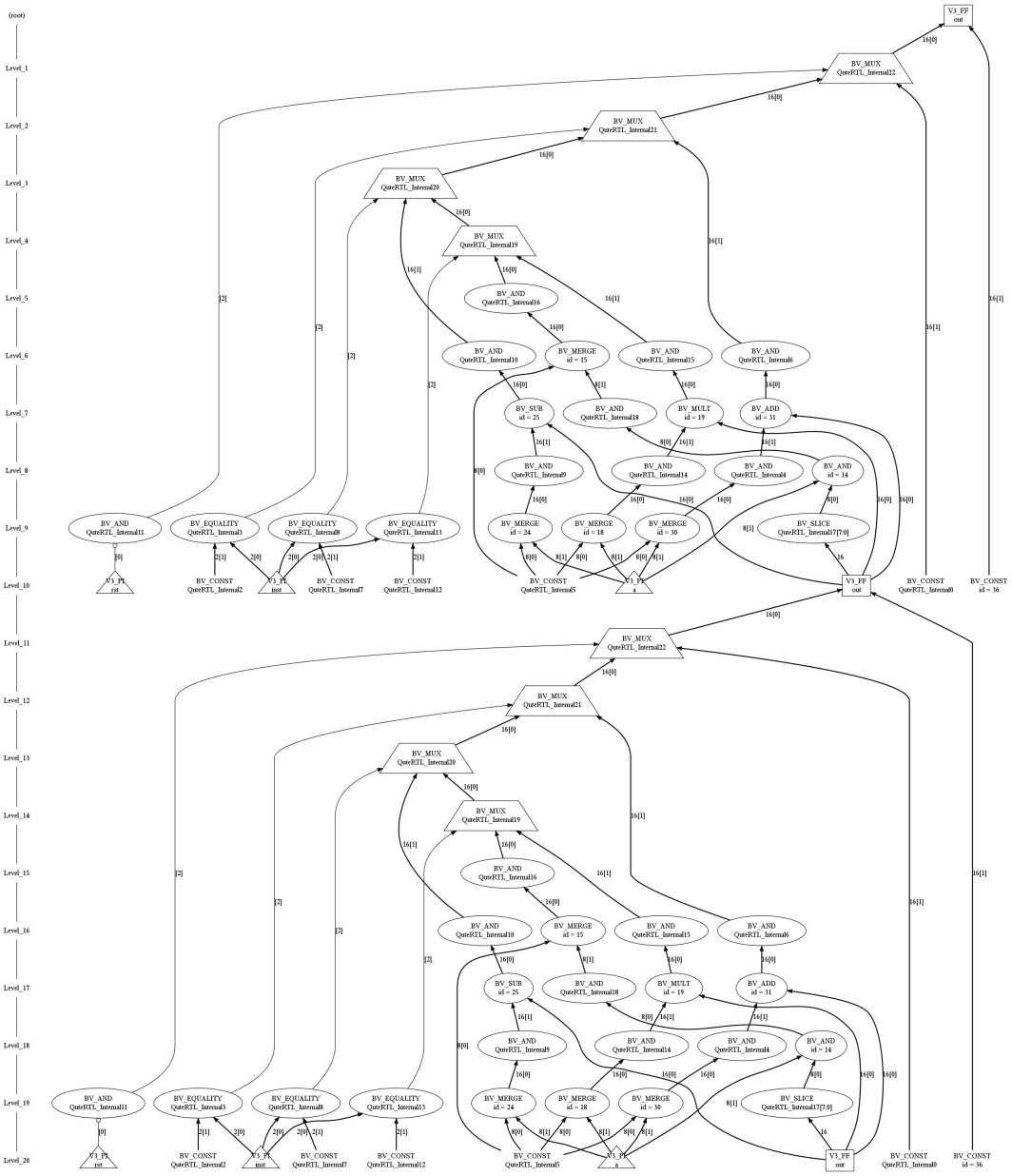

Figure 1.1: Network topology of the design `alu.v` for two cycles.

1.5 Read and Write AIGER Designs

AIGER [3] is a simple and popular format for an And-Inverter Graph (AIG), which is a compact representation of a Boolean circuit. In contrast to the modeling of Verilog and BTOR designs, $\mathcal{V}\mathcal{3}$ naturally models AIGER designs as Boolean-level networks. The input design for this tutorial is *alu.aig*, which is a Boolean-level representation of *alu* designs participated in previous sections. This command currently supports only designs in AIGER 1.0 format. However, if you are interested in reading designs in the new AIGER 1.9 [6] format, please refer to Chapter 4.

1. Type `read aig alu.aig` or `read aig alu.aig -symbol` to parse the design into $\mathcal{V}\mathcal{3}$.

   If the optional parameter `-Symbol` is specified, symbol tables in the AIGER file `alu.aig` is parsed. Then input/output ports and flip-flops are named according to the symbol table. Otherwise, they are nameless. Please notice that AIGER parser in $\mathcal{V}\mathcal{3}$ does not perform complete rule checking over the format.

2. Type `write aig v3_alu.aig` or `write aig v3_alu.aig -symbol` to a write Boolean-level network into an AIGER file.

   Parameter `-Symbol` writes port and flip-flop names if they are specified in the input design, or an auxiliary name will be assigned to a nameless port or flip-flop whenever it is necessary.

3. Type `write rtl v3_alu.v` or `write rtl v3_alu.v -symbol` to output the Boolean-level network into Verilog.

   $\mathcal{V}\mathcal{3}$ enables users to output any networks into Verilog. However, it always models a Verilog design as a word-level network. The effect of parameter `-Symbol` and the introduction of the *clock* signal have already described in Section 1.4.

4. Use `print ntk` and `plot ntk` to retrieve network (structural) information.

   Although it is straight-forward to consider an AIG as a word-level network such that 1) all signals are limited to 1-bit, and 2) the only operator is `AND` (inverters are naturally recorded as inverting input signals), $\mathcal{V}\mathcal{3}$ specialize the gate types for Boolean-level networks for performance concerns.

   The output of `print ntk -verbose` is shown as follows:

```
v3> print ntk -verbose
=================================================
AIG_NODE                                   1292
-------------------------------------------------
V3_PI                                        11
V3_PO                                        16
V3_FF                                        16
=================================================
TOTAL                                      1319
```

Notice that the number of ports and flip-flops are more than those in Verilog and BTOR designs, because in this case word-level signals are bit-blasted into 1-bit signals. However, the total number of bits for describing ports and flip-flops remains the same.

Since AIGER design `alu.aig` synthesizes every word-level operator in functionally equivalent Verilog/BTOR counterparts by two-input `AND` gates, the number of signals, gates, and network levels are increased. As a result, it takes more time and memory to plot Boolean-level networks than word-level counterparts. We suggest users to plot Boolean-level networks for smaller levels and depths.

Chapter 2

Design Simulation

2.1 Introduction

In this tutorial we perform simulation on a vending machine design, and then dump out simulation results in both plain text and VCD waveform formats. The VCD output is supported by several third-party tools, and in this tutorial we use a successful commercial tool *Synopsys/SpringSoft Verdi* [2] for waveform visualization. Please also check Appendix B out for detailed descriptions of our simulation pattern format.

2.2 Prerequisites

Please download the latest $\mathcal{V}3$, and let `V3` be the $\mathcal{V}3$ directory. Make sure $\mathcal{V}3$ has been successfully installed such that an executable named `v3` is located under `V3`. In addition, check if the following files exists under `V3/design/vending`:

- ☐ `vending.v` : Verilog design for this tutorial.
- ☐ `input.pattern` : Input pattern for design simulation.

2.3 The Vending Machine Design

The Verilog design `vending.v` describes a vending machine with the following characteristics:

1. The design is always in one of the following three states:
   - `SERVICE_OFF` (encoded with 00): The vending machine is outputting changes and items for a granted request.
   - `SERVICE_ON` (encoded with 01): The vending machine is waiting for a request.
   - `SERVICE_BUSY` (encoded with 10): The vending machine is computing and preparing the changes for a granted request.

2. Four types of coins are defined:

   - `COIN_A` (encoded with 00): stands for 50 dollars.
   - `COIN_B` (encoded with 01): stands for 10 dollars.
   - `COIN_C` (encoded with 10): stands for 5 dollars.
   - `COIN_D` (encoded with 11): stands for 1 dollar.

3. Four types of items are offered:

   - `ITEM_A` (encoded with 00): 15 dollars per item.
   - `ITEM_B` (encoded with 01): 25 dollars per item.
   - `ITEM_C` (encoded with 10): 75 dollars per item.
   - `ITEM_D` (encoded with 11): 100 dollars per item.

4. The capacity of changes for each type of coins is 63 in the vending machine, and the maximum allowed input of each coin type is also 63. The machine may lose coins if the capacity is reached.[1]

5. Users are allowed to buy only one type of items in a request; however, the amount of the requesting item is available between 1 to 7. Assume that the vending machine has an infinite number of items for sale.

6. Two modes for users: *normal* mode and *force* mode, which will be explained later.

Initially, the system is out of service until a *reset* signal is arrived. The system then resets to its initial state while setting numbers of coins in the vending machine storage to be 5, 30, 10, and 20 for `COIN_A`, `COIN_B`, `COIN_C`, and `COIN_D`, respectively.

The behavior of the system is illustrated as follows:

1. User starts a request by setting 1) *coinInA, coinInB, coinInC, coinInD* to be the number of input coins, and 2) *itemTypeIn, itemNumberIn* for an item type and the amount to buy, respectively. However, the request is valid and will be granted in the next cycle if and only if the current state is `SERVICE_ON` and *itemNumberIn* is not zero.

2. If the request is valid, vending machine sets state to `SERVICE_BUSY` in the next cycle. Then the system prepares items and computes the number of coins for the change.

   The value of input coins and the cost of items in the request are stored in registers *inputValue* and *serviceValue*, respectively. There are three potential cases for a request in *normal* mode:

   - If *inputValue* is smaller than *serviceValue*, the vending machine returns no items.
   - If *inputValue* is not smaller than *serviceValue*, but the coins in the machine is not available to the change, the machine returns no items.
   - Else, the vending machine computes the numbers of coins for the change, and preparing items for the output.

[1] For instance, if there are 60 `COIN_A` in the machine, the number of stored `COIN_A` becomes 63 after inserting extra 10 `COIN_A`, and thereby 7 coins are lost by the machine due to out of storage.

Please notice that vending machine computes the number of coins for the change in all cases above, even if there are no items sold out. More importantly, the computation always guarantees to output the fewest number of coins. (For instance, if the change is 60 dollars and there is one 50 dollar and six 10 dollar coins in the machine, it will always return the 50 dollar and a 10 dollar. However, if there is no 50 dollars, it returns six 10 dollars instead.)

The system supports another mode for customers, the *force* mode. In this mode, the system tries to supply as more items as possible according to a request: Let the number of items in the request to be N. If the service is not available for N items, the machine tries to service the maximum number of items ($N' < N$) that is available. Evidently, it usually takes more cycles to complete a request in *force* mode.

3. If the computation of change is completed and the system is ready for output, vending machine sets 1) state to `SERVICE_OFF`, 2) change to (*coinOutA, coinOutB, coinOutC, coinOutD*) and 3) items (*itemTypeOut, itemNumberOut*) in the next cycle according to the final result.

2.4 Simulating Word-level Networks

We describe the procedure of simulating word-level networks with a sequence of input patterns by $\mathcal{V}3$. Please refer to ***V3 User Guide*** for the format of input pattern and output text files.

1. Type `read rtl vending.v` to read the vending machine design.
2. Type `print ntk -primary` and `print ntk -verbose` to retrieve network information. Note that the primary input *forceIn* enables *force* mode of the vending machine system, and the primary output *serviceTypeOut* stands for the state of the system.
3. Next, to simulate the network with input patterns specified in the pattern file `input.pattern`, please type either `sim ntk -input input.pattern -output sim.output` or `sim ntk -input input.pattern -output sim.output -event`. Please refer to Appendix B for the format of an input pattern file.

   The optional parameter `-Event` enables event-driven simulation, i.e. the value of a signal is re-evaluated only if some values of its fanin signals are changed. In most cases, event-driven simulation improves the efficiency of simulation. A sample output of both commands are as follows:

   ```
   v3> sim ntk -input input.pattern -output sim.output
   256 Patterns are Simulated from input.pattern   (time = 0.05   sec)
   v3> sim ntk -input input.pattern -output sim.output -e
   256 Patterns are Simulated from input.pattern   (time = 0.03   sec)
   ```

   Contents in the file `input.pattern` involves 256 patterns; however, only four patterns reveal user requests to the vending machine. (The first pattern is the only pattern that resets the system.) System states under the other patterns are not `SERVICE_ON` and thereby those patterns are don't cares to the system. Thus we set all of them to don't care symbol **X** for highlighting the only four active patterns.

- The system resets to the initial configuration such that $(\texttt{COIN\_A}, \texttt{COIN\_B}, \texttt{COIN\_C}, \texttt{COIN\_D}) = (5, 30, 10, 20)$ after the first cycle.
- The system grant the first request in the next cycle under pattern: *coinInA* $=$ *coinInC* $= 0$, *coinInB* $= 7$, *coinInD* $= 10$, *itemTypeIn* $=$ `ITEM_A`, *itemNumberIn* $= 5$, and *forceIn* $= 0$. Therefore, the total value of input coins *inputValue* $= 80$ and the total cost of required items *serviceValue* $= 75$. Clearly, the change is 5 dollars and it is available for the vending machine. After several cycles of computation, the state turns into `SERVICE_OFF` and the output should be *coinOutA* $=$ *coinInB* $=$ *coinInD* $= 0$, *coinInC* $= 1$, *itemTypeIn* $=$ `ITEM_A`, *itemNumberIn* $= 5$ (see line 9 of the output file). In the meanwhile, the numbers of coins in the machine storage becomes $(5, 37, 9, 30)$.
- The second request sets *coinInA* $= 12$, *coinInB* $= 7$, *coinInC* $= 15$, *coinInD* $= 34$, *itemTypeIn* $=$ `ITEM_B`, *itemNumberIn* $= 3$, and *forceIn* $= 0$. Similarly, *inputValue* $= 779$, *serviceValue* $= 75$, and thereby the change is 704 dollars. This coincides with the output in the 34 line of the output file: *coinOutA* $= 1$, *coinInB* $=$ *coinInC* $= 0$, *coinInD* $= 4$, *itemTypeIn* $=$ `ITEM_B`, *itemNumberIn* $= 3$. As a consequence, the numbers of coins in the storage becomes $(3, 44, 24, 59)$.
- Please check the outputs of the remaining two requests.

4. Use the command `plot trace input.pattern vending.vcd` to simulate the patterns again and then write simulation results into VCD waveform format.

   After creating the VCD file `vending.vcd`, we show how to visualize the simulation trace using *Synopsys/SpringSoft Verdi* [2]:

   (a) Start Verdi.

   (b) (Read Verilog)

   (c) (New waveform) and read the VCD output file (it will write into FSDB format)

   (d) (Probing signals and see their values on the waveform)

   (e) Quit Verdi.

2.5 Simulating Boolean-level Networks

We describe the procedure of simulating a Boolean-level network in $\mathcal{VIS}$, which is very similar as explained in the last section. The network is created by synthesizing the original Verilog design into a Boolean-level network. In this part, `input.pattern` describes input patterns for simulating the resulting Boolean-level network instead.

1. Type `read rtl vending.v` to read the vending machine design.
2. Type `blast ntk` to transform the word-level network into a Boolean-level network.
3. Repeat steps 2 and 3 in Section 2.4.
4. Type `write rtl aiger-vending.v -symbol` to write the network into Verilog.

Because Verdi requires the Verilog design the waveform file is referenced from, we have to write out the transformed network for waveform visualization. Choose the Verilog design `aiger-vending.v` for Verdi and repeat step 4 in Section 2.4 to complete the tutorial.

Chapter 3

Design Verification – Part 1

3.1 Introduction

$\mathcal{V}\mathcal{3}$ is equipped with certain successful SAT-based model checking algorithms for checking both safety (i.e. something *bad* should never happen) and liveness properties (i.e. something *good* should eventually happen) on both Boolean-level and word-level networks. In addition, these algorithms are subtly implemented in a way such that a variety of SAT and SMT solvers can dynamically serve as the underlying formal engines.

Here we list all supported property checking techniques in the latest $\mathcal{V}\mathcal{3}$ package:

- (Random) simulation-based: `VERify SIM`.
- Bounded Model Checking (BMC) [5]: `VERify BMC` or `VERify UMC -NOProve`. The former creates a n-copy of the network, optimize it, and then check the satisfiability of the resulting instance. On the other hand, the latter characterizes different copies of a network with different propositional variables, encodes a BMC instance with propositional formula, and solve the formula by SAT solving.
- Unbounded Model Checking (UMC, BMC with Induction) [14, 11]: `VERify UMC`.
- Interpolation-based Model Checking [12] (an implementation of *NewITP* [16]): `VERify ITP`. By default this command computes backward reachability of a property (as described in the *NewITP* paper); however, an optional parameter `Reverse` renders *NewITP* to construct forward reachability instead.
- Property Directed Reachability [10] (a.k.a IC3 [7]): `VERify PDR` implements PDR and `VERify PDR -Incremental` follows IC3.
- Sequential Equivalence Checking (SEC) [13]: `VERify SEC`.
- K-liveness Algorithm [9]: `VERify KLIVE`.

Except for K-liveness algorithm that is designed only for liveness checking, all the above techniques are basically safety checkers. For liveness checking, $\mathcal{V3}$ implicitly transforms a liveness property into a safety version [4] for a safety checker.

In this tutorial, we demonstrate how to verify designs using $\mathcal{V3}$ while answering the following questions:

- how to specify a safety or a liveness property in $\mathcal{V3}$
- how to specify a SAT or SMT solver as the underlying formal engine to model checking algorithms
- how to configure verbosity outputs in design verification
- how to commence a verification algorithm for property checking
- how to confirm verification results (i.e. proofs or counterexamples)
- how to dump out and restore verification results

In this tutorial we focus on verifying single safety or liveness property by a certain type of model checking algorithm. All properties are expressed in their counterexample forms. In case that you are interested in specifying multiple properties and verifying a set of properties using a portfolio of model checking algorithms, please continue on Chapter 4 after finishing this tutorial.

3.2 Prerequisites

Please download the latest $\mathcal{V3}$, and let `V3` be the $\mathcal{V3}$ directory. Make sure $\mathcal{V3}$ has been successfully installed such that an executable named `v3` is located under `V3`. In addition, check if the following files exists under `V3/design`:

- ☐ `traffic/traffic.v` : A traffic light design for Section 3.3.
- ☐ `philosopher/ph64.v` : A sample dinning philosopher problem for Section 3.5.
- ☐ `philosopher/philosopher.v` : The model of a philosopher for the dining philosopher problem.

3.3 Verify the Traffic Light Design

To begin with, we illustrate how to verify safety properties in a simple traffic light design *traffic.v*. Here we briefly describe the behaviour of the system:

- Three light signs: `RED`, `GREEN` and `YELLOW`.
- `RED` light turns into `GREEN` after 60 cycles, `GREEN` light turns into `YELLO` after 40 cycles, and `YELLOW` light turns into `RED` after 5 cycles. There is a count down counter which shows the number of remaining cycles to trigger the next transition of the light sign.

- Initially (or after *reset*), light is in `RED` and it will immediately turn into `GREEN` in the next cycle.

We made primary outputs *p1*, *p2* and *p3* corresponding to the following three safety properties we want to verify, respectively.

- **F** (*p1*): Light signs can either be `RED`, `GREEN`, or `YELLOW`.
- **F** (*p2*): The count down counter cannot exceed 60, 40, and 5 if current light sign is `RED`, `GREEN`, and `YELLOW`, respectively.
- **F** (*p3*): Light signs can never turn into `YELLOW`.

We set `traffic.v` as the design under verification (DUV). Please change current working directory to `V3_DESIGN/traffic`.

1. Type `read rtl traffic.v` to read the design.
2. Type `print ntk -primary` to confirm indices of primary outputs which serve as (bad state) property logic.

   It is important that all primary ports and flip-flops of a DUV must be blasted into individual bits beforehand, or model checking of the design might end up with unexpected results. Since the last primary output of the current network is a eight-bit bit-vector signal, we transform current network into another word-level network such that primary ports and flip-flops are blasted into bits. This is achieved by the command `blast ntk -primary`, where the optional parameter `-Primary` limits bit-blasting engine to work on primary ports and flip-flops only. Type `print ntk -primary` to make a double check.
3. Type `print solver` to check the type of the formal engine for verification algorithms. By default, the solver should be MiniSat, and the output of the command looks like:

   ```
   v3> print solver
   Active Solvers:  "MINISAT"
   ```
4. Type `set solver -boolector` to change current solver to SMT solver Boolector. Notice that users can also use `-Minisat` to set current solver back to MiniSat.
5. Use `set safety 0` to specify the first safety property **F** (*p1*) to the current DUV.

   If the optional parameter `-Name` of the command `SET SAFEty` is not specified, $\mathcal{V}3$ automatically assigns a fresh property name "`pi`" to the property, where `i` is an integer that starts from `1`. A sample output message reported by the command is shown as follows:

   ```
   v3> set safety 0
   Property p1 has been set sucessfully !!
   ```
6. Type `verify umc p1` to verify the property "`p1`" using UMC.

   Here we choose UMC as the verification algorithm for verifying the property "`p1`"; however, users can also try either ITP or PDR and check if all the results are identical. Based on our

knowledge to the design, the property should be true, i.e. there is no counterexamples and thus *p1* should never happen. A sample output of UMC is shown as follows:

```
v3> verify umc p1
Inductive Invariant found at depth = 2 (time = 0 sec)
```

7. Type `set safety 1` to set **F** (*p2*) as the second safety property ("`p2`").

8. Type `verify itp p2` to verify the property "`p2`" using *NewITP*.

   Again, it is fine to choose either UMC or PDR as the verification algorithm. The property should also be true, and a sample output of ITP is shown as follows:

```
v3> verify itp p2
Inductive Invariant found at depth = 3 (time = 0.01 sec)
```

9. Type `set safety 2` to set **F** (*p3*) to be the third safety property ("`p3`").

10. Type `verify pdr p3` to verify the property "`p3`" using PDR.

    Apart from the former two properties, property "`p3`" is a false property because there exists a trace from initial states to a state with the light sign `YELLOW`. Therefore, there exist a counterexample for the property. A sample output of PDR is as follows:

```
v3> verify pdr p3
0 :  0 0
1 :  0 1 0
2 :  0 2 2 0
3 :  0 1 1 1 0
4 :  0 1 1 2 1 0
5 :  0 1 1 1 2 2 0
6 :  0 1 1 1 1 1 1 0
7 :  0 1 1 1 1 1 2 2 0
8 :  0 1 1 1 1 1 1 1 1 0
9 :  0 1 1 1 1 1 1 1 2 2 0
10 :  0 1 1 1 1 1 1 1 1 1 1 0
11 :  0 1 1 1 1 1 1 1 1 1 2 2 0
12 :  0 1 1 1 1 1 1 1 1 1 1 1 1 0
13 :  0 1 1 1 1 1 1 1 1 1 1 1 2 1 0
14 :  0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 0
15 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
16 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 0
17 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
18 :  0 1 1 1 1 1 1 1 1 1 2 2 3 4 2 3 2 3 2 0
19 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
20 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 0
21 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 3 4 1 1 2 2 0
22 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
23 :  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 0
24 :  ...  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

```
25 :  ...  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 0
26 :  ...  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
Counter-example found at depth = 28 (time = 0.42 sec)
```

Please notice that the output message does not state that the discovered counterexample has 28 cycles. Instead, it is the number of frames in PDR when the counterexample is found. The real length of the counterexample can be reported by the command `WRIte REsult`, which will be explained later.

11. Type `check result p3` to confirm verification result (in this case, a counterexample) of property "`p3`".

    If the verification result is correct (i.e. a real counterexample), the following message is reported:

    ```
    A real counter-example is found for property "p3".
    ```

    Otherwise if the verification is incorrect, the following message is reported:

    ```
    A spurious counter-example is found for property "p3".
    ```

12. Type `write result p3 p3.pattern` to the write out the verification result (i.e. a counterexample) of property "`p3`".

    The format of counterexample output is the same as the input pattern for simulation as described in the Appendix B. The counterexample of the property "`p3`" found by PDR has 43 cycles from the output file `p3.pattern`.

13. Type `check result p3 -trace p3.pattern` to check whether `p3.pattern` represents a real verification result (in this case, counterexample) for the property "`p3`".

    Notice that if the result in `p3.pattern` is real, $\mathcal{V}3$ automatically sets it as the verification result of the property "`p3`". Otherwise, the result in `p3.pattern` is abandoned.

14. Use the command `plot result p3 p3.vcd` to plot the verification result (in this case, counterexample) of the property "`p3`".

    Similar to the function of command `PLOt TRace`, the counterexample is dumped into a VCD waveform file `p3.vcd`. Please use waveform viewers such as Synopsys/SpringSoft Verdi [2] to visualize the counterexample as described in Chapter 2.

The procedure of verifying a Boolean-level network of the same design *traffic.v* with the same properties is very similar to the procedure described above. Please use command `BLAst NTk` right after reading the design *traffic.v* to transform the word-level network into a Boolean-level network. Next, perform model checking on the Boolean-level network following the same procedure as described above.

3.4 Configure Verification Verbosity Output

$\mathcal{V}3$ allows users to configure verbosity output of a verification algorithm in the run-time. We use the same design to demonstrate how to configure verification verbosity output settings, as follows:

1. Type `read rtl traffic.v` to read the design again.

2. Type `blast ntk -primary` to create a word-level network as the DUV.

3. Type `set safety 2` to specify property `p1` = **F** (*p3*), i.e. the only failing property of the design.

4. Type `print report` to see enabled verification verbosity outputs.

   By default, only `-RESUlt` (stands for interactive verification status report) and $\mathcal{V}3$ -Usage (stands for run-time usage) are enabled, and thus the following shows a sample output:

   ```
   Verification Report :  "-Interactive -Usage " ON
   ```

5. Type `verify pdr p1` to verify the property "`p1`".

   We use PDR in this example; however, users can also apply UMC or ITP to verify the property. Please notice that the interactive verification output might varied from executions.

6. Type `set report -all -on` to enable all verification verbosity output settings.

   After executing the command, $\mathcal{V}3$ automatically reports the updated verification verbosity output settings. A sample report is shown as follows:

   ```
   Verification Report :  "-Interactive -Endline -Solver -Usage -Profile " ON
   ```

7. Type `verify pdr p1` to verify the property"`p1`" using PDR again and see the differences in verbosity output.

   Because all verbosity outputs are enabled, the verification report includes all information about the verification process, including solver information and profiling data.

8. Type `set report -all -off` to disable all verification verbosity output. Then, type `verify pdr p1` to verify the property using PDR again.

   Now all verbosity outputs are disabled, there is no message output, even the verification result (i.e. the existence of a counterexample or inductive proof) will not be reported.

9. Use `set report -reset` the reset verification verbosity output settings. Then, type `set report -result -off` to disable the interactive status report only.

   This configuration enables the report of the final verification result as well as the time usage only. It is to our opinion the best configuration for benchmarking. A sample output of the two commands is shown as follows:

   ```
   v3> set report -result -off
   Verification Report :  "-NOInteractive -Usage " ON
   v3> verify pdr p1
   Counter-example found at depth = 20 (time = 0.03000 sec)
   ```

   Please also note that disabled verification verbosities are absent from the output of command `PRInt REport` ; however the parameter `-RESUlt` is always concealed.

3.5 Verify the Dinning Philosopher Problem

We show how to verify liveness properties by $\mathcal{V}3$ by taking the dinning philosopher design `ph64.v` as the target design. A brief description of the design is as follows:

The design contains 64 philosophers sitting at a round table, which are identical instances of the philosopher module *philosopher* described in `philosopher.v`. The characteristics of the *philosopher* module are:

- A philosopher must lies in one of the following four phases: `THINKING`, `READING`, `EATING`, and `HUNGRY`. Initially, every philosopher is `THINKING`.
- The transition from one phase to another for a philosopher depends on itself as well as the two adjacent philosophers.

Now, we are going to verify the following properties:

1. `p1` = **F** ($safe$) describes the fact that there does not exist two `EATING` philosophers sitting side by side. (It is an important setting to the game of dinning philosopher.)
2. `p2` = **G** ($live$) expresses that eventually a philosopher is not `EATING`. In other words, it states that a philosopher can not situated in `EATING` infinitely.

The verification of the design is proceeded in the following steps:

1. Type `read rtl ph64.v` to read the dinning philosopher problem.
2. Since the network is hierarchical and there exists *philosopher* 64 sub-modules (check out command `print ntk -verbose`), type `flatten ntk` to flatten the network.

   Another solution is to use `read rtl ph64.v -flatten` in the first step. The optional parameter `-flatten` forces *QuteRTL* to flatten a hierarchical circuit for $\mathcal{V}3$. However, in our experience *QuteRTL* usually flattens a design much slower than $\mathcal{V}3$ does.
3. Type `set safety 0` to set the first (safety) property "`p1`", and then verify the property using UMC, ITP, and PDR. The property should be proven by all of them.
4. Type `set liveness -invc PO[1]` to specify the second (liveness) property "`p2`". Then, use UMC, ITP, and PDR to verify the property.

   The property is true; however, please notice that UMC may not prove the property efficiently. Therefore we suggest users to add parameter `-Max-depth` to restrict the maximum number of cycles for UMC. According to our tests, set `-Max-depth 30` cancels UMC on verifying the property in about 10 seconds. A sample output of verifying the liveness property is depicted as follows:

```
v3> verify umc p2 -max 30
UNDECIDED at depth = 30 (time = 12.067 sec)
v3> verify itp p2
1 :  1 1 0
2 :  1 1 0
Inductive Invariant found at depth = 2 (time = 0.084092 sec)
v3> verify pdr p2
0 :  0 0
1 :  0 1 0
2 :  0 1 2 0
Inductive Invariant found at depth = 4 (time = 0.056331 sec)
```

Please also perform model checking on a Boolean-level network of the design `ph64.v` following the similar procedure after transforming the word-level network into a Boolean-level network using command `BLAst NTk`.

Chapter 4

Design Verification – Part 2

4.1 Introduction

In Chapter 3 we have exploited $\mathcal{V}3$ to verify one safety or liveness property by a certain model checking algorithm. In this tutorial, we consider a more general model checking problem: Given a design and a set of properties, which is in general a combination of safety and liveness properties, we now attempt to perform *multiple property checking* using *a portfolio of model checking algorithms* on a *multi-core* machine.

4.2 Prerequisites

Please download the latest $\mathcal{V}3$, and let `V3` be the $\mathcal{V}3$ directory. Make sure $\mathcal{V}3$ has been successfully installed such that an executable named `v3` is located under `V3`. In addition, check if the following files exists under `V3/design/quteTK6280`:

- ☐ `quteTK6280.btor` : BTOR design for Section 4.3.
- ☐ `quteTK6280.prop` : Property specification for Section 4.3.
- ☐ `quteTK6280.aig` : AIGER design for Section 4.4.

4.3 Multiple Property Checking of a Word-level Network

In this section, we focus on verifying a word-level network created from a BTOR input `quteTK6280.btor`. Different from the procedure of property creation explained in Section 3 that involves one command for specifying one property, we write all the properties to be verified into a *PROP* format file and create multiple properties at one time by parsing the file. Please refer to Appendix C for the *PROP* format for property specification.

Please set current working directory to `V3/design/quteTK6280` for the tutorial.

1. Type `read btor quteTK6280.btor -s` to read the BTOR design.

2. Type `blast ntk -primary` to generate DUV of which primary ports and flip-flops are blasted into bits.

3. Use `read prop quteTK6280.prop -prop` to read all properties specified in the *PROP* format file `quteTK6280.prop`.

   If something unexpected happens during either file parsing, predicate synthesis, or property recording, corresponding error or warning message will be reported. Otherwise, $\mathcal{V}3$ gives a statistic to the total number of properties that are successfully created. Here is a sample output message reported by the command:

   ```
   v3> read prop quteTK6280.prop -prop
   Totally 1557 Safety, 11 Liveness Properties are Added.
   ```

4. Type `run -time 1800 -memory 8000 -thread 8` to commence $\mathcal{V}3$ portfolio-based model checking.

   This command `RUN` defines the upper bound of three resource types, and each one of them is associated with one parameter:

   - `-TIMEout` : the maximum allowed wall-clock time in seconds (30 minutes in this example)
   - `-MEMoryout` : the maximum allowed memory usage in megabytes (8 GB in this example)
   - `-THReadout` : the maximum number of model checking algorithms running in parallel (eight in this example)

   Command `RUN` thus considers the verification of all properties (in this case, 1557 safety and 11 liveness) under these resource bounds. It should be noted that they are treated as *tight* bounds, which means that every procedure will be terminated and any results will be ignored if any of the bounds is exceeded. However, some threads are probably not terminated when a bound is found exceeded, and thus the command will return after all threads are successfully terminated.

   The mechanism working underneath the command is much complicated. Roughly speaking $\mathcal{V}3$ elaborates properties, simplifies the resulting DUV, initializes a proper order of properties, and then commences a portion of model checking algorithms for solving properties in parallel. After a period of time $\mathcal{V}3$ determines whether to cancel some running algorithms or to initialize others according to resource usage.

   During the model checking process $\mathcal{V}3$ reports messages about verification results. The following is a sample output of the command:

```
v3> run -time 1800 -memory 8000 -thread 8
Counter-example for property b0002 found at time = 1.32953 sec.
Counter-example for property b0003 found at time = 1.32961 sec.
Counter-example for property b0004 found at time = 1.32962 sec.
Inductive Invariant for property b0007 found at time = 1.32963 sec.
⋮
Inductive Invariant for property j0004 found at time = 122.378 sec.
Counter-example for property j0010 found at time = 122.378 sec.
Inductive Invariant for property b1230 found at time = 132.346 sec.
Inductive Invariant for property j0006 found at time = 155.381 sec.
Totally 1557/1557 Safety 11/11 Liveness Solved.
```

5. After model checking, verification results (i.e. counterexamples or proofs) are recorded such that users are permitted to use commands `CHEck REsult` to confirm the correctness of a result or `WRIte REsult` to dump the result out.

4.4 Multiple Property Checking of a Boolean-level Network

AIGER 1.9 [6] is an extension of AIGER 1.0 [6] format such that property-related features are introduced. These features are annotated with new symbols **B**, **C**, **J**, **F** in the header respectively for the numbers of safety properties, invariant constraints, justice properties, and fairness constraints. As a consequence, such an AIGER design defines **B** safety properties and **J** liveness properties.

`quteTK6280.aig` is an AIGER 1.9 file that describes a Boolean-level design that is functionally equivalent to the word-level design `quteTK6280.btor` expresses, and it defines the same set of properties as `quteTK6280.prop` specifies. The only difference is that some properties are rewritten and partially elaborated in `quteTK6280.aig` since the expressiveness of AIGER 1.9 is less flexible than that of *PROP*.

The script for model checking `quteTK6280.aig` is much simpler, since both design and properties are described in the same file.

1. Type `read prop quteTK6280.aig -aig` to read the design and properties.
2. Type `run -time 1800 -memory 8000 -thread 8` to commence $\mathcal{V}3$ portfolio-based model checking. And that's it!

Appendix A

File Format of FSM Specification for Design Intent Extraction

$\mathcal{V}\mathcal{3}$ extracts finite state machines (FSMs) from a network. Specifically, extracted FSMs are generally (*abstract*) *state transition systems* where (abstract) states are defined over a set of *predicates* and functions of states are defined by the topology of *predicate ancestries*. A predicate ancestry is a hierarchical binary tree, where each node of the tree represents a predicate whereas every predicate is associated with two sets of nodes: *true* and *false*.

Command `WRIte FSM` enables $\mathcal{V}\mathcal{3}$ to output the specification of abstract state transition systems such that the same FSMs can be extracted again according to the specification using command `ELAborate FSM`. Apparently, this also offers users a flexible approach to obtain customized FSMs from $\mathcal{V}\mathcal{3}$. We then describe the format of abstract state transition systems for FSM extraction, which in reality, describes topology of predicate ancestries instead of FSMs.

Literally the format is composed of four ordered and indispensable parts:

1. **Header:** One line that starts with a keyword `fsm`, followed by three ordered non-negative integers: the number of predicates P, the number of predicate ancestries S, and the number of finite state machines F.

2. **Predicates:** P lines that characterize names and functions of predicates in Verilog format.[1] Every individual line corresponds to the specification of a predicate. The name of a predicate is optional; however, it should be specified at first and associated with the representation of the predicate via a reserved connectivity "`:=`".

3. **Predicate Ancestries:** S lines that illustrates topology of predicate ancestries. Every individual line denotes a pre-order traversal of nodes of a corresponding predicate ancestry. Each node is represented by three elements: a net representation idx, and the number of children in its *true* and *false* branches. Basically idx is a non-negative integer that relates the predicate of a node with the idx-th predicate specified in **Predicates** if they have the same function. However, if their functions are inverted, an inversion (negation) symbol (i.e. `~`, `-`, or `!`) is added

[1] Currently we do not support representations with repetitions (e.g. `{4{a}}`) or integers (e.g. `2014`).

prior to idx. On the other hand, if a branch has no children and it corresponds to an empty function (i.e. this leaf node represents a formula that is unsatisfiable according to the behavior of the network), we put symbol `X` instead of `0` for the number of children under the branch. During the traversal of a node, the traversal of the *false* branch always comes after that of the *true* branch. Space characters are inserted between adjacent elements in the specification.

4. **Finite State Machines:** F lines for specifications of abstract state transition systems. Every single line corresponds to the specification of a FSM, which contains a sequence of elements separated with spaces. Each element is a non-negative integer that denote the index of a predicate ancestry (w.r.t the specification in **Predicate Ancestries**).

An example is illustrated in Figure A.1. A comment in the specification starts from "//".

```
// <fsm> <P : #predicates> <S : #predicate ancestries> <F : #FSMs>
fsm 3 5 3
// P lines of predicate functions
a + b == 3'b001              // the function of the first variable
var2 := x >= y               // specifying both name and function
⋮
// S lines of predicate ancestry topologies
3 1 2 -5 0 X 2 0 1 !8 X 0 ⋯  // the first predicate ancestry
⋮
// F lines of FSM specifications
1                            // components of the first FSM
2 1                          // components of the second FSM
⋮
```

Figure A.1: An Example of the Specification for Abstract State Transition Systems.

Appendix B

File Format of Input Pattern for Design Simulation

The format of the input pattern file is designed for two main purposes:

1. for representing both simulation and counterexample traces, and
2. for simulating on both word-level and Boolean-level networks

Therefore it is the common format of counterexample outputs for design debugging (generated by command `WRIte REsult`) and pattern inputs for design simulation (required by command `SIM NTk -Input` and `PLOt TRace`).

A simulation pattern file consists of three parts, however, the last part is optional:

1. **Header:** The first line contains two numbers P, V separated with a space: Positive integer P is the total number of patterns in the file while non-negative integer V is the total number of bits for a pattern.
2. **Patterns:** The next P lines describe patterns for simulation. Every pattern in the file is a bit-vector with V bits, and bit is either `0`, `1`, or `X`. While the former two clearly stands for Boolean values 0 and 1, the last one characterizes the don't care symbol in ternary simulation. [1]

   A pattern stands for an assignment to primary input and inout port signals. In our specification the order of ports conforms to the following rule:

   $$\texttt{pattern := } \cdots < inout_1 >< inout_0 > \cdots < input_1 >< input_0 >$$

   For instance, given a network with two primary inputs `a, b` and a primary inout `out` with bit-widths 2, 4, and 3, respectively, the pattern `1001011XX` indicates that $\texttt{a} = \texttt{2'bXX}, \texttt{b} = \texttt{4'b1011}$, and $\texttt{c} = \texttt{3'b100}$.

[1] Ternary semantics extends binary semantics, for instance, `AND(0, X) = AND(X, 0) = 0` and `AND(1, X) = AND(X, 1) = AND(X, X) = INV(X) = X` for AIG networks.

The rule above enables users to simulate word-level and Boolean-level versions of a design using a common pattern file: Since the order of ports after bit-blasting follows the same order in the original network, and the signal mapped from a least significant bit of a bit-vector signal is always ordered prior to signals mapped from more significant bits, it is clear that the bit mapping between patterns and port signals remains after network optimization and also transformation.

If a network has no primary inputs and primary inouts, V becomes zero. In such cases, P patterns are represented as P empty lines.

3. **Comments:** After specifying P patterns, the remaining part of the file leaves for comments. Currently *VIS* ignores everything after reading input patterns.

Figure B.1 illustrates a sample pattern input. Please note that lines starting with "//" are not part of the format.

```
1024 12        // 1024 patterns, each pattern ∈ {0,1,X}^12
0110XX110XX1   // 1st pattern
XXX110XXXXX0   // 2nd pattern
⋮
XX1XXXXXXXXX   // last pattern
comments start here ...
⋮
```

Figure B.1: A Sample Pattern File for Design Simulation and Counterexample Output.

Appendix C

File Format of Property Specification for Design Verification

Describing different abstraction levels of different functions of a hardware system is undoubtedly prevalent in an industrial verification plan nowadays. This suggests that functional verification of a hardware system with the existence of multiple properties has become a trend. As an industrial-strength model checker, *V3* supports two different interfaces for users to specify properties:

The simplest ways to create a property is via command-line interface: Using `SET SAFEty` and `SET LIVEness` to set a safety and a liveness property, respectively. However, the following problems are frequently encountered: First of all, apart from constraints or invariants generated from FSMs, property logic must be already synthesized and serves as a primary output for both commands. (There is no way to either synthesize a new predicate with an arbitrary expression or creating a new primary output logic by *V3* commands.) Next, for a huge number of properties to be verified, it is not practical to create tons of auxiliary primary outputs for the commands beforehand.

On the other hand, *V3* enables users to write properties in a file for multiple property checking, and then all properties will be stored after they are read successfully by command `READ PROperty -Prop`. This resolves all the issues that might be encountered in the command-line approach. However, it is not necessary if there is only a few properties in your verification plan.

We call the property specification file format the *PROP* format. Lines that start with "`//`" are regarded as comments and will be ignored. A typical *PROP* file can be partitioned into three required segments:

1. **Header:** Similar to previous *V3* file formats, the first part is the header that starts with a keyword *prop*, followed by two positive integers: the number of predicates D and the number of properties P.

2. **Predicates:** The following D lines (excluding comments) describe the expressions of predicates. The syntax of an expression should be compatible with Verilog.[1] The format currently does not allow attaching a name to a predicate.

[1] As mentioned in Appendix A, we do not support representations with repetitions (e.g. `{7{b}}`) or integers (e.g. `2020`).

3. **Properties:** In the last part P properties are specified, with one property per line. The name of a property is optional; however, it should be specified at first and associated with the specification of the property via a reserved connectivity "`:=`".

   The expressiveness of properties that can be specified by *PROP* format and from commands `SET SAFEty` and `SET LIVEness` are the same under the omission of invariants. In contrast to the latter that partitions a specification into several independent parts (e.g. bad signal, invariant constraints and fairness constraints) and annotated with different predefined keywords, a property should be specified in the form of LTL under *PROP* format. Here are the templates for safety and liveness properties (in the counterexample form):

   Safety: **F** b **G** c_1 c_2 $\cdots$ c_n, where b is the index of the bad signal (required) and every c_i stands for the index of an invariant constraint that are respectively defined in **Predicates**. **F** and **G** in the formula respectively correspond to temporal operators $\Diamond$ (i.e. *eventually*) and $\Box$ (i.e. *globally*) in LTL.

   Liveness: **G** c_1 c_2 $\cdots$ c_n **GF** f_1 f_2 $\cdots$ f_m, where every c_i and f_i respectively stand for the index of an invariant constraint and that of a fairness constraint specified in **Predicates**.

Figure C.1 shows an example that defines two safety and one liveness properties under five predicates.

```
// ⟨keyword: prop⟩ ⟨integer:#predicates D⟩ ⟨integer:#properties P⟩
prop 5 3 // D = 5, P = 3
// D lines for predicate functions
a + b == 3'b001
x >= y && {!z, 2'd3} < (a >> 3)
⋮
// P lines for property specifications
F 0
safe2 := F 0 G 1 2
live1 := G 1 GF 3 4
```

Figure C.1: An Example of Property Specification in *PROP* Format.

Bibliography

[1] Hardware Model Checking Competition. *http://fmv.jku.at/hwmcc/*.

[2] Synopsys Verdi. *https://www.synopsys.com*.

[3] A. Biere. The AIGER And-Inverter Graph (AIG) Format. *Available at fmv. jku. at/aiger*, 2007.

[4] A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[6] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and Beyond. *Available at fmv.jku.at/hwmcc11/beyond1.pdf*, 2011.

[7] A. Bradley. SAT-based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[8] R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-precise Modelling of Word-level Problems for Model Checking. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 33–38. ACM, 2008.

[9] K. Claessen and N. Sörensson. A Liveness Checking Algorithm that Counts. In *FMCAD*, pages 52–59, 2012.

[10] N. Een, A. Mishchenko, and R. Brayton. Efficient Implementation of Property Directed Reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.

[11] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[12] K. McMillan. Interpolation and SAT-based Model Checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003.

[13] A. Mishchenko, M. Case, R. Brayton, and S. Jang. Scalable and Scalably-verifiable Sequential Synthesis. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 234–241. IEEE, 2008.

[14] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 127–144. Springer, 2000.

[15] D. Thomas and P. Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer, 2002.

[16] C.-Y. Wu, C.-A. Wu, C.-Y. Lai, and C.-Y. Huang. A Counterexample-guided Interpolant Generation Algorithm for SAT-based Model Checking. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2013.