Verilog2SMV: A Tool for Word-level Verification

Ahmed Irfan*† , Alessandro Cimatti*, Alberto Griggio*, Marco Roveri* and Roberto Sebastiani†
*Fondazione Bruno Kessler, Italy
Email: `[lastname]`@fbk.eu
†University of Trento, Italy
Email: `[firstname].[lastname]`@unitn.it

***Abstract*—Verification is an essential step of the hardware design lifecycle. Usually verification is done at the gate level (Boolean level). We present *verilog2smv*, a tool that generates word-level model checking problems from Verilog designs augmented with assertions. A key aspect of our tool is that memories in the designs are treated without any form of abstraction. *verilog2smv* can be used for RTL verification by chaining with a word-level model checker like NUXMV. To this extent, we present also some experimental results over Verilog verification benchmarks, using *verilog2smv*+NUXMV as a tool-chain.**

I. Introduction

Verification has become a fundamental step to ensure safety and reduce overall cost of the design and production of hardware systems. Hardware systems are often designed by specifying their behavior using high-level programming languages like Verilog or VHDL. These high-level designs are then (automatically) synthesized into Register Transfer Level (RTL) designs. The RTL specification contains the word-level information of a design. At RTL memories present in a design are either treated as a whole or expanded into individual elements. RTL designs are synthesized into gate-level logic designs and further synthesized into physical-level logic designs. In general, formal verification is performed at the gate level, where the structural information of the original design is almost lost. There are successful attempts to perform formal verification directly at RTL, though these approaches use Model Checking (MC) techniques at the Boolean-level.

Our goal is to lift the verification of hardware designs from gate level to RTL, exploiting recent model checking algorithms based on Satisfiability Modulo Theories (SMT), like those provided by the NUXMV model checker [1], [2]. In particular we aim at handling efficiently designs with memories.

This paper describes *verilog2smv*, an opensource tool that takes a Verilog design with simple SystemVerilog assertions and generates a MC problem at RTL in two formats: the BTOR [3] and NUXMV [4] formats. We choose Verilog [5] and SystemVerilog assertions [6] as they are widely used in the hardware design automation industry. Our tool handles natively memories in the design using fixed size bit-vectors and arrays. This allows to avoid both the blasting of memories into bit-vectors and their abstraction (this is the culprit characteristic of our tool). We remark that, to the best of our knowledge, the majority of the Verilog conversion tools usually loose memory information by either abstracting or expanding memories.

The first and the last authors are supported by Semiconductor Research Corporation (SRC) under GRC Research Project 2012-TJ-2266 WOLF. We thank Clifford Wolf for support with Yosys code.

This paper makes the following contributions. First, we provide a verification tool-chain, based on Yosys [7]– a Verilog synthesizing tool, and on NUXMV [2]– a modern model checker. We have extended Yosys to generate MC problems from Verilog and SystemVerilog assertions, in the NUXMV format, to be then analyzed with the advanced model checking algorithms provided by NUXMV. Moreover, the tool-chain allows also to generate word-level MC problems in other target languages to experiment with different verification back-ends. Finally, we carried out an experimental analysis on real-world Verilog verification benchmarks with registers and memories. We compare our tool-chain using different back-end verification algorithms provided by NUXMV against other related tools. The results shows that our tool-chain performs sensibly better than the considered existing approaches. *verilog2smv* and all experimental data are freely available online [8].

This paper is organized as follows. In Section II we discuss related works. In Section III we describe the architecture of our tool-chain and we provide a small example. In Section IV we show the results of our experimental evaluation. Finally, in Section V we draw conclusions, and we outline future work.

II. Related Tools

We mention some tools that can be used to convert Verilog designs into verification problems. (We omit considering tools which are no more maintained, like e.g. vl2mv [9].)

The following tools are publicly available. V3 [10] reads Verilog designs and produces word-level BTOR designs using QuteRTL [11] as a Verilog frontend. ABC [12] has its Verilog frontend, which transforms the designs into gate-level designs for verification. It cannot produce word-level MC problems. EBMC [13] takes Verilog designs with assertions and produces SMT formulas by applying BMC and/or k-induction. It can also output Boolean-level MC problem in SMV format. (EBMC is the successor of VCEGAR [14], which is no more maintained.) Yosys [7] is a freely-available synthesis tool from high-level Verilog to RTL and gate-level Verilog. A very-recent version can also produce SMT formulas representing combinatorial circuit designs. Cadence-SMV [15] can take Verilog design as input, generating Boolean-level SMV design (in its publicly-available version). AVERROES [16] is a verification tool which takes input Verilog designs and invariant

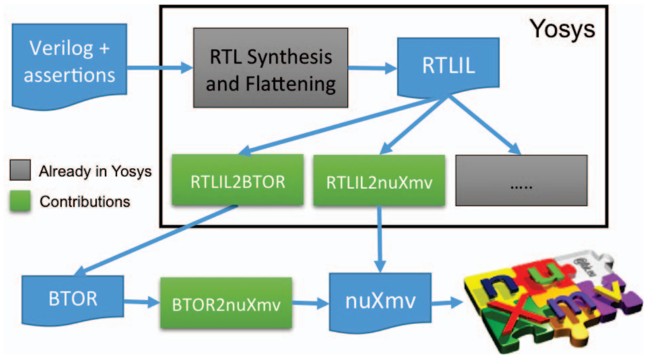

Fig. 1. *verilog2smv* architecture and verification tool-chain

```
 1  module array(input clk, output safety1);
 2    reg [7:0] counter;
 3    reg [7:0] mem [7:0];
 4
 5    always @(posedge clk) begin
 6      counter <= counter + 8'd1;
 7      mem[counter] <= mem[counter] + 8'd1;
 8    end
 9
10    assert property (!(counter > 8'd0) ||
11      mem[counter - 8'd1] == counter - 8'd1);
12
13    assign safety1 = (counter > 8'd0);
14  endmodule
```

Fig. 2. Specifying property in Verilog design

properties. With the exception of EBMC, they all cannot handle memories without abstracting or blasting them.

The following tools are not publicly available. AiPG [17] is a verification tool built on top of Boolector [18]. It takes Verilog designs and assertions as input. It can also produce BTOR designs. Reveal [19] is a tool for the verification of Verilog designs against assertions. SIXTHSENSE [20] is a verification tool by IBM which can handle Verilog designs.

III. Tool Architecture

The architecture of *verilog2smv* is depicted in Figure 1, which also shows *verilog2smv* + NUXMV tool-chain. *verilog2smv* takes in input a Verilog design, written in Verilog IEEE standard 2005 [5]. We assume the Verilog design falls in the synthesizable subset of Verilog [21]. We provide two complementary ways to specify the properties to be checked within the Verilog design. Properties can be specified within the Verilog model using the SystemVerilog `assert` statement. The `assert` statement can appear in the procedural block or in the module body of the Verilog design (see `assert` at lines 10-11 in the example in Figure 2). Properties can also be specified with a conventional notation by means of a single-bit output wire, whose name starts with `safety` (see e.g. the `safety1` wire in Figure 2). The value of this output wire needs to be driven high when the property is met.

verilog2smv is built on top of Yosys [7]. The flow of the translation is the following: We leverage on Yosys to first flatten the Verilog high-level design, and then to synthesize the RTL circuit from the result of the flattening process. The RTL circuit is stored in the Yosys internal representation RTLIL. Yosys follows the IEEE Verilog standard synthesis semantics [21]. The RTL circuit goes in to a new Yosys module that translates the input RTL circuit into a corresponding NUXMV and BTOR [22] problem. The translation preserves all the names of signals, registers, and memories of the original Verilog design. This makes it easy to interpret back in Verilog, possible counterexamples produced by the back-end model checkers. We explicitly model clock as Boolean input variable. This enables us to faithfully model flip-flops, memories, and latches, which the other translation without explicit clock can not do. The translation does not currently handle multiple clocks, multi-dimensional arrays, and combinational logic loops. Note that the translation module considers only the parts of the design that are in the cone of influence of the specified properties.

The two target verification languages, BTOR and NUXMV, allow to specify and reason about transition systems. NUXMV language allows to express transition systems using all the finite data types allowed by NUSMV [23] (Booleans, enumerative, bounded integers), plus bit-vectors, reals, integers, and (finite and infinite) arrays, with no restriction on the specification of the initial values of the variables. On the other hand, BTOR is much limited: 1) it can only deal with bit-vectors and one-dimensional arrays; 2) registers are implicitly initialized to value zero value, while arrays are uninitialized. In both cases we represent Verilog registers of width greater than one, with with corresponding variables of type bit-vectors within NUXMV and BTOR. However registers with width one are treated differently: NUXMV treats them as Boolean, while BTOR treats them as bit-vectors of width one. Memories in both cases are encoded with arrays of bit-vectors.

Notice that the conversion into BTOR is not complete in the sense that it currently supports zero-initialized registers and uninitialized memories.

Figure 3 shows the NUXMV file generated with *verilog2smv* starting from Verilog module `array` as described in Figure 2. We see that the memory `mem` is retained in the NUXMV file, as an array (declaration `mem : array word[3] of word[8]` at line 7). In the translation, we also introduce explicitly the clock and we model it as an input Boolean variable (see the `clk` input variable at line 3). Initial blocks in a Verilog design is converted into `INIT` constraints in the NUXMV file. In this example, since there is no initial block, the `INIT` constraint is simply the constant `TRUE`. The assignments to registers and memories are translated into `TRANS` constraint. (For details about the NUXMV syntax, we refer the reader the NUXMV user-manual [4].) Properties are simply translated into corresponding `INVARSPEC`. The `assert` command in Figure 2 is translated into the first `INVARSPEC`, while the property corresponding to wire `satisfy1` is encoded into the second and last `INVARSPEC`.

```
MODULE main
IVAR
"clk" : boolean;

VAR
"counter" : word[8];
mem : array word[3] of word[8];

DEFINE
__expr1 := resize(0ub8_11111111, 1);
__expr2 := bool(__expr1);
__expr3 := "counter"[2:0];
__expr4 := READ(mem, __expr3);
__expr5 := (__expr4 + 0ub8_00000001);
__expr6 := ("clk");
__expr7 := (__expr6 & __expr2);
__expr8 := WRITE(mem, __expr3, __expr5);
__expr9 := ("counter" + 0ub8_00000001);
__expr10 := ("clk");
__expr11 := (__expr10 ? __expr9 : "counter");
__expr12 := next("counter") = __expr11;
__expr13 := (case __expr7: __expr8;
                  TRUE: mem; esac);
__expr14 := next(mem) = __expr13;
.
.
__expr24 := (__expr18 | __expr23);
__expr25 := bool(0ub1_1);
__expr26 := (__expr25 -> __expr24);
__expr27 := ("counter" > 0ub8_00000000);

INIT TRUE;
TRANS __expr15;
INVARSPEC __expr26;
INVARSPEC __expr27;
```

Fig. 3. NUXMV file for the Verilog design shown in Figure 2.

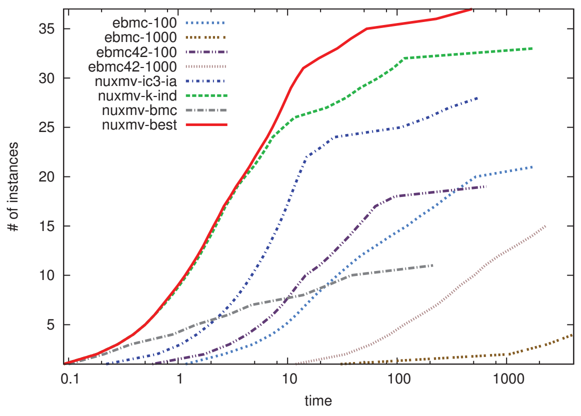

Fig. 4. Accumulated Plot for benchmarks with memories and registers

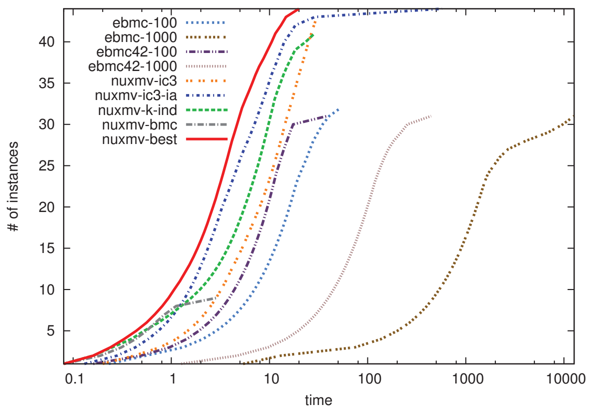

Fig. 5. Accumulated Plot for benchmarks with registers only

IV. EXPERIMENTAL EVALUATION

In this section we describe an experimental evaluation we carried out to show the effectiveness of the *verilog2smv* + NUXMV tool-chain.

A. Setup of the experimental evaluation

We have considered a set of benchmark problems, Verilog files and invariant properties files, from the VIS [24] and VCEGAR [25] benchmark suites. The collection includes 42 problems with memories and registers (40 from VIS and 2 from VCEGAR) and 44 problems with registers only (14 from VIS and 29 from VCEGAR), totalling 86 problems.

We have compared our tool-chain against v3, AVERROES, and EBMC on the collected benchmarks. We have also used the very-recently released version 4.2 of EBMC. Unfortunately, we can show results against EBMC only because v3 and AVERROES either were not able to process most of the Verilog designs we collected, or they crashed without producing results. One remark is in order: Since most of the memories and registers benchmarks have initialized memories, this prevents us from using the BTOR generator of our tool and from comparing against model checkers that can take BTOR as input. For benchmarks that contain both memories and registers, we have considered the following verification algorithms offered by NUXMV: a) `nuxmv-k-ind`, SMT-based *k-induction/BMC*; b) `nuxmv-bmc`, SMT-based *BMC*; c) `nuxmv-ic3-ia`, SMT-based *IC3 with implicit abstraction* [1]. For registers-only benchmarks, besides a), b), and c), we have also considered d) `nuxmv-ic3` SAT-based IC3 [26]. For each NUXMV configuration, the bound is 1000. For EBMC we use the the following configurations: e) `ebmc-100`, *k-induction/BMC* with bound 100; f) `ebmc-1000`, *k-induction/BMC* with bound 1000; g) `ebmc42-100`, EMBC 4.2 using *k-induction/BMC* with bound 100; h) `ebmc42-1000`, EBMC 4.2 using *k-induction/BMC* with bound 1000. We have run our experiments on a cluster of 64-bit Linux machines with 2.7 GHz Intel Xeon X5650 CPUs, with memory limit of 4GB and time limit of 3600 seconds.

B. Results

The results of the experiments are shown in form of accumulated plots, where on the x-axis we have the accumulated solving time and on the y-axis we have the number of solved instances. Figure 4 shows the results for the 42 benchmarks with memories and registers; Figure 5 shows the results for the 44 benchmarks with registers only. In the plots, we also show `nuxmv-best` which is virtual best configuration for NUXMV.

C. Discussion

The results clearly show that our tool-chain is performing better than EBMC, on the selected benchmarks. In particular, we see that on the benchmarks with memories and registers, `nuxmv-k-ind` has solved 33 benchmarks (22 safe, 11 unsafe), instead `nuxmc-ic3-ia` has been able to solve 28 benchmarks (25 safe, 3 unsafe). `ebmc42-100` has solved only 19 benchmarks (9 safe, 10 unsafe). We notice that `nuxmv-ic3-ia` has solved more safe instances than `nuxmv-k-ind`, probably since the former uses abstraction. However `nuxmv-ic3-ia` has solved less unsafe instances than `nuxmv-k-ind`, probably due to the limited support for array abstraction refinement in NUXMV. We skip the discussion for older version of EBMC, i.e. `ebmc-*` configurations.

For the benchmarks with registers only, `nuxmv-ic3-ia`, `nuxmv-ic3`, and `nuxmc-k-ind` are much closer. Indeed the first has solved all the 44 benchmarks (35 safe, 9 unsafe), the second has solved 43 benchmark (34 safe, 9 unsafe), and the last one has solved 41 benchmarks (32 safe, 9 unsafe). Interestingly `nuxmv-ic3-ia`, which is an SMT-based IC3, has shown better performance than SAT-based IC3 `nuxmv-ic3`. `ebmc42-100` has only solved 31 benchmarks (26 safe, 5 unsafe).

We conjecture that EBMC is slower than the nuXmv-based tools because it is not exploiting incrementality while solving.

Importantly, whenever terminating, on the latter benchmarks all tools always agree on the result, whilst on the former ones all `nuxmv-*` and `ebmc-*` always agree, whilst `ebmc42-*` disagrees with both `nuxmv-*` and `ebmc-*` on two instances.

We remark that, *verilog2smv* has the ability to generate benchmarks in different formats. In particular, It can generate Boolean-level benchmarks in AIGER [27] format either using *synthebtor* tool, from the Boolector distribution, or using the converter in AIGER format built-in in NUXMV. This will enable for generating benchmarks and experimenting with any model checker from the hardware model checking competition. Once we have extended the BTOR generator with explicit initialization (see future works in Section V), we will do an evaluation with other model checkers. It can also generate VMT [28] models in VMT format using the NUXMV model checker. VMT is an extension of SMTLIB [29] format to represent theory level transition systems.

V. Conclusion and Future Work

We have presented *verilog2smv*, which converts Verilog designs with assertions into RTL MC problems, while fully treating memories. We believe that *verilog2smv* will be helpful in pushing research on the verification of RTL designs with large memories. We also presented a Verilog RTL verification tool-chain using *verilog2smv* and NUXMV model checker, and have shown effectiveness of the tool-chain by evaluating on a collection of Verilog benchmarks.

There are many possible directions, where *verilog2smv* can be extended. One possible direction is to extend the support to full SystemVerilog properties. Another direction is to extend the BTOR generator to encode initialization of registers and memories in the transition constraint. A rather ambitious future direction would be extending *verilog2smv* to convert Verilog designs into threaded software program. This conversion will help in doing verification of high-level Verilog designs. With respect to the Verilog verification tool-chain, there is a possibility to improve the connection between *verilog2smv* and NUXMV, e.g. translating counterexample into VCD format.

References

[1] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction,” in *TACAS*, 2014.
[2] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, “The nuXmv symbolic model checker,” in *CAV*. Springer, 2014.
[3] R. Brummayer, A. Biere, and F. Lonsing, “BTOR: bit-precise modelling of word-level problems for model checking,” in *BPR*. ACM, 2008.
[4] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, “nuXmv 1.0 user manual,” 2014.
[5] “IEEE Standard for Verilog Hardware Description Language,” *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.
[6] D. Bustan, D. Korchemny, E. Seligman, and J. Yang, “SystemVerilog Assertions: Past, present, and future SVA standardization experience,” *IEEE Design & Test of Computers*, vol. 2, no. 29, 2012.
[7] C. Wolf, “Yosys open synthesis suite,” http://www.clifford.at/yosys/.
[8] https://es.fbk.eu/tools/verilog2smv.
[9] T. Villa, G. Swamy, T. Shiple, A. Aziz, R. Brayton, S. Edwards, G. Hachtel, S. Khatri, and Y. Kukimoto, “VIS users manual,” *Electronics Research Laboratory, University of Colorado at Boulder*, 1996.
[10] C.-Y. Wu, C.-A. Wu, and C.-Y. R. Huang, “V3,” http://dvlab.ee.ntu.edu.tw/ publication/V3/index.html.
[11] H.-H. Yeh, C.-Y. Wu, and C.-Y. R. Huang, “Qutertl: towards an open source framework for rtl design synthesis and verification,” in *TACAS*. Springer, 2012.
[12] R. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *CAV*. Springer, 2010.
[13] D. Kroening and M. Purandare, “EBMC,” http://www.cprover.org/ebmc/.
[14] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, “VCEGAR: verilog counterexample guided abstraction refinement,” in *TACAS*, 2007.
[15] K. McMillan, “Cadence smv,” *Cadence Berkeley Labs, CA*, 2000.
[16] S. Lee and K. A. Sakallah, “Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction,” in *CAV*. Springer, 2014.
[17] Y. Sugiura, “AiPG RTL property checker,” http://www.revsonic.com/e/business/lsisolution/eda/aipg/.
[18] R. Brummayer and A. Biere, “Boolector: An efficient smt solver for bit-vectors and arrays,” in *TACAS*, ser. TACAS ’09. Springer, 2009.
[19] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, “Reveal: A formal verification tool for Verilog designs,” in *LPAR*. Springer, 2008.
[20] J. R. Baumgartner, “Semi-formal verification at IBM,” in *HLDVT*. IEEE, 2006.
[21] “Verilog Register Transfer Level Synthesis,” *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1*, 2005.
[22] A. Irfan and C. Wolf, “Yosys App-Note 012: Converting Verilog to BTOR,” http://www.clifford.at/yosys/files/yosys_appnote_012_verilog_to_btor.pdf.
[23] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “Nusmv 2: An opensource tool for symbolic model checking,” in *CAV*. Springer, 2002.
[24] F. Somenzi, “VIS Verification Benchmarks,” ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.3.tar.gz.
[25] “VCEGAR Verification Benchmarks,” http://www.cprover.org/hardware/benchmarks/vcegar-benchmarks.tgz.
[26] A. R. Bradley, “SAT-based Model Checking Without Unrolling,” in *VMCAI*. Springer, 2011.
[27] http://fmv.jku.at/aiger/.
[28] https://es.fbk.eu/tools/nuxmv/index.php?n=Languages.VMT.
[29] C. Barrett, A. Stump, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” http://www.SMT-LIB.org, 2010.