2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Retrascope: Open-Source Model Checker for HDL Descriptions

Alexander Kamkin, Mikhail Lebedev, Sergey Smolov
Ivannikov Institute for System Programming of Russian Academy of Sciences, Moscow, Russia
{kamkin, lebedev, smolov}@ispras.ru

***Abstract* — Open source penetrates the microelectronics design, making hardware development more accessible. To leverage this effect, one needs open CAD tools, including formal verification facilities. This paper describes Retrascope, an open-source model checker that verifies Verilog descriptions against SVA properties. The primary application domain of the toolkit is functional verification of Verilog or VHDL modules. The toolkit allows analyzing of HDL descriptions, reconstructing and visualization of the underlying models and using the derived models for test generation and property checking. Retrascope is organized as an extendible framework with the ability to add new types of models as well as engines for their processing. The paper describes the main flow for SVA property checking and gives an experimental comparison with other model checkers on several benchmarks and industrial case study.**

Keywords* — *hardware description language, functional verification, test generation, static analysis, control flow graph, guarded actions decision diagram, model checking, benchmarks

I. Introduction

Microelectronic design – once carried out solely by semiconductor giants – becomes available to small R&D teams. Chip manufacturing costs a reasonable amount of money when using FPGA or multi-project wafer services. The primary bottleneck that holds back the development of many promising start-ups is CAD licenses (they might be expensive). Indeed, there are open tools: simulators (Icarus [1], Verilator [2], etc.), synthesizers (Yosys [3], ABC [4], etc.) and verifiers (SymbiYosys [5], EBMC [6], etc.). However, their capabilities are inferior to the commercial counterparts from Cadence, Synopsys and other companies. In the software world, the situation seems to be different: basic things, like operating systems, compilers, and IDE, are free (not all, but many of them). Looking at the RISC-V and MIPS Open initiatives, we see how open source is penetrating the world of hardware, and it is obvious that open tools are highly demanded and can stimulate hardware design industry and education.

This work has two contributions. First, we have developed an open-source model checker, called Retrascope, that allows verifying Verilog designs against temporal properties written as SystemVerilog Assertions (SVA). It works as follows: source code in Verilog and properties in SVA are parsed, elaborated, and translated respectively to SMV and PSL; after that, NuSMV [7] (or nuXmv [8]) tool is invoked to do the model checking. Second, we have conducted a series of experiments comparing Retrascope with its analogues, including EBMC (former VCEGAR), SymbiYosys, and Verilog2SMV [9]. The study has been based on open benchmarks such as Texas-97 [10], VCEGAR, and Verilog2SMV (some of the names repeat the names of the tools they were intended to test).

The rest of the paper is organized as follows. Section II overviews the related work. Section III describes the developed tool and the underlying model checking based approach. Section IV contains the tool evaluation. Section V concludes the paper.

II. Related Work

Many open tools for functional verification of hardware designs appear today. This trend is highly stimulated by the development of open architectures, like RISC-V. This section is dedicated to open tools for functional verification of HDL descriptions.

It is not a new idea to create a formal verification tool for hardware designs. There are a number of approaches that such tools are based on. The traditional one is the *synthesis* of a Verilog description into a *netlist*

description and its subsequent checking using automated tools like ABC [11]. There is a plenty of tools aimed at netlist formal checking, some of them take part in HWMCC (HardWare Model Checker Competition) [12]. Unfortunately, most of HWMCC participant tools cannot be used for Verilog checking, because they do not fully support this language (the standard input format for HWMCC is AIGER [13] that is an *and-inverter graph* similar to netlist representations).

To use a netlist checker for Verilog description a synthesizing tool is needed. In [14] a chain of *QuteRTL* and *V3* tools is proposed for hardware verification. QuteRTL tool takes a Verilog description as input, builds an internal representation and translates it into a netlist (AIGER or BTOR [15]). Then V3 applies model checking techniques (Bounded Model Checking, k-induction) to the netlist and prints counterexamples in a VCD format.

Today the promising open-source framework for Verilog synthesis is Yosys. The front-end driver for Yosys called SymbiYosys provides flows for several formal tasks: bounded/unbounded checking for a SVA subset, counterexample/testbench generation, etc. SymbiYosys is used for property checking of RISC-V cores [16].

Another approach to formal verification of Verilog descriptions is an *automated theorem proving*. The property checking process comes in iterative and semi-automated way: a user passes additional properties to a *prover* tool to speed up the property proving process. In [17] a method of Verilog translation into a format of a *UCLID* [18] prover tool was described. The method was applied to several non-industrial scale benchmarks.

Some hardware verification tools are based on *software verification* related components. The EBMC tool that performs bounded model checking technique on Verilog descriptions is based on the *CBMC* [19] tool for C/C++ programs. In [20] a *CV* tool for VHDL code verification is described. It uses CBMC too. In [21] a *v2c* translator is described, that builds C programs from Verilog modules. It is said that v2c can be combined with CBMC.

Finally, some tools use *external model checkers*. One of the first was SMV [22], and it was applied to netlist verification [23]. SMV is a core for such model checkers as NuSMV and nuXmv. In [24] a Verilog2SMV tool is described. It translates Verilog descriptions into a nuXmv format. The Verilog2SMV uses Yosys for Verilog parsing and elaboration and is able to check SystemVerilog properties. The paper [25] describes the extension of the Verilog2SMV tool, called *Ver2SMV*. It takes a Verilog description and a VCD random test trace as inputs, then generates a number of assertions with the help of mining techniques that are implemented in a GoldMine [26] tool, and, finally, runs Verilog2SMV to translate the data into the nuXmv format.

In [27] an EFSM based approach is proposed for semi-formal verification of HDL descriptions. The idea is to extract an EFSM model from the hardware description, perform state traversal and check states for reachability with the help of the NuSMV.

To conclude this chapter, there are a number of open tools for formal property checking for HDL descriptions. Some of them seem to be unsupported now (QuteRTL, V3, Ver2SMV), other are in active use in academy and industry. Unfortunately, there is a lack of comparison researches in this field. We have made experiments (see Section IV) that have shown that tools have issues both in Verilog support and in property checking. There is still a free place for a new open tool for formal verification of HDL descriptions.

III. PROPOSED APPROACH

A. *Retrascope toolkit*

In this paper we propose a new toolkit for functional unit-level verification of HDL descriptions called Retrascope (*Reverse Engineering*, visualization and *TRAnsformation* of digital hardware designs). It is an open-source framework that provides flows for static analysis, visualization and functional test generation of VHDL/Verilog modules. Retrascope is written in Java and uses auxiliary libraries: Fortress [28] (SMT solver interaction), JUNG [29] (graph representation and analysis), zamiaCAD [30] (preliminary elaboration of VHDL designs). The toolkit has both command line interface and an Eclipse [31] based GUI called Retrascope IDE. The last one provides user with all the features of the toolkit, including HDL navigation and test generation.

Retrascope consists of three subsystems: 1) data representations (*entities*); 2) active components (*engines*); 3) *library*. Entities are either internal representations (*models*) of HDL descriptions or *artifacts*. Models are based on the HDL code and can either be extracted from it or generated through a sequence of transformations. Artifacts are built from models or described by the user (like specifications). Retrascope entities include the following: 1) Control Flow Graph (CFG); 2) Guarded Actions Decision Diagram (GADD); 4) Extended Finite State Machine (EFSM); 5) input signals sequence (Test); 6) specification to be checked (Property).

Engines are active components that implement the logic of entities' analysis, transformation, etc. Every engine belongs to the one of several categories: 1) *Transformer* – transforms a model of one type to another type; 2) *Launcher* – runs external tool on an entity; 3) *Generator* – creates an artifact from an entity; 4) *Simulator* – performs model's execution; 5) *Printer* – translates an entity into a graphical or text format; 6) *Parser* – creates an entity from a text description. On Fig. 1 the main flow of the Retrascope toolkit and several used engines are shown. Engines are shown as black round nodes, models are shown as rectangles and artifacts are shown as elliptic nodes. Data flows are shown as arrows, file reading/writing interactions are shown as dotted arrows. Model checker is not a part of Retrascope, so there is a possibility to change it.

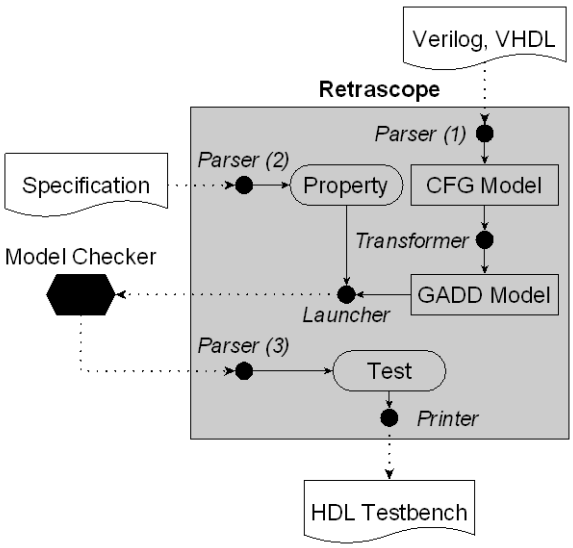

Figure 1. Hardware model checking flow in Retrascope toolkit

The engines cannot interact with each other. Instead, they can be combined in sequences, so-called *toolchains*. Two components can be combined if one of them returns an entity of some type and the other takes this entity as input. The library part of the Retrascope framework provides facilities to automatically build toolchains. Target types of engines and entities are user-defined. On Fig. 1 the following toolchain is shown: "*Parser* (*1*), *Parser* (*2*), *Transformer*, *Launcher*, *Parser* (*3*), *Printer*". The user specifies the target entity and main engines and the toolkit constructs the required toolchain and applies it to input data.

B. Model generation

This subsection starts the description of the main flow of the Retrascope toolkit. The flow includes the following steps: 1) *preliminary processing* of the target HDL module and SVA specifications; 2) *control flow graph* (CFG) building; 3) transformation of the CFG to a *guarded actions decision diagram* (GADD); 4) translation of the GADD into a formal model; 5) translation of the formal model into the SMV format of the model checker; 6) SMV model checking. All the steps are made automatically by the Retrascope.

At the first step HDL description is translated into the internal representation — an *abstract syntax tree* (AST) for Verilog modules or an *instance graph* (Instantiation Graph, IG) for VHDL modules. Different tools are used for preliminary processing (we use our own translator for Verilog modules and zamiaCAD framework for VHDL modules). Retrascope also translates user-written SVA specifications into the PSL format.

At the second step the CFG representation is constructed. CFG is a directed graph. Nodes of the graph contain HDL operators, edges of the graph mean control flows. CFG model is constructed by the algorithm that is a classical compiler-like traversal of either the AST or the IG [32].

At the third step the GADD model is built. GADD is a transformation of CFG. The idea is to split all the paths of the CFG model into fragments by branch (*switch* or *if*) operators in such way that every fragment does not contain an internal branch operator. Fragments are called *guarded actions* (GA). A guarded action is a pair $\gamma \to \delta$, where γ is a *guard* and δ is an *action*. A guard is a Boolean condition of a branch to be activated; action is a basic block (sequence of HDL assignments). In Fig. 2 examples of CFG and GA are shown.

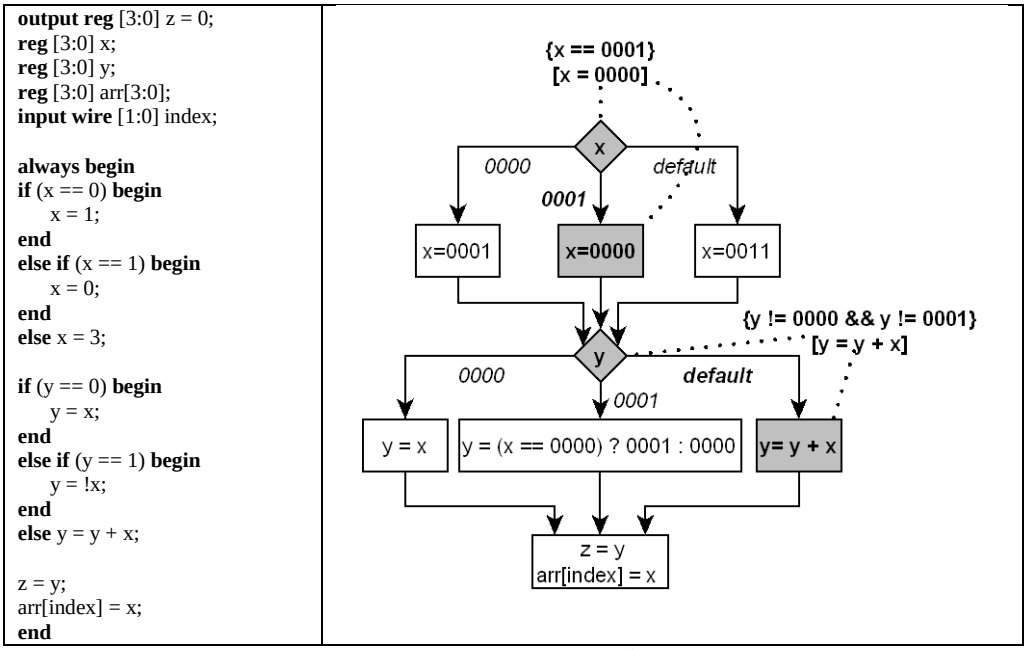

Figure 2. CFG model and guarded actions of Verilog process

On the left side a fragment of Verilog module is described. On the right side the CFG is shown. Rhombic vertices denote branch operators; rectangular vertices denote basic blocks. Continuous edges correspond to control flows; edges that are marked with italicized values are branches. Two GAs are shown in Fig. 2. Guards are shown in braces; actions are shown in brackets below. Both GAs are connected to the parts of the CFG they are related to by dashed line arrows (CFG parts are highlighted with bold font and grey color). One of the GAs is related to a so-called *default* branch: passing to it means that conditions to other branches are *false*.

GA-based model is quite effective for CFG path storage. As an example, imagine a CFG consisting of a sequence of m sequential non-nested branch operators. Let the number of branches in each statement be n, then the total number of execution paths is n^m, whereas the GAs number is $O(m * n)$.

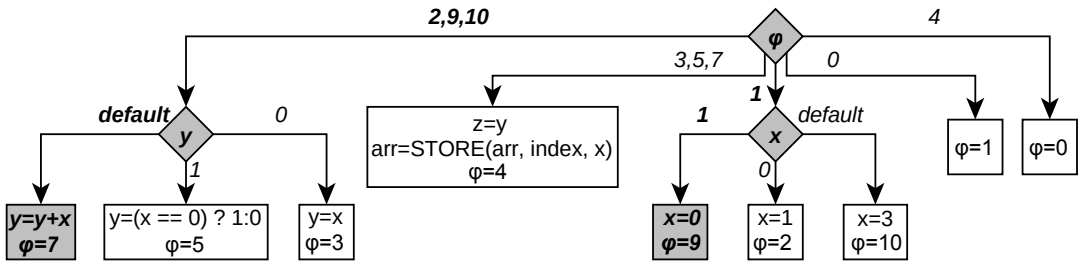

Figure 3. GADD for Verilog module

GAs are *atomic*, i.e. they do not contain any information about their execution order in a CFG. This problem can be solved by introducing a *phase* (*phase variable*). Phase is an internal variable that retains its value within a

single GA, but changes it when control is transferred to another GA. The initial value of the phase at the entry point of the process is 0. Phase is again assigned 0 at the exit point of the process. The GADD itself is a modified CFG model. At the top level of each process there is a phase branch operator. Each branch of this operator contains exactly one GA with the new value assignment to the phase. Fig. 3 shows the GADD for the example above (selected GA are grey-colored). When a single GA has been executed, a new value of a phase is assigned and the next GA is selected deterministically. At this step the GADD model is also transformed to a Static Single Assignment (SSA) form [33]. The constructed GADD is equivalent to the original CFG – the GAs appear strictly in the same order as in the original graph.

C. *Invariant construction and SMV translation*

At the next step, the formal model is constructed. The main part of the formal model is a process representation as a single logical formula that is called the process *invariant*. The invariant is a conjunction of the process GAs in the SSA form. To build the SSA representation, each variable is labeled by the corresponding *version* value.

The variables that are *used* in a GA are labeled by the previous value of the phase. The variables that are *defined* in an action are labeled by the corresponding phase value contained in this action. The phase itself is not labeled by the version value and not used in the invariant. To represent the guard, a special variable *guard* is introduced. Every guard variable value is linked (with the help of conjunction) to the previous phase guard variables to form the execution path. Assignments are represented as equations. Thus, for each GA a set of equations is created. At this point, values of some of the variable versions are unknown because either they are not defined on the previous phase or there is more than one previous phase (merge point). Backward traversal of the GADD is used to find their definitions. When all the variables are defined, equations are united into the invariant:

$$guard^{(k)} == condition^{(i,j)} \& \; y^{(k)} == f\left(x^{(i,j)}\right) \& x^{(i,j)} == guard^{(s)} \; ? \; x^{(s)} : guard^{(t)} \; ? \; x^{(t)} : \ldots : x \; \& \ldots$$

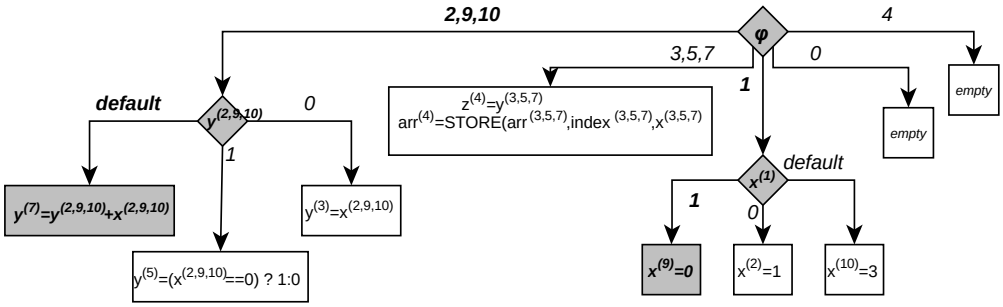

Figure 4. SSA version labeled GAs

Fig. 4 shows the SSA labeling of the variables from the previous example.

At the fifth stage the formal model is translated into the SMV language. The translation is straightforward unless the *next* transition for each non-input variable should be added to the model. This construct determines the value of the variable in the next state of the model. In our representation, this value equals the value of the last phase version of the variable.

D. *Properties*

Properties used for model checking can be manually written or automatically generated and be represented using temporal logics (CTL, LTL), PSL or as logical invariants in SMV format. In Retrascope there are two types of the automatically generated properties: properties representing the potential *data access conflicts* and properties that cover the Extended Finite State Machine (EFSM) transitions. User-written SVA properties are partly supported. They are automatically translated into PSL.

When building the data conflict properties, we consider the following types of potential conflicts: 1) *read-write* (RW): one process writes the variable and the other process reads it on the same clock tick; 2) *write-write* (WW): at least two processes write the same variable on the same clock tick; 3) *write-read-write* (WRW): we assume that the variable should be read between two writes; 4) *undefined* (UNDEF): the variable is read before it has been written at least once. The generated properties are based on usage and definition conditions of a variable. Conditions are extracted during the formal model construction. The properties are represented as the LTL formulas and state that the abovementioned conflicts never happen.

The Extended Finite State Machine (EFSM) is built from the GADD. Its extraction and property generation method is described in [34]. Shortly, for EFSM transitions to be enabled, the enabling condition should be satisfied. Enabling conditions are translated into CTL or LTL properties. Properties state that the transitions are never enabled. EFSM-based properties are used for *dead code* detection.

E. Model checking

For model checking we use nuXmv. There is a Retrascope engine that parses the model checker's output and extracts the counterexample data from it (if any). These data are translated into the *Test* entity representing the sequence of input signal values. The other engine generates an HDL test environment.

Usually bounded model checking is used because of the big model sizes, especially for industrial-scale designs. Using unbounded model checking for these designs will result in the state space explosion while building the BDD representation by the model checker. On the other hand, unbounded model checking shows better results on relatively small designs. It is up to the Retrascope user to choose the model checking method.

IV. EVALUATION

This chapter is dedicated to an experimental evaluation of the Retrascope toolkit. Another contribution is the experimental comparison with the other tools for Verilog checking: EBMC 4.4, SymbiYosys, Verilog2SMV 1.1.2. These tools are briefly described in Section II. The chapter is aimed at resolving the following questions:

- How do these model checkers support Verilog?
- How these tools are capable to check properties from open benchmarks and industrial-scale designs?

Several popular benchmarks have been chosen for experiments: Texas-97, VCEGAR, Verilog2SMV. Test machine is Intel Core i7 3.4 GHz, 8 Gb RAM, Ubuntu Linux OS.

At first, Verilog language support has been analyzed. ModelSim 10.5b Intel FPGA Starter Edition [35] has been chosen as a reference compiler. The experiment has included two stages. At the first stage original Verilog designs from benchmarks have been used. At the second stage all the compilation errors reported by ModelSim have been fixed and the updated modules have been processed again. The results are shown in the Table I. The benchmark-named columns contain the numbers of the Verilog files for which the tools have demonstrated one of the four scenarios: 1) OK (no errors); 2) ERR (an error is detected in the erroneous file); 3) FAIL (an error is detected in the *correct* file); 4) CRASH (the tool has crashed unexpectedly). For every cell the first number is related to the second (fixed Verilog) stage, the number in braces is related to the first (original Verilog) stage.

Table I. Verilog support evaluation

Tool	**Texas-97**				**VCEGAR**				**Verilog2SMV**			
	OK	**ERR**	**FAIL**	**CRASH**	**OK**	**ERR**	**FAIL**	**CRASH**	**OK**	**ERR**	**FAIL**	**CRASH**
EBMC	12 (9)	0 (48)	54 (20)	12 (1)	37 (35)	0 (1)	0 (0)	0 (1)	92 (3)	0 (3)	4 (90)	1 (0)
SymbiYosys	8 (6)	0 (47)	63 (18)	7 (7)	22 (20)	0 (4)	15 (13)	0 (0)	96 (96)	0 (0)	0 (0)	0 (0)
Verilog2SMV	9 (8)	0 (47)	65 (19)	4 (4)	21 (19)	0 (4)	15 (13)	1 (1)	96 (96)	0 (0)	0 (0)	0 (0)
Retrascope	21 (20)	0 (46)	57 (12)	0 (0)	36 (35)	0 (1)	1 (1)	0 (0)	93 (93)	0 (0)	3 (3)	0 (0)

Experiments have shown that the tools do not support the following Verilog constructions: 1) two or more complement declarations of the same signal (SymbiYosys, Verilog2SMV); 2) assigning inputs to registers or

outputs (EBMC); 3) *task* without parameters (EBMC); 4) different types of assignment sides (EBMC); 5) uninitialized variables (SymbiYosys, Verilog2SMV).

SymbiYosys and Verilog2SMV are based on Yosys so they have shown similar results on most modules. Usually, tools have shown the best results (most OK, less FAIL and CRASH) on their own benchmarks: EBMC on VCEGAR, Yosys based tools on Verilog2SMV. Retrascope has shown the best results on Texas-97; also it looks good on VCEGAR and Verilog2SMV. All the tools have found most of the errors and returned most of the failures in the Texas-97 benchmarks. This test suite seems to be immature and should be improved for future researches.

At the second stage of the experimental evaluation, property checking abilities of the tools have been analyzed. In benchmarks SVA properties are represented as invariants like ***assert property*** *expr*, where *expr* is a condition. EMBC does not support SVA so we have made a separate test suite for it, where properties are represented by *always* blocks with *assert* statements inside. The time limit for model checking of a separate module has been set to 1 hour. All the tools have been executed in bounded model checking mode, the bound has been set to 100. The Table II contains property checking results. The following notation for results is used: 1) FALSE – the property is *false* and the counterexample is generated; 2) TRUE – property is *true* or the model checking bound has been reached; 3) ERR – an error is returned for the erroneous Verilog file; 4) FAIL – an error is returned for the correct Verilog file; 4) CRASH – unexpected crash; 5) TIMEOUT – timeout expiration.

Table II. Property checking evaluation

Tool	**FALSE**	**TRUE**	**ERR**	**FAIL**	**CRASH**	**TIMEOUT**
EBMC	40	105	0	6	0	2
SymbiYosys	29	98	0	11	3	6
Verilog2SMV	27	90	0	13	3	14
Retrascope	44	81	4	1	5	15

As it can be seen from the Table II, Retrascope advantages are: 1) most powerful capabilities to find Verilog errors (“ERR” column); 2) wide support of Verilog syntax (“FAIL” column). Also Retrascope has found the maximum number of counterexamples (“FALSE” column). On the contrary, the model checker used (nuXmv) gives the maximum number of timeout expirations (and so does NuSMV for Verilog2SMV tool). In this field EBMC and SymbiYosys look more stable. Note, that total property amounts for tools are different, because Yosys-based tools are unable to check properties that are described in the same module in a separate way.

This paper also discusses the dispatcher module from the Intel Quartus Prime [36] design environment as an industrial case study. This module contains 2200 lines of code and consists of instances of 8 sub-modules. For experiments *dispatcher_checker* module that contains SystemVerilog properties has been used. None of the tools except Retrascope has been able to elaborate the *dispatcher* code without errors. The following limitations have been detected:

- EBMC: does not support *defparam* construction, local values for parameters;
- SymbiYosys: parameter redefinition in sub-modules behaves incorrectly;
- Verilog2SMV: sub-module instantiation and parameter redefinition behaves incorrectly;
- Because of the huge *dispatcher_checker* module size, NuSMV can perform model checking of the Retrascope-generated model only up to a small bound value.

V. CONCLUSION

We are witnessing the revolution in hardware design: the cost of production is reduced; open specifications and source code appear. Already, many labs around the world are developing their devices, which they could not think of earlier. However, the revolution is impossible without free CAD tools: simulators, synthesizers, verifiers, etc. In this paper, we have described Retrascope, an open-source model checker, and the results of its experimental analysis. The tool is comparable with other freely distributed solutions and in some ways surpasses

them. Directions of further research include Retrascope improvement, development of more adequate benchmarks, and comparison of open-source tools to the commercial ones.

REFERENCES

[1] Icarus Verilog. Verilog simulation and synthesis tool. http://iverilog.icarus.com

[2] Verilator. HDL simulation tool. https://www.veripool.org/projects/verilator/wiki

[3] Clifford Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys

[4] ABC. A System for Sequential Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc

[5] SymbiYosys. Front-end for Yosys-based formal verification flows. https://github.com/YosysHQ/SymbiYosys

[6] EBMC. The Enhanced Bounded Model Checker. http://www.cprover.org/ebmc

[7] NuSMV. A new symbolic model checker. http://nusmv.fbk.eu

[8] nuXmv. The nuXmv model checker. https://nuxmv.fbk.eu

[9] Verilog2SMV. https://es-static.fbk.eu/tools/verilog2smv

[10] Texas-97 Verification Benchmarks. https://ptolemy.berkeley.edu/projects/embedded/research/vis/texas-97

[11] R. Brayton, A. Mishchenko, "ABC: An academic industrial-strength verification tool", Proc. of the 22nd International Conference on Computer Aided Verification (CAV), pp. 24-40, 2010.

[12] A. Biere, T. van Dijk, K. Heljanko, "Hardware Model Checking Competition 2017", Proc. of the 17th International Conference on Formal Methods in Computer Aided Design (FMCAD), pp. 9-9, 2017.

[13] A. Biere, K. Heljanko, S. Wieringa, "AIGER 1.9 And Beyond", Technical Report 11/2, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, July 2011.

[14] H.-H. Yeh, C.-Y. Wu, C.-Y. Huang, "QuteRTL: towards an open source framework for RTL design synthesis and verification", Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 377-391, 2012.

[15] A. Niemetz, M. Preiner, A. Biere, "Boolector 2.0 system description", Journal on Satisfiability, Boolean Modeling and Computation, Vol. 9, pp. 53-58, 2015.

[16] RISC-V Formal Verification Frameworl. https://github.com/SymbioticEDA/riscv-formal

[17] Z.S. Andraus, K.A. Sakallah, "Automatic abstraction and verification of Verilog models", Proc. of the 41th Design Automation Conference (DAC),pp. 218-223, 2004.

[18] UCLID tool. https://people.eecs.berkeley.edu/~sseshia/research/uclid.html

[19] CBMC. Bounded Model Checker for C and C++ programs. https://www.cprover.org/cbmc

[20] D. Deharbe, S. Shankar, E.M. Clarke, "Model checking VHDL with CV", Proc. of the 2ndInternational Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 508-514, 1998.

[21] R. Mukherjee, D. Kroening, T. Melham, "Hardware verification using software analyzers", Proc. of the IEEE Computer Society Annual Symposium on VLSI, pp. 7-12, 2015.

[22] The SMV System. https://www.cs.cmu.edu/~modelcheck/smv.html

[23] J. BurchJ., E. Clarke, K. McMillan, D. Dill,"Sequential circuit verification using symbolic model checking", 27th ACM/IEEE Design Automation Conference, 1990.

[24] A. Irfan,A. Cimatti, A. Griggio,M. Roveri,R. Sebastiani, "Verilog2SMV: A tool for word-level verification", Proc. of the Design, Automation& Test in Europe Conference & Exhibition (DATE), pp. 1156-1159, 2016.

[25] M. Minhas,O. Hasan, K. Saghar, "Ver2Smv - A Tool for Automatic Verilog to SMV Translation for Verifying Digital Circuits", Proc. of the International Conference on Engineering & Emerging Technologies (ICEET), 2018.

[26] GoldMine: An automatic assertion generation tool. https://sites.google.com/view/goldmine-illinois/home

[27] G. Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, M. Roveri,"Semi-formal functional verification by EFSM traversing via NuSMV", IEEE International High Level Design Validation and Test Workshop (HLDVT), pp. 58-65, 2010.

[28] Fortress. FORmula Translation, Evaluation and Symbolic Simplification library. https://forge.ispras.ru/projects/solver-api

[29] JUNG. Java Universal Network/Graph framework. http://jung.sourceforge.net

[30] zamiaCAD. Open source platform for advanced hardware design. http://zamiacad.sourceforge.net/web

[31] Eclipse IDE. https://www.eclipse.org

[32] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers: principles, techniques, and tools", 1986, 796 p.

[33] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", ACM Transactions on Programming Languages and Systems, 13(4), pp. 451-490, 1991.

[34] S. Smolov, A. Kamkin, "A method of extended finite state machines construction from HDL descriptions based on static analysis of source code", St. Petersburg State Polyt. Univ. Journal. Computer Science. Telecommunications, No 1 (212), pp. 60-73, 2015.

[35] ModelSim simulator. https://www.intel.ru/content/www/ru/ru/software/programmable/quartus-prime/model-sim.html.

[36] Intel Quartus Prime Software Suite. https://www.intel.ru/content/www/ru/ru/software/programmable/quartus-prime/overview.html