DOI: 10.15514/ISPRAS-2025-37(5)-10

Application of SVAN Static Analysis Tool on Open RTL Benchmarks

S.M. Panova, ORCID: 0009-0008-5106-0915 <panova@ispras.ru>
S.A. Smolov, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>
M.M. Volkova, ORCID: 0009-0009-8324-7562 <volchonok03@bk.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Plekhanov Russian University of Economics,
36, Stremyanny lane, Moscow, 115004, Russia.

Abstract. The article presents an experimental evaluation of SVAN, a static analysis tool designed for functional verification of RTL models written in Verilog and SystemVerilog. The research addresses the growing need for reliable domestic EDA tools, particularly in light of restrictions on proprietary solutions. SVAN's architecture integrates formal methods and heuristic approaches to detect a wide range of errors, including syntactic issues, coding style violations, logical inconsistencies, and security vulnerabilities. Empirical testing on open-source hardware benchmarks demonstrates SVAN's superior effectiveness compared to Synopsys VCS and Verilator, with a 73% broader error detection spectrum and 25-23% higher error identification rate, respectively. Key advantages of SVAN include high analysis accuracy and detailed error classification. However, limitations such as reduced flexibility in handling mixed-language designs highlight areas for future improvement. The study underscores SVAN's potential as a competitive tool for static verification in electronic design automation.

Keywords: static analysis; RTL; SVAN; Verilog; SystemVerilog; functional verification; error detection; open-source benchmarks.

For citation: Panova S.M., Smolov S.A., Volkova M.M. Application of SVAN Static Analysis Tool on Open RTL Benchmarks. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 131-142. DOI: 10.15514/ISPRAS-2025-37(5)-10.

Acknowledgements: The project is supported by the Ministry of Industry and Trade of the Russian Federation within the R&D project "Development of a static analysis system for a hardware description language", code "CAD-Analysis" (as part of the R&D project "CAD microelectronics", the main contractor is JSC "ISTC MIET").

Применение инструмента SVAN статического анализа описаний аппаратуры для верификации открытых тестовых наборов

С.М. Панова, ORCID: 0009-0008-5106-0915 <panova@ispras.ru>
С.А. Смолов, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>
М.М. Волкова, ORCID: 0009-0009-8324-7562 < volchonok03@bk.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Российский экономический университет им. Г.В. Плеханова,
Россия, 115054, г. Москва, Стремянный переулок, д.36.

Аннотация. В статье представлены результаты экспериментального анализа инструмента SVAN статического анализа описаний цифровой аппаратуры на языках Verilog и SystemVerilog. Инструмент разрабатывается в ИСП РАН и предоставляет средства формального и эвристического анализа HDL-описаний, нацеленные на выявление синтаксических ошибок, нарушений стиля оформления кода, проблем безопасности. Эксперименты, проведенные на описаниях из открытого тестового набора hdl-benchmarks, демонстрируют более высокую эффективность SVAN в сравнении с открытым инструментом Verilator и проприетарным инструментом Synopsys VCS. В частности, SVAN обнаружил на 73% больше типов ошибок и на 23-25% больше ошибок в целом. Ключевые преимущества инструмента SVAN состоят в более высоком уровне локализации ошибок и развитой типологии ошибок. К выявленным недостаткам инструмента SVAN относится ограниченная поддержка RTL-моделей, в которых используется несколько языков описания. Полученные результаты подчеркивают потенциал SVAN как конкурентоспособного инструмента статического анализа в области автоматизации проектирования цифровой аппаратуры.

Ключевые слова: статический анализ; RTL-модель; HDL-описание; анализатор SVAN; языки описания аппаратуры Verilog, SystemVerilog; функциональная верификация; обнаружение ошибок; открытый тестовый набор.

Для цитирования: Панова С.М., Смолов С.А., Волкова М.М. Применение инструмента SVAN статического анализа описаний аппаратуры для верификации открытых тестовых наборов. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 131–142 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)-10.

Благодарности: Проект финансируется Министерством промышленности и торговли Российской Федерации в рамках ОКР «Разработка системы статического анализа для языка описания аппаратуры», шифр «САПР-Анализ» (в составе ОКР «САПР микроэлектроника», головной исполнитель – АО «МНТЦ МИЭТ»).

1. Introduction

The design of ultra-large-scale integrated circuits (ULSICs) is a complex task that requires high precision and reliability at all stages of development. In the early stages of designing digital ULSICs, they are described at the Register Transfer Level (RTL) using Hardware Description Languages (HDLs), such as Verilog [1] and SystemVerilog [2]. Modern RTL models can reach significant sizes, encompassing millions of lines of code [3], making automated functional verification as one of the key tasks in the development process. Functional verification is aimed at detecting design errors that could lead to incorrect operation of the ULSIC or its complete failure.

To solve the problem of functional verification, various methods are employed, each possessing their own advantages and limitations. Static analysis holds a special place among them, as a technique for examining HDL description without its simulation. This approach includes the analysis of the structure, syntax, and semantics of the source code, making applicable to identifying a wide range of issues, from syntactic errors to potential vulnerabilities [4].

Static analysis tools implement the following processing stages:

- *Code Analysis*. The source code is transformed into an abstract syntax tree (AST) or another internal representation (IR), enabling the tool to comprehend the structure of the code.
- *Rule Application*. A set of predefined rules or checks is applied to the IR, aimed at identifying issues of a specific type, such as syntactic errors, style violations, or logical inconsistencies.
- *Problem Reporting*. Upon detecting rule violations, the tool generates a report containing the exact location of the issue (usually, a file name, and a line number), a description of the problem, and, probably, a recommendation for its resolution.

Currently, there are both commercial and open-source tools available for static analysis of RTL models. Commercial tools include VCS [5] and SpyGlass [6] (Synopsys), JasperGold (Cadence) [7], and Questa (Siemens) [8]. Among the open-source tools for static analysis are Slang [9], Verilator [10], Svlint [11], Surelog [12], and Verible [13]. Open-source tools often exhibit limited functionality compared to their commercial counterparts. For example, they may not support the full range of HDL standards or demonstrate lower accuracy in detecting complex errors. Proprietary static analysis tools are currently either unavailable or restricted for use under sanction regimes, posing significant risks to the domestic electronics industry. This situation forces the industry to either adapt existing tools or completely abandon their use. Furthermore, the utilization of foreign EDA (Electronic Design Automation) tools carries inherent information security risks.

There are studies in the literature demonstrating the application of these tools [14-15]; however, experimental comparison of their effectiveness has not been conducted previously, making this task novel and relevant for scientific investigation.

Taking into account the above, the development of domestic EDA tools represents a strategic task that will ensure technological independence, information security, and the sustainable growth of the country's electronics industry. Currently, the Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS) is conducting research and development work aimed at creating SVAN, a static analysis tool for RTL models written in Verilog and SystemVerilog. SVAN tool development is supported by the Ministry of Industry and Trade of the Russian Federation.

The aim of this research is to apply the SVAN tool, currently under development [16], for the functional verification of open-source RTL models. The target design models selected for this purpose were benchmarks – sets of HDL descriptions originally intended for testing digital VLSI EDA tools, as well as for conducting comparative studies in this domain [17-18].

To achieve this goal, the following tasks were identified. First, an analysis of the applicability of SVAN for collection [19] of open-source RTL benchmarks was made. Next, the same benchmarks were analyzed using proprietary Synopsys VCS tool and open-source Verilator tool. After that, the results comparison was made.

A comparative analysis of open-source tools constitutes an independent scientific problem that falls outside the scope of the current study and has not been previously addressed in the literature. For the purposes of this comparison, we selected Verilator project – a widely recognized open-source tool that has been under active development since 2019 (currently in its sixth year of development). The tool’s popularity and broad acceptance within the community served as the primary criteria for its selection.

As the second tool in our comparison, we chose Synopsys VCS – a well-established commercial solution that is highly regarded in the industry and was accessible to us during the study. Both tools were selected due to their strong market presence, long-term development history, and widespread adoption in industrial applications.

An extended comparison of open-source tools may be pursued as a subject of future research and more comprehensive analysis.

2. SVAN tool description

Static analysis tools are designed to identify various issues in the code. These issues can be categorized into several classes.

1) *Simple syntactic* errors arise due to incorrect code writing, including the following:
   1). missing commas, semicolons, parentheses, etc.;
   2). incorrect use of language keywords or constructs;
   3). typo errors in variable names or function calls;
2) *Style violations* are issues related to coding standards [20-21], for example:
   1). inconsistent indentations;
   2). deprecated constructions usages;
   3). violations of naming conventions.
3) *Logical inconsistencies* represent a more complex class of issues that affect the behavior of the program or the design of the hardware. Examples of such inconsistencies include unreachable code (a situation where a portion of the code can never be executed due to logical errors) or race conditions (where the order of execution of code blocks directly impacts the overall outcome of the program).

Static analysis can also identify security-related issues, such as the use of uninitialized variables, which may lead to unpredictable behavior or improper handling of sensitive data, such as passwords or cryptographic keys.

The static analysis tool SVAN (Static Verification ANalysis tool) is a modern solution for verifying RTL models. SVAN supports Verilog (IEEE 1364-2005 standard) and SystemVerilog (IEEE 1800-2023 standard) HDL, enabling its use for verifying projects of varying complexity and purpose. The tool is designed to detect errors across various categories, which include the classes described above.

The architecture of SVAN is built on the principles of modularity and extensibility and is described on Figure 1. The tool includes the following key components: 1) SystemVerilog compiler (includes lexical analyzer, syntax analyzer and source code handling module); 2) analysis module (include rule detection module).

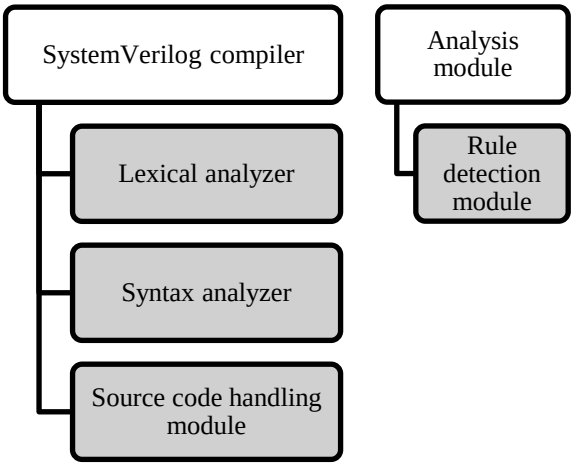

Fig. 1. SVAN Architecture.

The compiler is the central element of the SVAN architecture. It is responsible for processing the input code written in SystemVerilog and Verilog. The base compiler used is Slang – a modern, high-performance open-source compiler that supports the latest versions of the SystemVerilog standards (e.g., IEEE 1800-2023).

The advantages of using Slang lie in its ability to accurately analyze complex and large-scale projects. After processing the input code, the compiler transforms it into a specific IR, which serves as the input data for subsequent analysis stages.

The analysis module is the core of SVAN's functionality. It contains rule-based detectors designed to identify specific types of errors in the code. Each detector is focused on a particular category of issues, such as logical errors, coding style violations, or potential vulnerabilities.

All rules utilized by the tool have been classified into several categories based on language-related aspects and the development process. This classification ensures a systematic approach to identifying and addressing various types of issues in the code. Among the examples of categories are:

1) Assign – checks for assignment operations
2) Case – checks for case statements
3) Loop – checks for loop operators
4) Range – checks for out-of-bounds violations
5) Type – checks for type-related conflicts, and others

The detectors in the tool are implemented using two primary approaches: formal methods and heuristics.

To evaluate the performance and reliability of SVAN, it was tested on several well-known open-source hardware projects, including PicoRV32 [22], CVA6 [23], and OpenTitan [24]. The testing process involved analyzing these projects to identify potential issues across various categories, such as logical inconsistencies, coding style violations, and structural errors. The results of the testing confirm that SVAN is a reliable and efficient static analysis tool. Its ability to effectively handle complex real-world projects demonstrates its suitability for industrial use. By successfully processing large and intricate codebases, SVAN has proven its capability to meet the demands of modern hardware design verification, making it a valuable asset for both academic research and commercial applications in the field of electronic design automation (EDA).

3. Experimental analysis

In the course of the study, a file-by-file processing methodology was applied to the benchmark set. Each file was analyzed sequentially using a static analysis tool. In the event of errors indicating the absence of required modules or files, an attempt was made to locate the missing components within the benchmark collection itself.

If the necessary files could not be found, the case was labeled as an error, and a stub module with a required interface was created. These stub modules were placed in the same directory as the original file to ensure correct path resolution during inclusion. For each recurring type of error, a generalized description was compiled, including the error category and the recommended correction (see Table 1). Descriptions of certain errors related to implementation specifics were supplemented with detailed explanations.

After applying the corrections, each file was re-analyzed using the same tool to verify that the error had been resolved and that no further diagnostic messages were issued by the analyzer. No time limits were set for the static analysis tool.

The main contribution of this paper consists of two key aspects. First, a comprehensive comparative evaluation of existing static analysis tools was conducted in the context of digital circuit verification tasks. Second, a number of errors were identified and corrected within a widely used benchmark collection, thereby improving its overall quality and suitability for use in both academic and practical applications.

As part of the conducted research to analyze the applicability of SVAN for open RTL benchmarks, the tool was tested on the open-source `hdl-benchmarks` collection. This collection includes the following sets of benchmarks:

1) ICCAD-2015 CAD Contest benchmark suite;
2) ISCAS'85 benchmarks [25];
3) ISCAS'89 benchmarks [26];
4) IWLS'2005 benchmarks [17];
5) LGSynth'1991 benchmarks [27];
6) Quartus University Interface Program (QUIP) benchmarks [18];
7) Texas-97 benchmarks [28];
8) VCEGAR benchmarks [29];
9) Verilog2SMV benchmarks [22];

These benchmarks were widely used for the verification of HDL design and synthesis tools [23-24]. However, it has not yet been analyzed using static analysis methods. These projects encompass a broad spectrum of algorithms and functionalities, which served as the decisive factor in their selection as a suitable object for comprehensive analysis. Such a wide-ranging representation ensures that the tool's capabilities are tested under conditions that closely mimic real-world hardware design scenarios, thereby providing a robust assessment of its effectiveness and reliability.

Results of experimental evaluation of analyzed tools (with default running options) are shown in Table 1. On the first column error types are described. Last columns are named by the related static analysis tools: SVAN (ISP RAS), VCS (Cadence) V-2023.12 and Verilator 4.028 2020-02-06 respectively.

Table 1. Errors are detected by Static Analysis Tools.

Error type	SVAN	VCS	Verilator
Unknown macros or compiler directives detected	36	8	134
Expected expression	35	0	0
Unknown module	290	236	97
No implicit conversion; explicit conversion exists but casting is missing	2	0	0
Unable to resolve hierarchical path	1	0	0
Module redefinition	4	0	0
No such file or directory	26	77	79
Invalid delay value expression	2	0	0
Size requires a constant range	3	0	0
Too many arguments passed	1	0	0
Incompatible bit widths	1	0	0
Unknown system name	2	0	0
Missing identifier	5	0	0
Too many connections specified for port instantiation	2	2	0
Other	23	0	24
Total	*433*	*323*	*334*

The errors classified as "*Unknown macros or compiler directives detected*" are essentially a consequence of the absence of the source code referencing macros or directives that are not defined within the analyzed files or their associated include paths. Below is an example of the error containing code and related diagnostics:

```
unknown macro or compiler directive '`RDY'
            if (ACK == `RDY)
```

The "*Unknown module*" error indicates that the code instantiates a module whose declaration is not present in the example being analyzed by the tool.

The "*No such file or directory*" error occurs when the code attempts to include a header or other file that does not exist in the directory where the analyzed code is located. This type of error is the second most frequently encountered among those identified. Its presence in the benchmarks may be attributed to inaccuracies introduced during the aggregation of the original projects into the `hdl-`

`benchmarks` collection. However, configuring the tool execution flow does not provide an opportunity to rectify such errors, as it is not feasible to recover the missing components and reconstruct the projects outside the scope of this collection.

Table 2 presents the solutions proposed as part of the measures taken to rectify errors present in the open-source benchmarks. In particular, to address errors related to missing modules, functions, or macro definitions, a search was conducted for the corresponding file within the project directories. If the file was not found, a stub file implementing the required module, function, or macro was created accordingly. Additionally, to resolve the issue of missing explicit type casting, it was introduced using a system task.

During the experimental testing, it was determined that the SVAN tool demonstrates significant superiority over both the commercial Synopsys VCS and the open-source Verilator tool across several key metrics. In particular, the number of classes of errors detected by SVAN (15) was found to be 73% broader than that both of VCS (4) and Verilator (4). Additionally, the total number of errors identified by SVAN (433) in the analyzed RTL model modules exceeds the corresponding metric for VCS (323) by 25% and for Verilator (332) by 23% (see “Total” row in Table 1).

This difference can be attributed to several factors.

Table 2. Errors and Suggested Solutions.

Error type	Suggested correction
Unknown macros or compiler directives detected	Check the project directory to see whether the required file exists. If it does, import it into the module. If not, create a stub file in the project directory and import it into the module.
Expected expression	
Unknown module	
No such file or directory	
No implicit conversion; explicit conversion exists but casting is missing	Add an explicit cast using `$sformatf("%s", in_file)`
Unable to resolve hierarchical path	In the included file, initialize the used parameters and assign them appropriate values
Module redefinition	Review all included files to determine whether the same file is being included twice through different paths into the module under investigation. Then adjust the file inclusion order to prevent module redeclaration
Invalid delay value expression	Add a file containing the defined delay value expressions
Size requires a constant range	Include a `.vh` file that contains the required constant range definitions
Too many arguments passed	Modify both the function definition and its usage within the module so that no more arguments are passed than are required
Unknown system name	Add a stub file implementing a function with the specified name
Missing identifier	Declare the missing identifier
Too many connections specified for port instantiation	Correct the instance declaration so that no more port connections are provided than are necessary

3.1 Functional limitations of Synopsys VCS and Verilator

The VCS tool terminates its analysis upon encountering the first error of a specific category. For instance, if an error arises due to a missing included file or directory, the tool halts further error detection not only within that category but also across all other categories, even though additional errors may still exist in the code. There are no any options allowing to continue further errors detection, which leads to significant restrictions the comprehensiveness of the analysis.

Verilator does not correctly handle the `‘include <path-to-file>’` directive (even when using the additional command-line argument `+incdir+<path-to-dir>`) which is used to include external files in module code. As a result, the tool fails to locate included files, even when they exist in the same directory as the module being analyzed. This limitation leads to a large number of false negatives, as Verilator generates errors related to “missing” files that are actually accessible during compilation.

By contrast, SVAN continues its analysis regardless of the presence of initial errors, ensuring a more thorough and exhaustive examination of the codebase. This capability allows SVAN to identify a broader spectrum of errors, thereby providing a more complete picture of the design's integrity and potential vulnerabilities.

3.2 Granularity of error classification

SVAN provides a more detailed classification of errors compared to both VCS and Verilator. For instance, errors related to invalid delay value expressions and the absence of macro or compiler directive definitions are grouped under a single marker in VCS, labeled as "*Error-[UM] Undefined macro*".

Verilator assigns a broad category of "*Syntax error*" to multiple distinct error types identified by SVAN, such as missing identifiers, unknown system names, and passing too many arguments to a function.

This broad categorization can obscure the specific nature of the issues, making it more challenging for developers to identify and address the root causes effectively.

In contrast, SVAN distinguishes between these categories of errors, enabling more precise diagnostics. For example, issues like missing predefined delay expressions, undefined macros, unresolved hierarchical paths, and incorrect argument counts are flagged as independent errors. This approach allows users to pinpoint the exact nature of each issue, facilitating more efficient debugging and resolution. By providing clearer and more granular feedback, SVAN ensures that developers receive actionable insights into the specific problems within their code, ultimately enhancing the overall quality and reliability of the design process.

3.3 Absence of detection for certain error categories

Furthermore, the analysis revealed significant limitations in both Synopsys VCS and Verilator.

Synopsys VCS tool does not detect certain categories of errors, such as the absence of implicit type casting. This limitation can result in potentially problematic code sections going unnoticed, thereby increasing the risk of errors during synthesis or simulation stages. For instance, issues related to incompatible data types or missing explicit type conversions may remain undetected, leading to unpredictable behavior or functional failures in the hardware design.

Verilator has an even narrower range of detectable errors. It cannot identify issues such as unresolved hierarchical paths, module redefinitions, or many other errors that could impact the correctness of the system.

On the contrary, SVAN is capable of identifying such non-obvious errors due to its inclusion of detectors specifically designed to verify the correct usage of data types. These detectors explicitly check for cases where implicit type casting is expected but absent, ensuring that all type-related issues are flagged and addressed. By incorporating these advanced checks, SVAN provides a more thorough analysis of the code, reducing the likelihood of oversight and enhancing the overall robustness of the design process.

During experimental testing, it was also found that 5% of the analyzed RTL models were incorrectly classified as containing errors by SVAN and Verilator, whereas the Synopsys VCS tool marked them as correct and they were actually correct. They were marked as "*Other*" in the Table 1.

Those issues arise due to the specific characteristics of code analysis performed in accordance with the Verilog IEEE 1364-2005 and SystemVerilog IEEE 1800-2023 standards. When verifying benchmarks containing mixed code written in both languages, valid Verilog code did not meet the stricter requirements of the SystemVerilog standard. For example, a variable named `do` was recognized as a reserved keyword introduced in the SystemVerilog standard. As a result, such models were flagged as erroneous, even though they were actually compliant with the Verilog standard.

The broader spectrum of detectable errors, increased level of analysis detail, and absence of the limitations of VCS and Verilator make SVAN a more effective tool for the static verification of RTL models.

Table 3 shows the errors distribution on the selected benchmarks. The analysis reveals that the IWLS-05 benchmark contains the highest number of errors among the evaluated datasets. Specifically, SVAN, VCS, and Verilator detected 214, 214, and 212 errors (see "*IWLS-05*" row in Table 3) respectively, in this benchmark, making it the most error-prone dataset in absolute terms. A significant proportion of these errors – 77% (165 out of 214) – are attributed to the absence of a file containing the module whose instance is instantiated in the analyzed file. This issue represents the most frequently encountered error type in the IWLS-05 benchmark.

Another notable dataset is the QUIP benchmark, which was released as part of Altera's educational program. While QUIP is not the largest benchmark in terms of error count, it exhibits a relatively high error rate, with 99 errors detected (see "QUIP" row in Table 3). The higher error frequency in QUIP can be attributed to its lesser degree of debugging compared to other benchmarks. Similar to IWLS-05, the most common error in QUIP is the absence of a module corresponding to an instantiated instance. Additionally, the minor variability in error detection rates among the tools suggests that QUIP presents specific challenges for verification tools.

Table 3. Errors to Benchmarks distribution.

Benchmark	SVAN	VCS	Verilator
Verilog2SMV	1	1	1
VCEGAR	5	0	5
TEXAS-97	17	17	17
QUIP	99	99	97
IWLS-05	214	214	212
ISCAS89	4	4	4

The benchmark with the fewest errors, based on the results of the empirical study, is Verilog2SMV (see "Verilog2SMV" row in Table 3). This may be attributed to the fact that this benchmark serves as a test suite specifically developed for testing the eponymous tool by the Bruno Kessler Foundation [30].

Additionally, only four errors of the type "*Unknown Module*" were detected by all tested static analysis tools on the ISCAS-89 benchmark. The ISCAS-89 benchmark was originally distributed on tape to participants of the Special Session on Sequential Test Generation at the International Symposium on Circuits and Systems in May 1989 and is partially characterized in [26]. This is the reason for the number of errors contained in this benchmark being minimal and representing only one class of errors outlined in Table 1.

Overall, the tools SVAN and VCS demonstrate robust performance across all benchmarks, consistently identifying the highest number of errors. Their consistent results highlight their reliability in detecting issues, even in less refined datasets like QUIP.

In terms of tool performance, SVAN and VCS demonstrate nearly identical results across all benchmarks, consistently detecting the same number of errors. The primary discrepancy lies in the complete absence of error detection by VCS within the VCEGAR benchmark, where SVAN demonstrates superior performance by identifying errors, matching the results achieved by Verilator. This highlights the robustness of SVAN in handling the complexities of the VCEGAR dataset, further solidifying its position as a leading tool in error detection across diverse benchmarks.

4. Conclusions and final remarks

Based on the empirical study, a number of unique characteristics of SVAN were identified, setting it apart from existing solutions for the verification of hardware descriptions written in Verilog and SystemVerilog. The key advantage of SVAN compared to Synopsys VCS and Verilator is high analysis accuracy. SVAN's ability to provide more granular and precise diagnostics ensures that

even subtle issues are identified and properly categorized, reducing the likelihood of undetected errors that could compromise the design process.

Despite its significant advantages, the SVAN tool has certain limitations that must be taken into account when using it. One of the key drawbacks is its insufficient flexibility in handling RTL models are written in two hardware description languages – both Verilog and SystemVerilog. This limitation highlights the need for further improvements in SVAN's ability to handle mixed-language designs and to differentiate between language-specific constructs more effectively. While SVAN's strict adherence to SystemVerilog standards ensures high accuracy in many cases, it can also lead to false positives when analyzing legacy Verilog code or designs that combine both languages. Addressing this challenge will be critical for enhancing the tool's compatibility and usability across a wider range of hardware design projects.

The results presented in this work are valuable both in terms of comparing the developed SVAN tool with proprietary counterparts and in a broader context. On one hand, the study has a scientific focus, involving the analysis and comparison of functional capabilities among existing static analysis tools. On the other hand, it can be regarded as a technical report reflecting the current state and practical capabilities of the SVAN tool.

The further development of this research includes a broader comparison of the static analysis tool SVAN with existing commercial and non-commercial tools, including an assessment of performance and functional capabilities. This will be the subject of future studies and will allow for a deeper evaluation of the proposed method's potential under real-world application conditions.

References

[1]. IEEE SA. IEEE Standard for Verilog Hardware Description Language [Online]. Available: https://standards.ieee.org/ieee/1364/3641/ (accessed 2025, Mar. 27).

[2]. IEEE SA. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language [Online]. Available: https://standards.ieee.org/ieee/1800/7743/ (accessed 2025, Mar. 27).

[3]. NVDLA Project, "NVIDIA Deep Learning Accelerator (NVDLA) – Open-Source Inference Accelerator", GitHub repository, 2017-2023. [Online]. Available: https://github.com/nvdla (accessed 2025, Mar. 27).

[4]. GOST R 71207-2024, "Information Protection. Secure Software Development. Static Code Analysis. General Requirements", Moscow: Standartinform, 2024. [Online]. Available: https://protect.gost.ru/v.aspx?control=7&id=257752 (accessed 2025, Mar. 27).

[5]. Synopsys, "VCS® Functional Verification Solution", Synopsys Inc., 2023. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html (accessed 2025, Mar. 27).

[6]. Synopsys, "SpyGlass: Early Design Analysis Tools for SoCs", Synopsys Inc., 2025. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html (accessed 2025, Jun.12).

[7]. Cadence Design Systems, "Formal and Static Verification Solutions", Cadence Inc., 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html (accessed 2025, Mar. 27).

[8]. Siemens EDA (2025, Mar. 27). Questa Verification Solutions, Siemens Digital Industries Software, 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/ (accessed 2025, Mar. 27).

[9]. M. Popoloski, slang: SystemVerilog Language Services, version 8.0, 2015-2025. [Online]. Available: https://github.com/MikePopoloski/slang (accessed 2025, Mar. 27).

[10]. Verilator. Verilator – a fast Verilog/SystemVerilog simulator. GitHub repository, 2003-2024. [Online]. Available: https://github.com/verilator/verilator (accessed 2025, Mar. 27).

[11]. D. Alance. svlint – SystemVerilog linter. GitHub repository 2020-2024. (2025, Mar. 27). [Online]. Available: https://github.com/dalance/svlint (accessed 2025, Mar. 27).

[12]. CHIPS Alliance, Surelog v1.84 – SystemVerilog 2017 Toolchain, 2024. [Online]. Available: https://github.com/chipsalliance/Surelog (accessed 2025, Mar. 27).

[13]. CHIPS Alliance. Verible – Suite of SystemVerilog developer tools. GitHub repository, 2019-2024. [Online]. Available: https://github.com/chipsalliance/verible (accessed 2025, Mar. 27).

[14]. T. Pecenka, L. Sekanina, and Z. Kotasek, "Evolution of synthetic RTL benchmark circuits with predefined testability", ACM Trans. Des. Autom. Electron. Syst., vol. 13, no. 3, pp. 1–21, Jul. 2008.

[15]. F. Yuan, “Design and Test for Timing Uncertainty in VLSI Circuits”, Ph.D. dissertation, Chinese University of Hong Kong, Hong Kong, 2012.
[16]. Ya. A. Churkin, R. A. Buchatskiy, K. N. Kitaev, A. G. Volokhov, E. V. Dolgodvorov, A. S. Kamkin, A. M. Kotsynyak, and D. O. Samovarov, “Static Analysis System for SystemVerilog Hardware Description Language”, Proceedings of the ISP RAS, vol. 37, no. 1, pp. 7–40, 2025.
[17]. International Workshop on Logic & Synthesis (IWLS), “IWLS 2005 Benchmark Suite”, 2005. [Online]. Available: https://iwls.org/iwls2005/benchmarks.html (accessed 2025, Mar. 27).
[18]. N. Isaac, “QUIP Toolkit Benchmarks (v9.0)”, ECE496 GitHub repository, 2020. [Online]. Available: https://github.com/neilisaac/ece496/tree/master/reference/quip_toolkit-9.0/benchmarks (accessed 2025, Mar. 27).
[19]. hdl-benchmarks – Collection of digital hardware modules & projects (benchmarks), GitHub repository, 2019-2023. [Online]. Available: https://github.com/ispras/hdl-benchmarks (accessed 2025, Apr. 06).
[20]. Freescale Semiconductor, Verilog HDL Coding: Semiconductor Reuse Standard. Freescale Semiconductor, Inc., 2005.
[21]. M. Taylor and Bespoke Silicon Group. BSG SystemVerilog Coding Standards, University of Washington, 2023.
[22]. C. Wolf, “PicoRV32 – A Size-Optimized RISC-V CPU Core”, YosysHQ GitHub repository, 2015-2023. [Online]. Available: https://github.com/YosysHQ/picorv32 (accessed 2025, Mar. 27).
[23]. OpenHW Group, “CVA6 – An Open-Source 64-bit RISC-V CPU Core”, GitHub repository, 2019-2023. [Online]. Available: https://github.com/openhwgroup/cva6 (accessed 2025, Mar. 27).
[24]. lowRISC, “OpenTitan – Open Source Silicon Root of Trust”, GitHub repository, 2018-2023. [Online]. Available: https://github.com/lowRISC/opentitan (accessed 2025, Mar. 27).
[25]. M. Hansen, H. Yalcin, J. Hayes, “Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering”, IEEE Design & Test of Computers, vol. 16, no. 3, pp. 72-80, 1999.
[26]. F. Brglez, D. Bryan, K. Kozminski, “Combinational Profiles of Sequential Benchmark Circuits”, Proc. IEEE Int. Symposium on Circuits and Systems, pp. 1929-1934, May 1989.
[27]. Microelectronics Center in North Carolina. LGSynth91 benchmarks. [Online]. Available: https://ddd.fit.cvut.cz/www/prj/Benchmarks/LGSynth91.7z (accessed 2025, Jun. 14).
[28]. Department of Electrical and Computer Engineering at the University of Texas. Texas-97 benchmarks. [Online]. Available: https://ptolemy.berkeley.edu/projects/embedded/research/vis/texas-97 (accessed 2025, Jun. 14).
[29]. University of Oxford. VCEGAR benchmarks. [Online]. Available: http://www.cprover.org/hardware/benchmarks/vcegar-benchmarks.tgz (accessed 2025, Jun. 14).
[30]. FBK, “Verilog2SMV”, Fondazione Bruno Kessler (FBK). [Online]. Available: https://es-static.fbk.eu/tools/verilog2smv/ (accessed 2025, Mar. 29).

Информация об авторах / Information about authors

София М. Панова является лаборантом-исследователем отдела Технологий Программирования Института системного программирования им. В.П. Иванникова Российской академии наук (ИСП РАН). Область научных интересов: цифровая аппаратура, статический анализ, функциональная верификация.

Sophia M. Panova – Laboratory assistant at the Software Engineering Department of the Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS). Her research interests include digital hardware design, static analysis, and functional verification.

Сергей А. СМОЛОВ является научным сотрудником отдела Технологий Программирования Института системного программирования им. В.П. Иванникова Российской академии наук (ИСП РАН), старший научный сотрудник научной лаборатории «Гетерогенные компьютерные системы» РЭУ им. Г.В. Плеханова. Область научных интересов: автоматизация проектирования цифровой аппаратуры, верификация и тестирование.

Sergey A. SMOLOV – Researcher at the Software Engineering Department of Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS), senior researcher at the

Heterogeneous Computing Systems research lab of Plekhanov RUE. His research interests include digital hardware design automation, verification and testing.

Марина М. ВОЛКОВА является лаборантом отдела Технологий Программирования Института системного программирования им. В.П. Иванникова Российской академии наук (ИСП РАН). Область научных интересов: машинное обучение, интеллектуальный анализ данных, методы статического анализа и верификации.

Marina M. VOLKOVA – Laboratory assistant at the Software Engineering Department of Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS). Her research interests include machine learning, data analysis, methods of static analysis and verification.