DOI: 10.15514/ISPRAS-2026-38(4)-18

Automatic Synthesizability Checking of Hardware Descriptions

[1,2] A.A. Shevtsov, ORCID: 0009-0001-3633-9466 <s02250601@gse.cs.msu.ru>
[2] Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>
[2] R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
[1,2] A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>
[2] D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>

[1] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

[2] Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Modern VLSI design flows use the Verilog and SystemVerilog hardware description languages for both logic synthesis and simulation. However, not every language construct is synthesizable, the sets of supported constructs differ across synthesis tools, and the tools do not always diagnose non-synthesizable constructs: a construct may be silently dropped or reinterpreted, causing a mismatch between the behavior of the RTL model and the synthesized netlist. Problems discovered late in the design flow require corrections at every stage and incur significant time and financial costs, which motivates automated synthesizability checking of RTL descriptions. This paper presents a set of 34 synthesizability rules for Verilog and SystemVerilog hardware descriptions formulated on the basis of the IEEE/IEC 62142-2005 standard, the STARC RTL Design Style Guide, and the Reuse Methodology Manual, as well as the experience of commercial tools. The rules cover constructs not used in synthesis, the semantics of hardware element descriptions, and synthesis–simulation divergence. Static checkers verifying source code against these rules were implemented within the SVAN static analysis system: ten new checkers were developed (SYNTH14–SYNTH23), one existing checker was updated (SYNTH13), and further rules are covered by previously implemented SVAN checkers. A configurable conservativeness mechanism is proposed that adapts checker strictness to the capabilities of a particular synthesis tool, including the use of data-flow analysis; recommended checker configurations were derived for the Yosys, Vivado, and Genus synthesis tools. An evaluation using the open-source Yosys synthesis tool confirmed that non-synthesizable constructs can be processed silently with altered semantics, demonstrating the practical value of early static checking. A comparison with the commercial SpyGlass analyzer revealed specific gaps in its analysis, and rule violations were found in all ten open-source projects the checkers were applied to; one of the violations was reported to the developers of the OpenC910 project.

Keywords: static analysis; synthesizability; logic synthesis; SystemVerilog; Verilog; hardware description language; RTL; VLSI; EDA; Yosys.

For citation: Shevtsov A.A., Churkin Y.A., Buchatskiy R.A., Kamkin A.S., Melnik D.M. Automatic Synthesizability Checking of Hardware Descriptions. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 4, part 2, pp. 45-66. DOI: 10.15514/ISPRAS–2026–38(4)–18.

Автоматическая проверка синтезируемости описаний аппаратуры

[1,2] А.А. Шевцов, ORCID: 0009-0001-3633-9466 <s02250601@gse.cs.msu.ru>
[2] Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>
[2] Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
[1,2] А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>
[2] Д.М. Мельник, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>

[1] Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

[2] Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Современное проектирование цифровых СБИС предполагает использование языков описания аппаратуры Verilog и SystemVerilog как для логического синтеза, так и для симуляции, однако не все конструкции этих языков синтезируемы, а множество поддерживаемых конструкций различается между инструментами синтеза. При этом инструменты синтеза не всегда диагностируют несинтезируемые конструкции и могут молча изменять семантику схемы, что приводит к расхождению поведения RTL-модели и синтезированной схемы, а исправление проблем, обнаруженных на поздних этапах маршрута проектирования, требует значительных временных и экономических затрат. В работе описано формирование набора правил синтезируемости для описаний аппаратуры на языках Verilog и SystemVerilog на основе стандарта IEEE/IEC 62142-2005, руководства STARC RTL Design Style Guide и Reuse Methodology Manual, а также опыта коммерческих инструментов: сформулировано 34 правила, охватывающих не используемые при синтезе конструкции, семантику описаний аппаратных элементов и расхождение синтеза и симуляции. В рамках системы статического анализа SVAN реализованы статические детекторы для проверки соответствия исходного кода этим правилам: разработано 10 новых детекторов (SYNTH14–SYNTH23), обновлен 1 существующий (SYNTH13), часть правил покрывается ранее реализованными детекторами системы. Предложен механизм настройки уровня консервативности, позволяющий адаптировать строгость проверок к возможностям конкретного инструмента синтеза, в том числе с применением анализа потока данных; получены рекомендуемые конфигурации детекторов для инструментов синтеза Yosys, Vivado и Genus. Апробация на открытом инструменте Yosys показала, что несинтезируемые конструкции могут обрабатываться им без предупреждений с изменением семантики схемы, что подтверждает практическую значимость раннего статического анализа. Сравнение с коммерческим анализатором SpyGlass выявило конкретные пробелы в его анализе; нарушения правил найдены во всех десяти открытых проектах, на которых запускались детекторы, об одном из нарушений сообщено разработчикам проекта OpenC910.

Ключевые слова: статический анализ; синтезируемость; логический синтез; язык описания и верификации аппаратуры SystemVerilog; язык описания аппаратуры Verilog; язык описания аппаратуры; язык передачи регистров RTL; сверхбольшие интегральные схемы СБИС; системы автоматизации проектирования САПР; синтез соединений Yosys.

Для цитирования: Шевцов А.А., Чуркин Я.А., Бучацкий Р.А., Камкин А.С., Мельник Д.М. Автоматическая проверка синтезируемости описаний аппаратуры. Труды ИСП РАН, 2026, том 38, вып. 4, часть 2, стр. 45–66 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(4)–18.

1. Introduction

Modern VLSI design encompasses multiple stages, including RTL modeling, logic synthesis, and functional verification. The first step is typically the creation of an RTL (register transfer level) model of the circuit. Hardware Description Languages (HDL) such as Verilog [1] and SystemVerilog [2] are used for this purpose. Synthesizability is the property of an HDL description of having an equivalent representation as a netlist of logic gates from the target technology library.

SystemVerilog is a hardware design, specification, and verification language. Its broad applicability allows a single design to be reused across different stages without needing separate descriptions for

synthesis and simulation. However, simulation-oriented constructs are not always interpretable by synthesis tools. Moreover, the set of unsupported constructs varies across tools, making it difficult to formally define a synthesizable language subset.

An attempt to define the synthesizable Verilog subset was made by the Design Automation Standards Committee: the resulting standard was published as IEEE Std 1364.1-2002 and reissued as the international standard IEC 62142:2005 [3] (hereinafter IEEE/IEC 62142-2005). The standard was subsequently withdrawn, and no counterpart for SystemVerilog has appeared. Guidance documents such as the STARC RTL Design Style Guide [4] and the Reuse Methodology Manual [5] exist, as well as documentation for particular synthesis tools, but they do not provide a complete picture of construct synthesizability, and developers rely largely on personal experience.

Existing synthesis tools lack thorough non-synthesizability diagnostics. The open-source tool Yosys [6] may silently alter the semantics of non-synthesizable constructs in the generated netlist without indicating the reason. Problems discovered late in the design flow may require corrections at every stage, incurring time and financial costs.

There is therefore a need for automated synthesizability checking of RTL descriptions. This paper describes the formulation of a synthesizability rule set on the basis of the IEEE/IEC 62142-2005 standard and related documents, and the implementation of static checkers verifying source code against these rules within the SVAN static analysis system [7]. A test suite with synthesizable and non-synthesizable SystemVerilog constructs was developed; the implemented solution was evaluated by analyzing the behavior of the open-source Yosys synthesis tool on it, the checkers were compared with their SpyGlass counterparts in terms of precision and recall, and they were applied to open-source hardware projects.

2. Overview of Existing Tools

Open-source and commercial static analysis tools for hardware descriptions support synthesizability checks to varying degrees. This section reviews the tools closest to the goals of this work; a more complete overview of static analysis tools for SystemVerilog descriptions, including the linting module built into the Verilator simulator, is given in [7], where the SVAN checkers are also compared with Verilator.

2.1 Verible

Verible [8] is an open-source toolset for SystemVerilog development. It includes the static analysis tool verible-verilog-lint. The tool's rule list consists mostly of style recommendations and contains few rules affecting synthesizability: (1) `always-comb` – forbids `always @*` and requires combinational logic to be explicitly wrapped in `always_comb` blocks; (2) `always-comb-blocking` – forbids non-blocking assignments in `always_comb`; (3) `always-ff-non-blocking` – requires non-blocking assignments in `always_ff` (except for local variables); (4) `case-missing-default` – checks for a `default` branch in `case` statements; (5) `invalid-system-task-function` – forbids non-synthesizable system functions.

2.2 svlint

svlint [9] is an open-source SystemVerilog linter. It provides numerous rules aimed at coding style and backward compatibility with Verilog. Several rules can be attributed to preventing synthesis–simulation divergence: (1–3) `blocking_assignment_in_always_at_edge`, `blocking_assignment_in_always_ff`, `blocking_assignment_in_always_latch` – forbids blocking assignments when modeling sequential logic; (4–7) `case_default`, `explicit_case_default`, `implicit_case_default`, and `explicit_if_else` – require completeness of `case` and `if` statements; (8) `general_always_no_edge` – forbids `always @*` in favor of `always_comb`.

2.3 slang-tidy

The open-source slang library [10] provides lexing, parsing, type checking, and elaboration of SystemVerilog code. It ships with a set of executable tools, one of which is slang-tidy, intended for static analysis of SystemVerilog code. The tool implements a set of checkers, some of which target the synthesizability of descriptions:

(1) `LoopBeforeResetCheck` – requires the reset check to be inside a loop rather than outside;
(2) `UndrivenRange` – searches for undriven bits;
(3) `UnusedSensitiveSignal` – forbids signals in a sensitivity list that are not used in the block;
(4) `OnlyAssignedOnReset` – forbids registers that are assigned only while reset is asserted;
(5) `RegisterHasNoReset` – forbids registers sensitive to reset but lacking an assignment in the corresponding branch;
(6) `XilinxDoNotCareValues` – forbids the `x'd` notation for don't-care values;
(7) `CastSignedIndex` – forbids casting array indices to signed types;
(8) `NoLatchesOnDesign` – detects latches in a design;
(9) `AlwaysFFAssignmentOutsideConditional` – forbids assigning to a register with reset outside a conditional statement.

2.4 SpyGlass

SpyGlass [11] is a static verification and RTL analysis tool by Synopsys. SpyGlass Lint is its static analysis module that checks RTL code against a rule set. SpyGlass rules are organized into categories; most relevant to this work are the STARC category, containing hundreds of rules from the RTL Design Style Guide for Verilog HDL, and the SYNTH category, containing 85 rules devoted to synthesizability checking. The tool is commercial, which limits its free use and extension. Nevertheless, the SpyGlass experience in restricting language constructs was taken into account when formulating the rules (see Section 3), and a comparison of the implemented checkers with their SpyGlass counterparts on synthetic tests allowed us to assess the effectiveness of the implemented solution (see Section 6.3).

Table 1. Support for synthesizability checking in the reviewed tools.

Tool	Availability	Synthesizability-related rules
Verible	open source	5 rules
svlint	open source	8 rules
slang-tidy	open source	9 checkers
SpyGlass	commercial	85 rules in the SYNTH category; hundreds of STARC-based rules

The reviewed open-source tools contain few synthesizability-related rules (Table 1) and do not cover many of the rules considered in the next section; SpyGlass is more complete but commercial.

2.5 Formal approaches to the synthesizable subset

A separate line of research approaches synthesizability formally rather than empirically. Formal semantics have been developed for Verilog [12] and, more recently, for its synthesizable fragment [13], and verified compilation flows translate a restricted subset of the language into a netlist with a machine-checked proof of correctness [14]. Such approaches prescribe a subset within which correctness can be established, and their guarantees hold only for descriptions written inside that

subset. The problem considered in the present paper is complementary: existing RTL code is written in full SystemVerilog, and the practically relevant question is whether its constructs are accepted, and interpreted in the same way, by the synthesis tools actually used in the design flow. That this question is not vacuous is confirmed independently by fuzzing studies of FPGA synthesis tools, which found defects that make the netlist differ from the semantics of the source description [15].

3. Formulation of Synthesizability Rules

The rule set is based on a number of industry documents imposing requirements on RTL code: the IEEE/IEC 62142-2005 standard, the STARC RTL Design Style Guide for Verilog HDL, and the Reuse Methodology Manual.

3.1 The IEEE/IEC 62142-2005 standard

Work on the synthesizable Verilog subset started in a working group of Open Verilog International; in 1998 the project was transferred to the IEEE, where it was published as IEEE Std 1364.1-2002, Verilog Register Transfer Level Synthesis, and in 2005 it was reissued as the international standard IEC 62142:2005 [3]. The standard defines the syntax and semantics of the subset of Verilog (as of IEEE Std 1364-2001) suitable for register-transfer-level synthesis. Its purpose is to ensure the portability of descriptions across compliant synthesis tools and to minimize functional mismatches between the RTL model and the synthesized netlist. The SystemVerilog language standard itself (IEEE 1800 [2], also published as IEC/IEEE 62530) defines the language and its simulation semantics but does not define a synthesizable subset.

The standard divides all language constructs into three categories: supported – a synthesis tool shall interpret the construct and map it to hardware; ignored – the construct is not mapped to hardware, encountering it shall not cause synthesis to fail, but it may cause a functional mismatch between the RTL model and the synthesized netlist, and the mechanism by which the tool notifies the user is not defined; not supported – a synthesis tool shall fail upon encountering the construct. It is the ignored category that poses the greatest danger of synthesis–simulation divergence and motivates the use of static analysis: the standard permits silently excluding such constructs from the circuit.

In terms of content, the standard constrains data types, assignment operators, control constructs, memory element descriptions, and system function usage; defines modeling styles for hardware elements (combinational logic, flip-flops and latches, tri-state drivers, the `x` and `z` values (the unknown and high-impedance values, respectively), ROM and RAM); introduces synthesis attributes (`full_case`, `parallel_case`, `ram_block`, `fsm_state`, etc.) that a compliant tool shall support; and describes a verification methodology for matching the model before and after synthesis. Annex B lists typical sources of functional mismatches: incomplete sensitivity lists, mis-ordered assignments, `casex` and `casez` statements, timing delays, and assignments in variable declarations. Although the standard was withdrawn, many of its constraints are still used in practice.

3.2 STARC RTL Design Style Guide and Reuse Methodology Manual

The STARC RTL Design Style Guide for Verilog HDL [4] describes rules, recommendations, and suggestions for RTL code authoring. A subset of the rules covers the synthesizability of hardware element descriptions and the correspondence between synthesis results and the original RTL model. Many static analysis tools (including SpyGlass and Alint-PRO [16]) implement rules based on this document, which confirms its practical significance.

The Reuse Methodology Manual [5] presents rules and recommendations that formed the basis of the OpenMORE tool for assessing IP block reusability. The OpenMORE rules, like STARC, largely shaped the rule sets of the aforementioned static analyzers.

3.3 The rule set

Based on these documents, a set of 34 rules for static checkers was compiled. The standard is not structured as an enumerated list of rules: its requirements are stated as separate normative statements spread over the clauses, so the rules were derived by working through these statements, while the STARC guide, the Reuse Methodology Manual, and the SpyGlass rule set were used to refine the resulting formulations rather than as independent sources of rules. Some rules are new; others adapt existing SVAN checkers; the working rule list was subsequently extended with further synthesizability-related rules that are covered by previously implemented SVAN checkers. The rules cover the main check categories shown in Table 2.

Table 2. Categorization of synthesizability rules.

Category	Checker IDs	Example constructs
Constructs not used in synthesis	SYNTH02–SYNTH04, SYNTH06, SYNTH08–SYNTH11, SYNTH13–SYNTH15, SYNTH20, SYNTH21	Constructs with no physical equivalents: the === and !== operators, PLI and system functions, time delays, sensitivity lists in block bodies, non-constant loop bounds, non-constant operands of division and exponentiation, some types of UDPs
Semantics of hardware element descriptions	SYNTH01, SYNTH05, SYNTH18, SIGNAL09, SYNTH25	Correct description of ROM, RAM, FSM, latches, flip-flops, tri-state devices, combinational logic
Simulation divergence and correctness	SYNTH07, SYNTH12, SYNTH16, SYNTH17, SYNTH19, SYNTH22–SYNTH24, SYNTH26, DESIGN01, DESIGN05, LATCH05, SIGNAL05, RESET07, CASE05, CASE06	Constructs causing synthesis–simulation divergence as well as logical and other errors: combinational loops, sensitivity list misuse, blocking/non-blocking assignment errors, full_case and parallel_case coverage checks, x and z literal misuse

The first category unites constructs that have no physical equivalent and are intended for simulation only. The second category contains rules on how hardware elements (memories, registers, latches, finite state machines, tri-state devices) should be described so that synthesis tools infer the intended logic. The third category covers constructs that are formally acceptable for synthesis but can make the behavior of the synthesized netlist diverge from simulation or indicate likely design errors. Of the 34 initially formulated rules, thirteen belong to the first category, five to the second, and sixteen to the third.

Since those documents were published, Verilog has been extended by SystemVerilog and synthesis tools have expanded their capabilities. The described rules therefore required closer examination and adaptation to reflect current SystemVerilog and synthesis-tool capabilities. For example, IEEE/IEC 62142-2005 requires tasks and functions to be declared only with the `automatic` keyword; SystemVerilog provides more ways to control variable lifetime, so the corresponding rule was reformulated in terms of the lifetime of task and function members.

A separate clause of the standard is devoted to synthesis tool support for attributes. An investigation of this question showed that synthesis tools do not support many of the attributes described in the standard. The `full_case` and `parallel_case` attributes are supported by all of the tested tools, while `black_box`, `keep`, and `keep_hierarchy` are not supported by Genus [18]. Only Yosys supports the `logic_block`, `ram_block`, and `rom_block` attributes. At the same time, the `combinational`, `async_set_reset`, and `sync_set_reset` attributes are encountered in open-source projects, which makes checking their use practically relevant.

4. Conservativeness Configuration Mechanism

Language construct support varies across synthesis tools, so a fixed set of checks is either overly strict for some tools or insufficiently strict for others. To ensure portability across synthesis tools, a conservativeness-level configuration mechanism was implemented in the checkers. Conservativeness is understood here as the strength of the assumptions a checker makes about the target tool: the more conservative the configuration, the fewer constructs are assumed to be supported by that tool, so fewer real problems are missed, at the cost of warnings that would be unjustified for a more capable tool. For each rule involving constructs whose support depends on a particular synthesis tool, a strictness level is configurable via an optional integer checker parameter `strictLevel`. Level 0 corresponds to the strictest interpretation, closest to the requirements of the standard; higher levels progressively admit the additional forms of a construct that are supported by real synthesis tools. The number of levels differs between checkers. For rules whose exceptions can be proven statically, checkers additionally provide parameters that enable data-flow analysis (`allowSvanDFA`) and the use of initial values of variables in it (`allowInitial`). The following situations justify the need for such a mechanism.

4.1 Default values of variables

According to IEEE/IEC 62142-2005, default values of variables are not synthesizable and shall be ignored by synthesis tools. However, the Yosys and Vivado [17] tools take these values into account during synthesis (the Genus tool, by contrast, does not support initial values of variables). An example of using initial values in an RTL model and a fragment of the corresponding netlist generated by Yosys are shown in Listings 1 and 2.

```
reg r;
initial begin
    r = 1;
end
always @(posedge clk) begin
    if (rst)
        out <= 0;
    else
        out <= r;
end
```

Listing 1. An example of initial block usage.

```
cell $_DFF_P_ /* ... */
    connect \C \clk
    connect \D $0\out[0:0]
    connect \Q \out
end
/* ... $0\out is driven by \rst ... */
connect \r 1'1
```

Listing 2. Yosys synthesis result for Listing 1 (fragment).

Listing 2 shows that when `rst` is 0, the flip-flop input receives the value 1 taken from the `initial` block of the original model. Processing initial values can be enabled in the corresponding checkers.

4.2 Data-flow analysis

IEEE/IEC 62142-2005 restricts the synthesizability of `for` loops by requiring loop bounds to be statically computable. The reasonableness of this requirement is confirmed in practice: Yosys requires loop bounds to be constant, which is an even stricter restriction. However, the Vivado tool can synthesize more complex constructs than Yosys. An example of such a construct is given in Listing 3.


```
if (in == 4)
    for (int i = 0; i < in; ++i)
        out[i] = 1;
```

Listing 3. A for loop with a variable as its bound.

Vivado correctly determines the loop bounds by taking into account that the loop is nested in a conditional statement. For tools capable of synthesizing such constructs, some checkers can use data-flow analysis to prove that values are constant at the point of use.

Unlike programs written in programming languages for sequential execution on a processor, a hardware description describes the behavior of a circuit with many interconnected elements operating simultaneously. The data-flow analysis used in the checkers is therefore implemented at the level of procedural blocks, whose statements execute sequentially. The analysis collects information about the values of nets assigned by continuous assignments, values driven onto module input ports, and default values of variables (when allowed by the configuration), and uses these values when analyzing each procedural block separately.

4.3 Strictness of formulations

Requirements for the same construct differ in strictness across documents. IEEE/IEC 62142-2005 requires that “the power operator (`**`) shall be supported only when both operands are constants or if the first operand is 2”. An analysis of the corresponding SpyGlass rule showed that it has a less strict formulation: exponentiation is supported when the base can be statically evaluated and is a power of two; in addition, SpyGlass allows a statically non-computable base when the exponent is 0, 1, or 2. Exponentiation is generally hard to map to logic during synthesis, but if the base is a power of two, the operation reduces to a bit shift. Listing 4 shows a construct that was submitted to the Yosys and Vivado tools. The construct violates the constraint from the standard, yet both tools handled it correctly, synthesizing combinational logic corresponding to the original operator. Genus behaves similarly to Vivado on most such tests; for an unsupported exponentiation it reports an error listing the supported styles (const ** const, var ** const, (2^N) ** var). These examples are further confirmation of the need for flexible checker configuration.

```
input wire [3:0] in;
output reg [3:0] out;

always @(posedge clk) begin
    if (in)
        out <= 4 ** in;
end
```

Listing 4. The power operator (**) with a non-constant exponent.

4.4 Selecting parameter values for a target tool

To select checker parameter values matching a particular synthesis tool, a set of configuration modules was developed. Each module probes one capability of a synthesis tool that corresponds to a checker parameter; a comment inside the module states which synthesis result corresponds to which parameter value. A module is synthesized by the target tool, and the parameter value is determined from the resulting netlist. For example, the module in Listing 5 probes whether a tool takes initial values of variables into account: if the resulting netlist drives the `out` port with `4'b0111`, the tool supports them, and `allowInitial` can be enabled.

When synthesizing this module, Genus first warns that the initial value assignment is an unsynthesizable construct and is ignored, and then reports an error because the loop initialization expression does not evaluate to a constant; `allowInitial` should therefore be disabled for Genus. Vivado, by contrast, synthesizes the module taking the initial value into account (see Section 4.1), so `allowInitial` can be enabled. Similar modules probe the remaining parameters; e.g., a module whose loop bounds are computable only by data-flow analysis probes the `allowSvanDFA`

value. There are a few modules with different arguments of the power operator; these modules probe the value of the `strictLevel` parameter for the SYNTH15 checker. The purpose of the modules is to establish which checker options are meaningful for a given synthesis tool, not to reproduce its internal heuristics; the final choice of a configuration remains with the user. The recommended parameter values of the SYNTH15, SYNTH20, and SYNTH21 checkers (described in Section 5.2) derived in this way for the three synthesis tools are summarized in Table 3.

```
module top(input [3:0] in, output reg [3:0] out);
reg t = 1;

always @* begin
    out = 4'b0000;
    for (int i = t; i < 4; ++i)
        out[i] = 1;
end

endmodule
```

Listing 5. A configuration module for selecting the allowInitial parameter value.

Table 3. Recommended checker parameters for the synthesis tools.

Checker	Yosys	Genus	Vivado

SYNTH15	`strictLevel=1`
	`allowSvanDFA=false`
	`allowInitial=false`
		`strictLevel=3`
		`allowSvanDFA=true`
		`allowInitial=false`
			`strictLevel=3`
			`allowSvanDFA=true`
			`allowInitial=true`

SYNTH20	`allowSvanDFA=false` `allowInitial=false`	`allowSvanDFA=true` `allowInitial=false`	`allowSvanDFA=true` `allowInitial=true`

SYNTH21	`strictLevel=1`
	`allowSvanDFA=false`
	`allowInitial=false`
		`strictLevel=1`
		`allowSvanDFA=true`
		`allowInitial=false`
			`strictLevel=1`
			`allowSvanDFA=true`
			`allowInitial=true`

5. Implementation of Static Checkers

5.1 The SVAN static analysis system

SVAN is a static analysis system for SystemVerilog designed to check hardware descriptions for typical errors (violations of syntactic, semantic, and structural properties as well as safety properties) and to build an intermediate representation for use in simulators, synthesis tools, and simulation-based and formal verification frameworks [7]. At the time of writing, SVAN provides 114 static checkers for Verilog and SystemVerilog. The SVAN architecture is shown in Fig. 1.

SVAN provides infrastructure for creating static checkers that perform pattern matching of syntactic constructs via an abstract syntax tree (AST) visitor mechanism; the AST is built using the slang front-end [10]. Verification of an RTL model against the described rule set is performed by such checkers. Checkers are grouped into categories by one or more semantic features; the SYNTH category, containing synthesizability checkers, already existed in SVAN when this work started. In addition to AST analysis, SVAN implements data-flow analysis in the form of partial path interpretation with information collected along the paths; the collected information is used when issuing warnings at specific points of the design hierarchy. SVAN thus combines two analysis methods (AST traversal and data-flow analysis), which distinguishes it from open-source alternatives. The work was carried out as part of the svan-tidy analysis module.

New checkers were developed for ten rules (SYNTH14–SYNTH23), and the existing checker for one rule (SYNTH13) was updated. The remaining rules of the set are covered by previously implemented SVAN checkers: sensitivity lists (DESIGN05, SIGNAL05), combinational loops (DESIGN01, LATCH05), reset signal polarity (RESET07), the `full_case` and `parallel_case` attributes (CASE05, CASE06), case-equality operators (SYNTH10), the

structure of always blocks with asynchronous logic (SYNTH01), tri-state device descriptions (SIGNAL09), implicit sequential descriptions inside block bodies (SYNTH02, SYNTH03), functions returning `real` values (SYNTH04), multiple clock edges in sensitivity lists (SYNTH05), non-constant `repeat` expressions (SYNTH06), mixing assignment types within one procedural block (SYNTH07), consistency of loop condition, initialization, and step variables (SYNTH08, SYNTH09), comparisons with the high-impedance state (SYNTH12), and non-synthesizable types of user-defined primitives (SYNTH11).

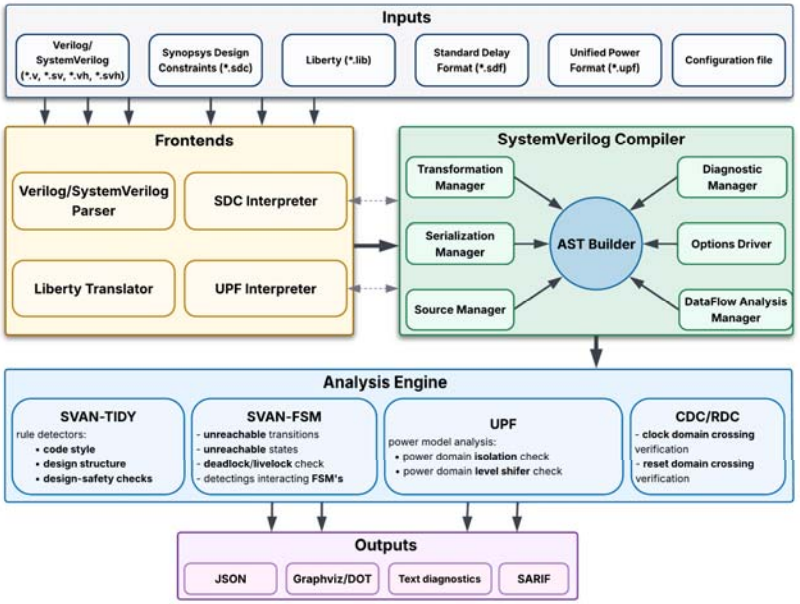

Fig. 1. Architecture of the SVAN static analysis system.

User-defined primitives (UDPs) serve mainly for simulation modeling of library cells and lie outside the commonly accepted synthesizable subset: tool support varies (Genus and Yosys reject them outright), so the checker treats combinational UDPs as synthesizable and warns on sequential ones that most tools cannot handle. Rules for initial blocks and statically bounded recursion do not yet have checker implementations (see Section 3.3). The new and updated checkers are described below.

5.2 Constructs not used in synthesis

5.2.1 SYNTH13

New checks were added to the SYNTH13 checker: prohibition of PLI and system functions (except `$signed` and `$unsigned`) and prohibition of pull gates, MOS switches, and bidirectional switches. These constructs shall not be supported for synthesis according to IEEE/IEC 62142-2005 and must be used only for simulation purposes. The rule also checks that the block identifier in a `disable` statement matches the identifier of its immediately enclosing block, as required by the standard. An example of these constructs is shown in Listing 6.

To check a `disable` statement, the checker tracks the sequential blocks enclosing it and verifies that the block being disabled is the innermost one. Expressions assigned to parameters are ignored:

they are evaluated during elaboration and never reach logic synthesis, and system function calls are frequent in them, so reporting them would make the checker noisy.

```
reg r;
always @(*) begin : block
    int i;
    i = $random;    // Violation
    disable block;  // No violation
end

wire w1, a, b;
pmos p(w1, a, b);   // Violation
```

Listing 6. An example of constructs checked by the SYNTH13 checker.

5.2.2 SYNTH14

According to IEEE/IEC 62142-2005, sensitivity lists shall be used only for modeling combinational procedural blocks, latches, and flip-flops. In all these cases, sensitivity lists appear immediately after the `always` keyword. Furthermore, the absence of a sensitivity list in an always block header can lead to an infinite loop during simulation. The SYNTH14 checker verifies that a procedural block contains exactly one sensitivity list in its header and none in its body, including inside task calls. Listing 7 shows both a non-synthesizable sensitivity list (an event control inside the block body) and a synthesizable one (in the block header).

```
reg r1, r2;
always begin
    @(posedge clk) r1 <= 1;  // Violation
end

// No violation
always @(posedge clk) begin
    r2 <= 1;
end
```

Listing 7. An example of constructs checked by the SYNTH14 checker.

The checker verifies that a procedural block has a sensitivity list in its header and reports the lists found in its body, including those in called tasks. Synthesis tools handle such constructs differently: Yosys reports a syntax error for an event control inside a block body, whereas Genus and Vivado can synthesize a block whose only event control is the first statement of the body but fail on subsequent event controls.

5.2.3 SYNTH15

The SYNTH15 checker forbids the exponentiation operator (`**`) when its operands do not meet the requirements of the selected strictness level. IEEE/IEC 62142-2005 permits the `**` operator only when both operands are constants or the first operand is 2. Taking into account the less strict SpyGlass formulation (see Section 4.3), several operating modes with requirements of varying strictness were implemented for SYNTH15; the forms of the operator allowed at each level are shown in Table 4.

*Table 4. Forms of the ** operator allowed at each strictness level of the SYNTH15 checker.*

strictLevel	Allowed forms of the ** operator
0	`CONST ** CONST`
1	`CONST ** CONST, 2 ** n`
2	`CONST ** CONST, (2^K) ** n`
3	`CONST ** CONST, (2^K) ** n, n ** 0, n ** 1, n ** 2`

In addition to the strictness level, the checker can use data-flow analysis to prove the operand properties required at the selected level; initial values of variables can also be taken into account in the analysis (this setting has no effect when data-flow analysis is disabled). The depth of loop unrolling and the number of paths stored during the analysis are bounded to prevent combinatorial explosion.

The recommended SYNTH15 configurations for the three synthesis tools are given in Table 3 (see Section 4.4).

5.2.4 SYNTH20

The SYNTH20 checker verifies that `for`-loop bounds are statically computable, as required by IEEE/IEC 62142-2005: a loop whose bounds cannot be computed statically cannot be unrolled and synthesized. By default, only loops with constant bounds are allowed; loops whose bound constancy is proven by data-flow analysis can additionally be permitted, and initial values of variables can be taken into account in the analysis. The construct in Listing 3 (see Section 4.2) is synthesized by Vivado but rejected by Yosys. The recommended settings for the three tools are given in Table 3. The analysis may give an incorrect result if the loop counter is modified inside the loop body; the slang front-end provides related diagnostics for detecting such loops, since some synthesis tools may mishandle them.

5.2.5 SYNTH21

The SYNTH21 checker forbids the division (`/`) and modulo (`%`) operators when their operands do not meet the requirements of the strictness level: at level 0 only the `CONST / CONST` and `CONST % CONST` forms are allowed, and at level 1, following the SpyGlass STARC02-2.10.6.6 rule, division and modulo by a power of two are also allowed. The checker supports the same data-flow analysis settings as SYNTH15. Listing 8 shows an operator considered synthesizable at strictness level 1 with data-flow analysis enabled: the equality of the divisor `t` to the constant 2 is proven by the analysis.

```
input [3:0] in;
output reg [3:0] out;
reg [3:0] t;

always @* begin
    if (t == 2)
        out = in / t;  // strictLevel = 1 and allowSvanDFA = true:
                       // no violation - divisor equality to 2
                       // proven by data-flow analysis
end
```

Listing 8. An example of a construct checked by the SYNTH21 checker.

Synthesis experiments showed that Yosys, Vivado, and Genus can synthesize the division and modulo operators even when both operands are statically unknown, so the checker may be disabled entirely for these tools. In the recommended configurations (Table 3) it is kept enabled at strictness level 1, which permits division and modulo by a power of two.

5.3 Semantics of hardware element descriptions

5.3.1 SYNTH18

The SYNTH18 checker restricts blocking assignments in sequential logic: they are permitted only for temporary variables used exclusively within the given procedural block. Such variables are used to model combinational inputs of sequential elements; in other cases, blocking assignments can lead to race conditions. Temporary variables are understood as variables declared in the procedural block

with automatic lifetime: IEEE/IEC 62142-2005 allows blocking assignments only for intermediate variables that are assigned and used within a single always statement. This requirement can be relaxed by increasing the strictness level, in which case static variables used in a single procedural block and nowhere else are also considered temporary. Listing 9 shows an example construct addressed by this checker: the blocking assignment to the temporary variable is not a violation if the strictness level is 1 and the variable `t` is not used anywhere outside this procedural block; otherwise the blocking assignment is a violation.

```
input in;
output reg [3:0] out;
reg [3:0] t;

always @(posedge clk) begin
    t = in + 1;  // strictLevel = 1: no violation;
                 // strictLevel = 0: violation
    out <= t;
end
```

Listing 9. An example of a construct checked by the SYNTH18 checker.

The checker examines the lifetime of the variables written by blocking assignments in sequential blocks, including assignments made in tasks called from them. At strictness level 0 a warning is issued for all static variables; at level 1 the checker additionally records which static variables are used in which procedural blocks in order to identify temporary ones. The checker conforms to IEEE/IEC 62142-2005 at strictness level 1; nevertheless, strictness level 0 is recommended in practice, because static variables in sequential logic can lead to the synthesis of unintended latches or flip-flops.

5.4 Simulation divergence and correctness

5.4.1 SYNTH16

Blocking assignments should be used to model combinational logic, and non-blocking assignments to model sequential logic (latches and flip-flops) [19]. This separation is necessary to avoid race conditions during simulation; IEEE/IEC 62142-2005 explicitly forbids assigning to the same variable with both blocking and non-blocking assignments within one module. The SYNTH16 checker finds and reports such assignments. Listing 10 shows a construct causing a race condition: when `rst` is asserted, it cannot be determined unambiguously what happens at the clock edge – the assignment of 0 to `out` or the reading of its value for use in the addition operator.

```
always @(posedge clk) begin
    out <= out + 1;
end

always @(posedge clk) begin
    if (rst)
        out = 0;
end
```

Listing 10. An example of a construct checked by the SYNTH16 checker: a race condition in flip-flop modeling.

The checker records the assignment type used for each bit of a variable, taking module ports and task arguments into account, and warns when the same bit is assigned in both ways. Argument passing to tasks and functions is treated as a blocking assignment to the corresponding variable (see Section 6.3).

5.4.2 SYNTH17

The SYNTH17 checker verifies that the high-impedance value (`z`) does not propagate through chains of variable assignments, as required by IEEE/IEC 62142-2005. The `z` value has a different meaning in synthesis than in simulation, so its propagation through variable assignments may lead to divergence from simulation. In Listing 11, the `z` value must not reach the `out` variable; in this situation the checker issues a warning.

```
reg tmp;
always @(in) begin
    if (set)
        out = 1'b1;
    else begin
        tmp = 1'bz;
        out = tmp;  // Violation: z propagation through tmp
    end
end
```

Listing 11. An example of a construct checked by the SYNTH17 checker.

The checker builds a directed graph whose nodes are variables and whose edges reflect assignments between them, including continuous assignments of `z`, task arguments, and module port connections. The value is then propagated over this graph, and a warning is issued when it reaches a variable that appears on the right-hand side of an assignment. Whether default values of variables are taken into account is configurable: for closer conformance to IEEE/IEC 62142-2005 they should be ignored, since initial values of variables are not synthesizable from the standard's point of view, and they can be taken into account when the target synthesis tool supports them (see Section 4.1).

5.4.3 SYNTH19

The SYNTH19 checker restricts the use of the `x` and `z` literals in a design. According to IEEE/IEC 62142-2005, the `x` value may be used on the right-hand side of an assignment to represent a don't-care value for synthesis and in case item expressions of a `casex` statement; the `z` value may be used on the right-hand side of an assignment to infer a tri-state driver and in case item expressions of `casex` and `casez` statements. In all other contexts (in particular, in combination with any operators, mixed into other expressions, or in selector expressions) these literals are forbidden, and the checker issues a warning. Listing 12 shows a forbidden z literal in a casex selector expression and a forbidden x literal in a casez item expression.

```
reg r;
always @* begin
    casex (1'bz) // Violation: z literal in a selector expression
        1'bz: r = 1;
        1'bx: r = 1;
    endcase
    casez (in)
        1'bx: r = 1; // Violation: x literal in a casez item
    endcase
end
```

Listing 12. An example of constructs checked by the SYNTH19 checker.

5.4.4 SYNTH22

The SYNTH22 checker forbids members of tasks and functions with static lifetime: static internal variables may require preserving state between calls, which is not supported by some synthesis tools. IEEE/IEC 62142-2005 requires tasks and functions to be declared with the non-optional `automatic` keyword; since SystemVerilog provides more ways to control variable lifetime, the rule was reformulated in terms of the lifetime of task and function members (see Section 3.3). In

Listing 13, the static variable `r` is expected to behave as a flip-flop with a synchronous reset: the task logic executes at the clock edge, and `r` retains its value between calls. However, Yosys does not generate the corresponding flip-flop and issues no warning (see Section 6.2).

```
always @(posedge clk) begin
    t(out);
end
task t(output q);
    reg r;  // Violation: r is static
    begin
        if (rst)
            r <= 0;
        else if (en)
            r <= ~r;
        q = r;
    end
endtask
```

Listing 13. An example of a construct checked by the SYNTH22 checker.

5.4.5 SYNTH23

The SYNTH23 checker verifies that the attributes used in a design are supported by the target synthesis tool. Synthesis tool vendors provide substantial sets of non-standard attributes, and their support varies significantly between tools; using an unsupported attribute typically has no effect on synthesis and gives no error, silently discarding the designer's intent (see Section 4.1). Listing 14 shows a `black_box` attribute attached to a module, which is a construct supported by Yosys and Vivado.

```
(* synthesis, black_box *) // Violation: not supported by Genus
module add2 (input [7:0] dataa, datab, input cin,
             output [7:0] result, output cout, overflow);
endmodule
```

Listing 14. An example of an attribute checked by the SYNTH23 checker.

The target tools are passed to the checker via the `targetTools` parameter; if it is not specified, all three tools are used. Attributes attached to modules, instances, statements, nets, registers, and port connections are resolved against an internal support table, and a warning is issued if at least one of the target tools does not support the attribute.

6. Evaluation and Results

6.1 Test suite

For evaluation, a test suite was assembled containing positive and negative examples: positive tests contain correct synthesizable constructs, and negative tests contain constructs violating the corresponding rule. For the eleven implemented checkers, the suite contains 351 examples of synthesizable and non-synthesizable constructs (Table 5); for the rules that do not have checker implementations, one negative example each was created.

6.2 Synthesis tool behavior on synthetic tests

A part of the developed tests was submitted to Yosys to analyze its behavior when processing non-synthesizable constructs. The results showed that certain constructs in the netlist generated by Yosys have semantics different from the original. In `casex` and `===` operators, bits equal to `x` are treated as don't-care values, which differs from the behavior in simulation. An example of such code and the resulting netlist is given in Listings 15 and 16.

Table 5. Tests developed for the implemented checkers.

Checker	Negative tests	Positive tests	Total
SYNTH13	24	10	34
SYNTH14	13	8	21
SYNTH15	11	3	14
SYNTH16	8	9	17
SYNTH17	13	11	24
SYNTH18	33	21	54
SYNTH19	28	32	60
SYNTH20	9	3	12
SYNTH21	12	6	18
SYNTH22	8	8	16
SYNTH23	78	3	81
Total	237	114	351

```
always @* begin
    if (state === 2'b1x)
        err = 1;
    else
        err = 0;
end
```

Listing 15. The case equality operator with an x bit.

```
module \top
    wire width 2 input 1 \state
    wire output 2 \err
    connect \err \state [1]
end
```

Listing 16. Yosys synthesis result for Listing 15.

During simulation, the `===` operator in Listing 15 returns true only if the value of `state[0]` is `x`, whereas in the netlist the `err` signal is connected to `state[1]`: if `state` holds the value `2'b11`, `err` is assigned 0 in simulation but 1 in the synthesized circuit.

Yosys also silently resolves a variable assigned with both blocking and non-blocking assignments (the SYNTH16 rule): the output is permanently driven by the blocking assignment, whereas in simulation the non-blocking one takes effect at the end of the simulation cycle.

Attempts to use static variables declared in tasks (the SYNTH22 rule, see Section 5.4.4) also produce results that differ from those expected in simulation, and Yosys issues no warnings in these cases. A similar mismatch was observed with Vivado: in simulation, a static variable declared in a task retained its value between calls and behaved as a latch, whereas the synthesized netlist contained only combinational logic without state-holding elements. The comparative analysis of Yosys on the test suite also revealed cases where the tool successfully synthesized constructs formally outside the IEEE/IEC 62142-2005 synthesizable subset (e.g., default values of variables, see Section 4.1) and cases where non-synthesizable constructs were processed without any diagnostic messages. These findings were used to configure the conservativeness mechanism of the checkers. The results

confirm the relevance of the developed tool: static analysis detects potential problems before synthesis and provides the developer with detailed diagnostics.

6.3 Comparison with SpyGlass

The developed checkers were compared with SpyGlass on seven of the implemented rules for which semantically comparable SpyGlass rules could be identified: SYNTH13–SYNTH16, SYNTH18, SYNTH20, and SYNTH21. Both tools were run on the same set of synthetic tests developed specifically for this comparison (separate from the test suite described in Section 6.1); precision is the share of true violations among the reported warnings, and recall is the share of the violations present in the tests that were reported. The comparison results are shown in Table 6.

Table 6. Comparison of SVAN and SpyGlass checkers on synthetic tests.

Checker	Warnings, SVAN	Warnings, SpyGlass	Precision, SVAN	Precision, SpyGlass	Recall, SVAN	Recall, SpyGlass
SYNTH13	20	20	100%	100%	100%	100%
SYNTH14	4	4	100%	100%	100%	100%
SYNTH15	4	7	100%	42.9%	100%	75%
SYNTH16	11	5	100%	100%	100%	46%
SYNTH18	19	5	100%	100%	100%	26.3%
SYNTH20	6	5	100%	100%	100%	83.3%
SYNTH21	5	9	100%	44.4%	100%	80%

On all seven rules the developed checkers reported every violation present in the tests without false positives, achieving 100% precision and recall; the number of violations seeded in the tests is therefore equal to the number of warnings reported by SVAN in Table 6. SpyGlass matched this result only on SYNTH13 and SYNTH14. The lower SpyGlass precision and recall on the operator-related rules (SYNTH15 and SYNTH21) are caused by the fact that its analysis does not take into account the values connected to module input ports and does not support modules whose input ports have default values. The lower recall on the assignment-type rules (SYNTH16 and SYNTH18) is caused by the fact that SpyGlass does not treat argument passing to tasks as a blocking assignment, although according to the SystemVerilog standard (IEEE 1800-2023, Section 13.3), passing input arguments to a task is equivalent to a blocking assignment to the task's internal variable, and passing output arguments is equivalent to a blocking assignment to the passed expression. Thus, the construct in Listing 17 constitutes two assignment types to the same variable (the SYNTH16 rule), yet SpyGlass does not flag a violation on it.

```
reg r;
always @(posedge clk) begin
    r <= 1;  // Non-blocking assignment to r
    t(r);    // On completion: blocking assignment r = t.q
end

task t(output q);
    q = 1;
endtask
```

Listing 17. An example of a construct that violates the SYNTH16 rule.

SpyGlass also does not take into account assignments made inside tasks to variables external to them, and its checker for mixed assignment types performs analysis only within a single procedural block. For the remaining checkers, no SpyGlass rules with comparable semantics were identified; for example, no SpyGlass counterpart was found for the SYNTH17 rule prohibiting z-value propagation. It should be noted that the synthetic tests were developed by the authors together with the checkers, which may bias the comparison in favor of the developed tool; a comparison on independent test suites is planned as part of future work.

6.4 Testing on open-source projects

The developed checkers were run on the ten open-source hardware projects used for SVAN testing [7]: aes [20], OpenC910 [21], CVA6 [22], Ibex [23], OpenTitan [24], PicoRV32 [25], VeeR EH1 [26], VeeR EL2 [27], SCR1 [28], and BlackParrot [29]. Table 7 shows the numbers of reported warnings, counted only in the parts of the projects intended for synthesis; the rules are referred to by their checker identifiers (see Section 5), and the checkers that reported nothing anywhere are combined into a single row. The counts are given for the configurations matching the tested synthesis tools (see Section 4.4); for SYNTH18 and SYNTH21 these correspond to strictness level 1, since the stricter level 0 additionally reports constructs that are synthesizable but discouraged by good coding practice. Enabling data-flow analysis changes the counts of SYNTH15, SYNTH17, and SYNTH20 by at most one warning. The shaded cells mark the counts obtained in the configuration corresponding to the synthesis tool known for the project from publicly available sources (aes – Quartus Prime; OpenTitan and BlackParrot – Synopsys Design Compiler; VeeR EH1, PicoRV32, and Ibex – Yosys; CVA6 – Vivado and Quartus Prime); for the rules whose configuration does not depend on the target tool, the cells are shaded for all projects. The remaining cells give the counts obtained in the default, strictest configuration; for SYNTH17, whose configurations are not ordered by strictness, initial values of variables are ignored.

Table 7. Warnings reported by the developed checkers on open-source projects.

Rule	aes	OpenC910	CVA6	Ibex	OpenTitan	PicoRV32	VeeR EH1	VeeR EL2	SCR1	BlackParrot
SYNTH13	0	0	28	42	19	1	38	137	156	144
SYNTH14–17, SYNTH19	0	0	0	0	0	0	0	0	0	0
SYNTH18	0	24	0	1	1	0	0	0	1	0
SYNTH20	3	15	1	4	6	1	0	14	0	0
SYNTH21	0	0	0	0	3	0	0	0	0	0
SYNTH22	0	0	0	0	0	0	8	1	0	0
SYNTH23	0	0	0	0	2	0	0	0	0	0

Violations of the formulated rules were found in all ten projects; all reported warnings were manually verified and confirmed to be true positives. A warning indicates a genuine violation of the corresponding rule, whereas whether the construct causes a problem in a particular design flow depends on the synthesis tool used for the project. In particular, blocking assignments in sequential logic (the SYNTH18 rule) were detected in the OpenC910 project; a simplified excerpt illustrating this violation is shown in Listing 18.

```
integer i;
always @(posedge dsqclk or negedge cpurst_b) begin
    if (!cpurst_b)
        for (i = 0; i < DSQ_DEPTH; i = i + 1)
            dsq_vld[i] <= 1'b0;
    else /* ... */
end

always @(posedge dsqclk or negedge cpurst_b) begin
    if (!cpurst_b)
        for (i = 0; i < DSQ_DEPTH; i = i + 1)
            dsq_apb[i] <= 1'b0;
    else /* ... */
end
```

Listing 18. An excerpt from the OpenC910 project that violates the SYNTH18 rule.

The variable `i` is static and is used in two procedural blocks with the same sensitivity list; concurrent writes to it in the two blocks can lead to a race condition (the SystemVerilog standard explicitly warns that sharing a loop control variable between parallel processes may lead to one loop modifying the variable while the other is still using it [2]). The issue was reported to the project developers [30].

Recommended checker configurations are currently derived only for Yosys, Vivado, and Genus (see Section 4.4); deriving configurations for Quartus Prime and Synopsys Design Compiler is left for future work.

6.5 Analysis time and memory consumption

Table 8 reports the size of each design, the peak memory consumption, and the total running time of the eleven implemented synthesizability checkers on the same ten projects. Memory is the peak resident set size of the whole analysis process, which includes parsing and elaboration, whereas the reported time covers only the execution of the checkers themselves. Two configurations were measured: with data-flow analysis disabled in all checkers and with data-flow analysis and initial values enabled in the checkers that support them. Several checkers have more than one parameter combination within each of these configurations, so the time is given as an interval whose bounds are the sums over the eleven checkers of the smallest and the largest per-checker running times. Each measurement is a single-threaded run performed once on a machine with a six-core AMD Ryzen 5 4600G processor (12 threads, up to 4.3 GHz) and 16 GB of RAM running Ubuntu 24.04 LTS.

Table 8. Design sizes, peak memory, and total running time of the synthesizability checkers.

Project	Size, kLOC	Peak memory, MB	Time, s (data-flow analysis disabled)	Time, s (data-flow analysis enabled)
aes	4.6	68	< 0.01	< 0.01
PicoRV32	7.8	38	< 0.01	< 0.01
SCR1	16.9	180	0.01–0.02	0.20–0.22
VeeR EH1	21.2	355	0.12–0.15	0.61–0.64
VeeR EL2	51.5	1836	0.41–0.46	6.71–6.87
Ibex	85	148	0.01–0.02	0.10–0.12
BlackParrot	148.2	589	0.23–0.25	2.74–2.77
CVA6	270.5	9256	0.26–0.30	101.52–101.76
OpenC910	406	2390	3.13–3.37	3.13–3.37
OpenTitan	839	12527	1.97–2.22	300.34–301.71

With data-flow analysis disabled, the total time of all eleven checkers does not exceed four seconds even on the largest projects. Enabling data-flow analysis noticeably increases the analysis time on projects with complex loop structures: on OpenTitan and CVA6, almost all of the time is spent in the SYNTH20 loop-bound analysis (about 298 and 73 seconds, respectively) and, on CVA6, in SYNTH21 (about 27 seconds). Peak memory consumption ranges from about 38 MB on PicoRV32 to about 12.5 GB on OpenTitan.

6.6 Limitations

The presented approach has several limitations. Static checking does not guarantee the absence of synthesizability issues: the rule set does not cover all tool-specific restrictions, and the rules for the initial blocks and recursion bounds do not have checker implementations yet. The data-flow analysis is performed at the level of individual procedural blocks: although it uses the values of continuous assignments and module input ports, it does not perform full inter-procedural analysis, so some provable properties remain unproven and the checkers behave conservatively in such cases. The checkers do not currently interpret the guards commonly used to exclude simulation-only code from synthesis, such as the `` `ifdef SYNTHESIS `` directives and the `translate_off` pragmas, so

constructs placed inside such regions are analyzed as well. The recommended configurations reflect the capabilities of the synthesis tool versions used in this work and may need to be re-derived for other versions. Finally, precision was measured on synthetic tests and, for the open-source projects, by manual inspection of the reported warnings, whereas recall on real projects (the share of missed violations) has not been evaluated yet.

7. Conclusion

Based on the IEEE/IEC 62142-2005 synthesizability standard, the STARC RTL Design Style Guide, the Reuse Methodology Manual, and the rules of the commercial SpyGlass tool, a set of 34 synthesizability rules for Verilog and SystemVerilog hardware descriptions was formulated for static checkers of the SVAN static analysis system; the list was subsequently extended with further rules covered by previously implemented SVAN checkers. New checkers were developed for ten rules (SYNTH14–SYNTH23), and the existing checker for one rule (SYNTH13) was extended. The proposed conservativeness configuration mechanism adapts checker behavior to the capabilities of a specific synthesis tool; recommended configurations were derived for Yosys, Vivado, and Genus.

The evaluation on Yosys confirmed the practical relevance of the solution: cases were identified where the tool synthesized constructs formally outside the synthesizable subset, and where non-synthesizable constructs were processed without diagnostics and with altered circuit semantics. The comparison with SpyGlass on the seven rules with semantically comparable counterparts revealed specific gaps in its analysis, reflected in its precision of 43–100% and recall of 26–100% on these tests against 100% for the developed checkers. Testing on ten open-source projects revealed violations in every project examined, all manually confirmed as true positives; with data-flow analysis disabled, the total running time of the checkers does not exceed four seconds even on the largest projects.

Future work includes implementing checkers for the remaining rules (initial blocks and statically bounded recursion), taking into account the guards that exclude simulation-only code from synthesis, extending the rule set to account for the specifics of other synthesis tools, conducting a quantitative evaluation of analysis precision and recall on open-source projects, and continuing the comparison with existing solutions such as SpyGlass.

Список литературы / References

[1]. IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006, pp. 1-590. DOI: 10.1109/IEEESTD.2006.99495.

[2]. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017), 2024, pp. 1-1354. DOI: 10.1109/IEEESTD.2024.10458102.

[3]. IEEE Standard for Verilog Register Transfer Level Synthesis. IEC 62142:2005 / IEEE Std 1364.1-2002, 2005. 116 p.

[4]. RTL Design Style Guide for Verilog HDL. Semiconductor Technology Academic Research Center (STARC).

[5]. Keating M., Bricaud P. Reuse Methodology Manual for System-on-a-Chip Designs. Springer, 2002.

[6]. Yosys Open SYnthesis Suite. Available at: https://yosyshq.readthedocs.io/projects/yosys/en/latest/, accessed 01.06.2026.

[7]. Чуркин Я.А., Бучацкий Р.А., Китаев К.Н., Волохов А.Г., Долгодворов Е.В., Камкин А.С., Коцыняк А.М., Самоваров Д.О. Система статического анализа для языка описания аппаратуры SystemVerilog. Труды ИСП РАН, 2025, том 37, вып. 1, стр. 7-40. DOI: 10.15514/ISPRAS-2025-37(1)-1. / Churkin Ya.A., Buchatskiy R.A., Kitaev K.N., Volokhov A.G., Dolgodvorov E.V., Kamkin A.S., Kotsynyak A.M., Samovarov D.O. System for Static Analysis of SystemVerilog HDL. Trudy ISP RAN/Proc. ISP RAS, 2025, vol. 37, issue 1, pp. 7-40 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-1.

[8]. Verible – SystemVerilog parser, linter, and formatter. Available at: https://github.com/chipsalliance/verible, accessed 01.06.2026.

[9]. svlint – SystemVerilog linter. Available at: https://github.com/dalance/svlint, accessed 01.06.2026.

[10]. Popoloski M. slang – SystemVerilog Language Services. Available at: https://github.com/MikePopoloski/slang, accessed 01.06.2026.
[11]. SpyGlass: Early Design Analysis Tools for SoCs. Available at: https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html, accessed 01.06.2026.
[12]. Meredith P., Katelman M., Meseguer J., Roşu G. A Formal Executable Semantics of Verilog. In Proc. of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign, 2010, pp. 179-188. DOI: 10.1109/MEMCOD.2010.5558634.
[13]. Lööw A. The Simulation Semantics of Synthesisable Verilog. Proceedings of the ACM on Programming Languages, 2025. DOI: 10.1145/3720484.
[14]. Lööw A. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In Proc. of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2021. DOI: 10.1145/3437992.3439916.
[15]. Herklotz Y., Wickerson J. Finding and Understanding Bugs in FPGA Synthesis Tools. In Proc. of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2020. DOI: 10.1145/3373087.3375310.
[16]. ALINT-PRO. Available at: https://www.aldec.com/en/products/functional_verification/alint-pro, accessed 01.06.2026.
[17]. AMD Vivado Design Suite. Available at: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html, accessed 01.06.2026.
[18]. Genus Synthesis Solution. Cadence Design Systems. Available at: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, accessed 01.06.2026.
[19]. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them. Springer, 2010. 218 p.
[20]. AES – Hardware implementation of AES encryption algorithm. Available at: https://github.com/secworks/aes, accessed 01.06.2026.
[21]. OpenXuantie – OpenC910 Core. Available at: https://github.com/XUANTIE-RV/openc910, accessed 01.06.2026.
[22]. CVA6 – an application class 6-stage RISC-V CPU. Available at: https://github.com/openhwgroup/cva6, accessed 01.06.2026.
[23]. Ibex – a small 32-bit RISC-V CPU core. Available at: https://github.com/lowRISC/ibex, accessed 01.06.2026.
[24]. OpenTitan – open source silicon root of trust. Available at: https://opentitan.org/, accessed 01.06.2026.
[25]. PicoRV32 – a size-optimized RISC-V CPU. Available at: https://github.com/YosysHQ/picorv32, accessed 01.06.2026.
[26]. VeeR EH1 Core. Available at: https://github.com/chipsalliance/Cores-VeeR-EH1, accessed 01.06.2026.
[27]. VeeR EL2 Core. Available at: https://github.com/chipsalliance/Cores-VeeR-EL2, accessed 01.06.2026.
[28]. SCR1 – an open-source RISC-V MCU core. Available at: https://github.com/syntacore/scr1, accessed 01.06.2026.
[29]. BlackParrot – a Linux-capable RISC-V multicore. Available at: https://github.com/black-parrot/black-parrot, accessed 01.06.2026.
[30]. OpenC910, GitHub issue #52. Available at: https://github.com/XUANTIE-RV/openc910/issues/52, accessed 01.06.2026.

Информация об авторах / Information about authors

Алексей Алексеевич ШЕВЦОВ – старший лаборант отдела компиляторных технологий ИСП РАН, студент ВМК МГУ имени М.В. Ломоносова. Научные интересы: статический анализ программ, языки описания аппаратуры, компиляторные технологии.

Aleksey Alekseevich SHEVTSOV – senior laboratory assistant at Compiler Technology department of ISP RAS and student at the CMC Faculty, Lomonosov MSU. Research interests: static program analysis, hardware description languages, compiler technologies.

Ян Андреевич ЧУРКИН – научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации, языки описания аппаратуры.

Yan Andreevich CHURKIN – researcher at the Compiler Technology department of ISP RAS. Research interests: static program analysis, compiler technologies, optimizations, hardware description languages.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), senior researcher at the Compiler Technology department of ISP RAS. Research interests: static program analysis, compiler technologies, optimizations.

Александр Сергеевич КАМКИН – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий программирования ИСП РАН. Научные интересы: формальные методы, синтез и верификация цифровой аппаратуры, гетерогенные компьютерные системы.

Alexander Sergeevich KAMKIN – Cand. Sci. (Phys.-Math.), leading researcher at the Software Engineering department of ISP RAS. Research interests: formal methods, synthesis and verification of digital hardware, heterogeneous computer systems.

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: компиляторные технологии, оптимизации.

Dmitry Mikhailovich MELNIK – senior researcher at the Compiler Technology department of ISP RAS. Research interests: compiler technologies, optimizations.