DOI: 10.15514/ISPRAS-2025-37(1)-1

Система статического анализа для языка описания аппаратуры SystemVerilog

[1] *Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,3] *К.Н. Китаев, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>*
[1] *А.Г. Волохов, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>*
[3] *Е.В. Долгодворов, ORCID: 0000-0001-6962-6448 <dolgodvorov.ev@phystech.edu>*
[1,2,3,4] *А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *А.М. Коцыняк, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,2] *Д.О. Самоваров, ORCID: 0009-0000-2428-6654 <dmitrysamovarov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*
[3] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*
[4] *Российский экономический университет имени Г.В. Плеханова, 117997, Россия, г. Москва, Стремянный переулок, д. 36.*

Аннотация. Рост сложности современных цифровых систем и увеличение объемов кода на языках описания аппаратуры требуют эффективных инструментов для выявления ошибок на ранних этапах разработки цифровых СБИС. Для своевременного обнаружения ошибок составляются сборники правил, регламентирующие описание аппаратуры. Эти сборники содержат набор правил, описывающих неточности, ошибки и последствия их нарушения. В данной работе рассмотрен список правил, разработанный на основе опыта работы инженеров, использующих язык SystemVerilog, и представлена система статического анализа SVAN, разработанная для языка SystemVerilog и учитывающая специфику описаний аппаратуры. Предлагаемая система обеспечивает полную поддержку стандарта SystemVerilog IEEE 1800-2017 и предоставляет возможности анализа описаний на наличие структурных и семантических ошибок.

Ключевые слова: статический анализ; язык описания аппаратуры; язык описания аппаратуры Verilog; язык описания аппаратуры SystemVerilog; интегральная схема; система автоматизации проектирования (САПР); инструментарий анализа, компиляции и выполнения описаний аппаратуры slang, slang-tidy, KLEE, Yosys, Verilator, CIRCT; проект LLVM; абстрактное синтаксическое дерево АСД.

Для цитирования: 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, том 37, вып. 1, 2025 г., стр. 7–40. DOI: 10.15514/ISPRAS–2025–37(1)– 1.

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

System for Static Analysis of SystemVerilog HDL

[1] *Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,3] *K.N. Kitaev, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>*
[1] *A.G. Volokhov, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>*
[3] *E.V. Dolgodvorov, ORCID: 0000-0001-6962-6448 <dolgodvorov.ev@phystech.edu>*
[1,2,3,4] *A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *A.M. Kotsynyak, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,2] *D.O. Samovarov, ORCID: 0009-0000-2428-6654 <dmitrysamovarov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.
[4] *Plekhanov Russian University of Economics,*
36 Stremyanny lane, Moscow, 117997, Russia.

Abstract. The growing complexity of modern digital systems and the increasing volumes of code written in hardware description languages demand effective tools for early error detection in the development of digital ASICs. To facilitate timely error detection, rule sets are created to regulate hardware descriptions. These rule sets contain a collection of rules that describe inaccuracies, errors, and the consequences of their violation. This paper discusses a list of rules developed based on the experience of engineers using the SystemVerilog language and presents the SVAN static analysis system, designed for SystemVerilog and tailored to the specifics of hardware descriptions. The proposed system provides full support for the SystemVerilog IEEE 1800-2017 standard and offers capabilities for analyzing descriptions for structural and semantic errors.

Keywords: static analysis; HDL; Verilog; SystemVerilog; IC; CAD system; analysis, compilation and simulation tools slang, slang-tidy, KLEE, Yosys, Verilator, CIRCT; LLVM project; abstract syntax tree AST.

For citation: Churkin Y.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, vol. 37, issue 1, 2025, pp. 7-40 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-1.

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 “SAPR-Analiz” (as part of the R&D project “SAPR mikroelektronika”, the main contractor is JSC “ISTC MIET”).

1. Введение

Проектирование цифровых интегральных схем является многоэтапным процессом [1], включающим спецификацию на языке описания аппаратуры (HDL, Hardware Description Language), разработку архитектуры, логический и физический синтез. Каждый этап включает верификацию, нацеленную на проверку соответствия полученного описания результату предшествующего этапа или техническому заданию, при этом исправление ошибок, обнаруженных на поздних этапах, требует значительных временных и экономических затрат. Ошибки, выявленные после внедрения изделий, могут потребовать внесения правок в проектную модель, повторного выполнения синтеза и перевыпуска микросхем. Примерами таких ошибок в больших интегральных схемах могут служить:

1. Ошибка Pentium Fdiv [2], 1994 год. При делении чисел с плавающей точкой происходила ошибка в модуле, выполняющем эту операцию. Ее причиной послужили неточности в таблице поиска, используемой при проведении операции

деления. Исправление этой ошибки требовало замены процессора и привело к убыткам в 475 миллионов долларов для компании Intel.

2. Группа уязвимостей Spectre [3], 2017 год. Ошибка в процессорах, использующих спекулятивное выполнение для получения доступа к виртуальной памяти процесса. Нейтрализация данной уязвимости требует обнаружения определенной последовательности команд в коде программ. В настоящее время для решения этой проблемы используется дополнительный микрокод или патчи операционной системы.

В данной работе рассматривается этап описания цифровой аппаратуры, на котором происходит его реализация на соответствующих языках. Описания, полученные на этом этапе, задают функциональность и структуру конечной интегральной схемы.

Одним из часто используемых языков описания цифровых схем является SystemVerilog [4]. К его преимуществам можно отнести:

1. Наличие конструкций для описания верификационных окружений (verification environment, testbench).
2. Наличие средств описания аппаратуры на различных уровнях абстракции (уровень регистровых передач и уровень логических вентилей).
3. Большое разнообразие синтаксических конструкций, ускоряющих разработку модели, например, специализированные блоки процедурной логики, такие как `always_comb`, `always_latch` и `always_ff`, интерфейсы, различные виды операторов `case` и другие.

В больших проектах наличие большого числа абстракций, предоставляемых этим языком, может привести к семантически некорректным описаниям, допустимым с точки зрения синтаксиса. При синтезе логической схемы конструкции языка преобразуются в логические вентили, а при симуляции RTL-модель вместе с тестовым окружением преобразуется в представление, пригодное для исполнения и верификации на целевом оборудовании (чаще всего в программный код на C/C++). Наиболее часто встречаются ошибки, связанные с расхождением результатов симуляции исходной RTL-модели и синтезированной логической схемы: например, различие в семантике операций 4-значной логики, частично определенные значения сигналов, различные виды присваиваний в процедурных блоках и т.д.

Для предотвращения подобных ошибок составляются сборники правил, описывающие неправильные варианты использования конструкций языка и последствия неправильного использования. Для проверки соответствия описаний списку правил может использоваться ручная или автоматическая проверка кода. Ручная проверка, выполненная экспертом, зачастую имеет более высокую точность, однако автоматическая проверка с использованием инструментов статического анализа позволяет более эффективно и быстро проверять большие проекты на соответствие списку правил. При разработке проекта должна использоваться комбинация этих техник для повышения качества проверки.

На сегодняшний день коммерческие решения для статического анализа описаний на языке SystemVerilog представлены малым количеством компаний (например, Synopsys и Cadence Design Systems), а открытые решения, хотя и находятся на стадии активного развития, на данный момент широко не представлены. В связи с этим, разработка решения статического анализа описания аппаратуры на SystemVerilog является особенно актуальной.

Данная работа посвящена разработке статических проверок описаний схем на языке SystemVerilog на соответствие разработанному списку правил, направленных на выявление и предотвращение неточностей и ошибок на ранних этапах разработки сверхбольших интегральных схем (СБИС).

2. Языки описания аппаратуры

Цифровая схема может быть описана с помощью логических вентилей и соединений между ними (gate-level netlist). Учитывая высокую сложность современных интегральных схем (10^6-10^9 вентилей), такое описание практически невозможно выполнить вручную. В связи с этим в проектировании цифровых схем получил распространение подход к описанию модели аппаратуры на уровне регистровых передач (RTL, Register-Transfer Level). На этом уровне цифровая схема описывается в терминах сигналов, регистров и логических операций между ними, не вдаваясь в подробности реализации при помощи логических вентилей. Такое описание представляет передачу значений сигналов в рамках одного периода тактового сигнала.

Языки описания аппаратуры, такие как Verilog, SystemVerilog и VHDL [5], позволяют описывать структурные и поведенческие модели на разных уровнях абстракции: на уровне регистровых передач и на уровне логических вентилей. Основным сценарием использования является проектирование RTL-модели, имитационное тестирование (simulation-based verification) RTL-модели в симуляторе и синтез по RTL-модели логической схемы. С помощью этого описания высокого уровня можно получить описание низкого уровня (gate-level netlist), которое в конечном итоге будет преобразовано в фактическую схему.

Начальными этапами синтеза HDL-описаний является построение предварительной (pre-elaborated) и развернутой (elaborated) моделей. Построение первой происходит после синтаксического разбора. Построенная предварительная модель содержит лишь описание иерархии модулей (Verilog/SystemVerilog) и сущностей (VHDL) без создания экземпляров с конкретными значениями параметров, что позволяет провести классификацию – какие модули являются верхнеуровневыми, а какие нет. Затем строится развернутая модель, в которой вычисляются все параметры всех экземпляров модулей (в том числе зависимые от значений других параметров), создаются непосредственно экземпляры модулей, а также разрешаются иерархические имена и блоки генерируемой логики (Verilog и SystemVerilog).

Языки описания аппаратуры, использующие RTL-представление, имеют синтаксис, схожий с такими процедурными языки программирования, как Pascal и C, но могут также допускать и более высокоуровневые стили описания, например, объектно-ориентированные описания в SystemVerilog. Главное отличие от языков общего назначения состоит в модели вычислений, например, существуют операции в RTL-модели схемы аппаратуры, которые должны выполняться все вместе одновременно в конце такта сигнала тактовой частоты в реальной микросхеме или в модели при симуляции несмотря на то, что они могут быть описаны в RTL-модели в императивном стиле вперемешку с последовательными операциями. Соответственно, это стоит учитывать при разработке HDL-описаний, в то время как в языках программирования все операции происходят последовательно.

В описании аппаратуры применяется аналогичный языкам программирования набор средств для объявления и использования процедур, функций и типов данных. Эти типы данных включают булевы значения, целые числа, битовые векторы и массивы фиксированной длины. Арифметические и логические операции, включая побитовые, также схожи. Основной структурной единицей [6], используемой для описания аппаратуры, является модуль (`module`), который соответствует элементам аппаратных систем. Каждый модуль имеет интерфейс, включающий входные (`input`) и выходные (`output`) сигналы. Обычно интерфейс модуля содержит специальные входные сигналы, такие как сигналы тактовой частоты (`clk`) и сброса (`reset`). Функциональная логика модуля описывается в его теле. Тело модуля может содержать фиксированное количество экземпляров других модулей (в случае иерархического описания) и фиксированное количество процессов (`process`, `always`). Процессы выполняются параллельно в рамках одного модуля. Каждый процесс может иметь список чувствительности (`sensitivity list`) и тело, содержащее последовательность операторов. Семантика некоторых операторов совпадает с семантикой аналогичных

операторов в императивных языках программирования: например, последовательные (блокирующие) присваивания, условные операторы и операторы цикла. Также используются параллельные (не блокирующие) присваивания. События в описаниях включают изменения уровня входного сигнала: приход переднего фронта сигнала (`rising edge, posedge`), заднего фронта сигнала (`falling edge, negedge`) и произвольного фронта сигнала (`event`). Важной особенностью описаний аппаратуры является специфический режим выполнения процессов: во время исполнения каждый процесс постоянно ожидает прихода события, и, если событие перечислено в списке чувствительности процесса, то выполняется соответствующее тело, и процесс вновь переходит в состояние ожидания. Присваивание значений переменным в блоках параллельных присваиваний осуществляется одновременно, а их обновленные значения становятся доступными только при следующем выполнении тела процесса.

Синтаксис описания аппаратуры на примере языка SystemVerilog приведен в листинге 1.

```
1  module SAMPLE (                          интерфейс модуля
2    input logic RST,                       входной сигнал сброса
3    input logic CLK,                       входной тактовый сигнал
4    input logic D,                         входной однобитный сигнал
5    output logic OUT                       выходной однобитный сигнал
6  );
7
8    always @(posedge CLK or negedge RST)   процесс и список чувствительности
9    begin
10     if (!RST)                            условный оператор
11       OUT <= 1'b0;                       параллельное присваивание
12     else if (D)                          условный оператор
13       OUT <= ~OUT;                       параллельное присваивание
14     else                                 условный оператор
15       OUT <= OUT;                        параллельное присваивание
16   end
17 endmodule
```

Листинг 1. Пример простого описания схемы на SystemVerilog.
Listing 1. An example of a simple circuit design in SystemVerilog.

SystemVerilog [4] – язык описания и верификации аппаратуры, являющийся расширением языка Verilog [7]. В 2005 SystemVerilog был принят как стандарт IEEE 1800-2005. В 2009 стандарт 1800-2005 был объединен со стандартом языка Verilog (IEEE 1364-2005), и была принята актуальная на тот момент версия SystemVerilog – стандарт IEEE 1800-2009. На данный момент актуальными считаются стандарты SystemVerilog IEEE 1800 редакции 2017 и 2023.

На конструкции языка SystemVerilog накладываются различные ограничения в зависимости от того, являются они частью схемы, подлежащей синтезу или нет. Синтезируемая часть описания это та, которую можно представить с помощью логических вентилей. К конструкциям синтезируемых частей описания относятся объявления портов, непрерывные присваивания, триггеры и другие. Напротив, такие конструкции как `program`, `initial` и `final` относятся к несинтезируемой части описания. Для некоторых конструкций возможен синтез, но с рядом ограничений, например, `task`.

Описание схемы должно удовлетворять определенным свойствам, которые инструмент синтеза проверяет перед его выполнением. Однако инструмент не всегда сообщает о нарушениях, обнаруженных в описании, и может исключить несинтезируемую часть из схемы, либо же некорректно осуществить синтез. Например, вызов функции может быть синтезируемым или несинтезируемым.

Конструкции верификации языка SystemVerilog позволяют проверять свойства схемы без ее непосредственной реализации в аппаратуре. С помощью симулятора можно проверить выполнение ограничений, заданных в форме темпоральных утверждений (SVA, SystemVerilog Assertions). Сценарии симуляции работы схемы описываются разработчиком с помощью этих конструкций.

Для большинства конструкций языка SystemVerilog можно статически определить, относятся они к синтезируемой части схемы или нет. Эти конструкции отличаются назначением, а также особенностями интерпретации (поведение схемы при симуляции отличается от поведения реальной схемы после синтеза).

3. Статический анализ описаний на языке SystemVerilog

Статический анализ – анализ кода, проводимый без компиляции программ (без логического синтеза аппаратуры) и их исполнения. Для статического анализа описаний аппаратуры могут быть применены техники, используемые при анализе программного обеспечения.

Рассмотрим техники, которые активно применяются в анализе описаний аппаратуры.

- **Сопоставление синтаксических конструкций**. Для использования этой техники необходим парсер языка, способный строить абстрактное синтаксическое дерево (АСД), которое представляет собой конечное помеченное ориентированное дерево, где вершины однозначно сопоставляются операторам языка описания. АСД используется для поиска паттернов кода. Поиск осуществляется с помощью специальных обходчиков дерева, собирающих данные о его вершинах. Эта техника анализа является одной из самых простых в реализации, а также самых быстрых, так как сложность выполнения обхода пропорциональна размеру дерева. Минусом такой техники является высокий процент ложных срабатываний (false positives) при проверке свойств схемы, требующих знаний о возможных значениях переменных и путях исполнения программы.
- **Анализ потока данных**. Для использования этой техники должен быть построен граф потока управления. Этот граф позволяет отслеживать зависимости по данным и управлению в схеме. Вершины этого графа представляют собой блоки последовательного выполнения операторов в описании. Операторы внутри этих блоков могут порождать факты, которые исследуются в ходе анализа, или говорить о недостоверности факта при выполнении конкретного оператора. В ходе анализа потока данных информация о фактах относительно операторов описания распространяется по всем возможным путям исполнения тестового описания или распространяется на возможные состояния схемы. Распространение происходит итерационно до момента полного совпадения состояний на соседних итерациях. Этот метод анализа позволяет решать такие задачи как: нахождение мертвого кода, анализ живых переменных. Зачастую такой анализ используется для оптимизации вычислений в процессах или для удаления частей схемы, которые не влияют на ее выходы.
- **Символьное исполнение (симуляция)**. Техника позволяет симулировать схему без заданных значений входных сигналов. Инструмент символьного исполнения анализирует возможные пути выполнения для данной схемы и для каждого такта тактового сигнала генерирует набор входных данных для обхода всех путей. Обычно такой запуск может работать лишь ограниченное число тактов, так как символьное исполнение порождает экспоненциальное число возможных путей. При обнаружении пути, на котором есть нарушение определенного свойства схемы, путь достижения запоминается и выдается инструментом символьной симуляции. Минусом данного метода является большая затрата ресурсов на выполнение симуляции множества путей. На данный момент есть несколько работ [8-9], в

которых используются техника символьного исполнения программ, полученных трансляцией SystemVerilog-описания в описание на языке программирования общего назначения. С помощью этого метода можно эффективно решать такие задачи, как обнаружение выхода за границы массива, выявление множественного присваивания одной переменной и другие. Однако на данный момент нет открытых инструментов, реализующих символьное исполнение с полной поддержкой описаний на языке SystemVerilog.

Описанные методы статического анализа покрывают значительную часть ошибок, которые обрабатываются статическим анализатором.

3.1 Обзор инструментов статического анализа описаний на языке SystemVerilog

Для определения методов реализации детекторов был проведен обзор доступных статических анализаторов описаний на SystemVerilog. В настоящее время имеется множество инструментов различного назначения, обладающих функциональностью для проведения статического анализа описаний на языке SystemVerilog, однако в большинстве своем эти инструменты, предоставляются по коммерческим лицензиям, но есть небольшая часть с открытым исходным кодом.

В данной главе представлен обзор существующих открытых и коммерческих решений, включая линтеры, парсеры, симуляторы, инструменты предварительного синтеза.

Кроме того, в данной главе произведен анализ соответствия открытых инструментов ряду критериев. Инструменты оценены с точки зрения поддержки стандарта языка SystemVerilog, наличия необходимых для анализа представлений, лицензирования, функциональных возможностей, таких как интерфейс для реализации статических детекторов, а также интеграции с другими инструментами. Поддержка языка оценивается на основании результатов запуска парсеров на синтетическом наборе тестов sv-tests [10], а также с учетом документации к инструментам. Интеграция рассматривается как возможность совместного использования инструментов для выполнения статического анализа, что в свою очередь может повысить эффективность процедур оценки и улучшить качество анализируемого кода.

3.1.1 Коммерческие инструменты для анализа описаний на языке SystemVerilog

Существует ряд коммерческих инструментов, позволяющих выявить ошибки описаний дизайна цифровой аппаратуры на языках Verilog, SystemVerilog. Наиболее популярными являются SpyGlass, VCS, JasperGold и Xcelium, краткий обзор которых представлен ниже. Стоит отметить, что синтаксические и семантические проверки свойств описаний дизайнов аппаратуры в разной степени присутствуют во всех коммерческих инструментах, в том числе и в тех, которые не представлены в обзоре (Vivado от компании Xilinx, Questa от компании Siemens и другие).

Spyglass [11] – это инструмент статической верификации и анализа RTL-кода, разработанный компанией Synopsys. Инструмент содержит модуль Lint, выполняющий статический анализ кода, проверяя его на соответствие определенным правилам проектирования и выявляя потенциальные ошибки при проектировании. Также в инструменте представлены модули CDC, RDC и Power, предназначенные для выявления ошибок пересечения доменов тактовой частоты, некорректного использования асинхронных сбросов и анализа доменов питания соответственно. Полный список проверок, реализуемых SpyGlass, не опубликован в открытом доступе, но по заявлениям разработчиков покрывает основные ошибки, возникающие в ходе проектирования и разработки описаний схем. Полнота поддержки стандартов не указана явно, но инструмент предназначен для работы с описаниями на SystemVerilog, что подразумевает соответствие стандартам IEEE 1800.

VCS (Verilog Compiled Simulator) [12] представляет собой высокопроизводительный симулятор и инструмент для анализа описаний на языках Verilog и SystemVerilog, разработанный компанией Synopsys. VCS практически полностью поддерживает актуальные стандарты Verilog и SystemVerilog (заявлена поддержка IEEE 1800-2023), осуществляет проверку набора правил линтинга и предлагает широкий спектр средств отладки симулируемой модели, включая визуализацию, трассировку и анализ значений сигналов. Также инструмент поддерживает верификацию по методологии UVM (Universal Verification Methodology) [13].

JasperGold [14] – это инструмент формальной верификации, разработанный компанией Jasper Design Automation (сейчас принадлежит Cadence Design Systems). Он обеспечивает формальную верификацию описаний на языках Verilog и SystemVerilog, проводит доказательства свойств и проверки характеристик цифровых схем на соответствие заявленным спецификациям без использования симуляции. JasperGold также позволяет разрабатывать и добавлять модульные проверки, которые можно комбинировать для создания комплексных сценариев верификации.

Xcelium [15], разработанный компанией Cadence, является аналогом VCS. Инструмент практически полностью поддерживает актуальные стандарты Verilog и SystemVerilog, осуществляет статические проверки правил линтинга, предоставляет возможности для многопоточной симуляции, поддерживает верификацию по методологии UVM и обладает графическим интерфейсом (в отличие от VCS).

3.1.2 Инструменты с открытым исходным кодом для анализа описаний на языке SystemVerilog

В области разработки и анализа описаний аппаратуры на языке SystemVerilog существует ряд инструментов с открытым исходным кодом, каждый из которых предоставляет специфические возможности для синтаксического анализа, линтинга и симуляции кода. В данной работе рассматриваются и сравниваются шесть таких инструментов: Verible, svlint, Verilator, Surelog, CIRCT и slang. Популярные инструменты Yosys [16] и Icarus Verilog [17] в обзоре не фигурируют, так как изначально создавались для логического синтеза и симуляции описаний цифровой аппаратуры на Verilog и не являются инструментами статического анализа.

Verible [18] представляет собой набор инструментов для работы с описаниями на языке SystemVerilog, включая средства статического анализа и линтинга. Особенностью Verible является построение конкретного синтаксического дерева (дерева разбора) для HDL-описания до запуска препроцессора. На текущем этапе разработки Verible частично поддерживает стандарт SystemVerilog, и некоторые конструкции языка могут быть не реализованы. Инструмент предоставляет собственную инфраструктуру для реализации новых детекторов и матчеров – функций для поиска определенных паттернов в дереве разбора. На данный момент реализованы линтер и языковой сервер. К преимуществам Verible также можно отнести свободную лицензию Apache 2.0, которая способствует его широкому использованию и развитию.

svlint [19] – это инструмент линтинга, предназначенный для анализа описаний на языке SystemVerilog. Он использует sv-parser [20] в качестве парсера, который практически полностью покрывает стандарт SystemVerilog. svlint выполняет анализ на основе дерева разбора, построенного sv-parser; при этом, в отличие от Verible, инструмент выполняет препроцессирование описания. svlint предоставляет инфраструктуру для реализации собственных детекторов и матчеров синтаксических конструкций. Кроме того, инструмент включает собственный языковой сервер svls, реализующий протокол LSP. Распространяется под лицензией MIT, что обеспечивает его свободное использование и модификацию.

Verilator [21] – это инструмент симуляции описаний аппаратуры на языках Verilog и SystemVerilog, разрабатываемый CHIPS Alliance. Он частично поддерживает конструкции стандарта IEEE 1800-2017 языка SystemVerilog и обладает несколькими внутренними представлениями описаний аппаратуры: АСД (включая XML-представление АСД), представление в виде кода на языках C++ или SystemC для эмуляции аппаратуры, а также представление DfgGraph на основе графа потока данных для оптимизатора комбинационной логики. В частности, в работе [9] эмуляция аппаратуры в виде C++ кода использовалась для символьной симуляции схем при интеграции с инструментом символьного выполнения KLEE [22]. Verilator предоставляет инфраструктуру для реализации собственных детекторов, включая обходчики АСД. Встроенный в инструмент модуль линтига анализирует и находит семантические ошибки, которые могут повлиять на результаты симуляции или синтеза. Инструмент лицензирован под GNU Lesser General Public License v3.0 (LGPL-3.0), что ограничивает его коммерческое использование.

Surelog [23] – это компилятор описаний аппаратуры на языке SystemVerilog, который на выходе строит модель аппаратуры в специальном формате (Universal Hardware Data Model, UHDM [24]). UHDM является открытым форматом, разработанным CHIPS Alliance для представления RTL-модели в универсальном формате, подходящем для интеграции парсера Surelog с различными инструментами синтеза и симуляции, например, Yosys и Verilator (для которых существуют трансляторы из UHDM в их внутренние представления). С помощью средств UHDM можно описывать как предварительную, так и развернутую RTL-модель. Парсер Surelog практически полностью поддерживает стандарт IEEE 1800-2017. Инструмент строит АСД, имеет встроенный препроцессор и генерирует развернутую модель UHDM, что позволяет выполнять межмодульный анализ. Surelog предоставляет Python API для создания правил линтинга и обхода АСД, однако не имеет C++ API для реализации статических детекторов. Лицензируется под Apache 2.0.

CIRCT [25] – представляет собой набор инструментов для симуляции, линтинга и компиляции описаний на языках описания аппаратуры, таких как Chisel, SystemC и Verilog. Основной целью CIRCT является представление моделей, описанных на различных языках в виде промежуточного представления MLIR [26], соответствующего форме единственного присваивания (SSA). MLIR (Multi-Level Intermediate Representation) – часть проекта LLVM, предоставляющая инфраструктуру для создания и работы с промежуточными представлениями. Она включает операции, регионы и блоки, образующие рекурсивную структуру. Диалекты объединяют операции, типы и атрибуты для конкретных задач. MLIR поддерживает более 40 встроенных диалектов и позволяет создавать пользовательские с помощью специализированного языка.

CIRCT предоставляет диалекты, относящиеся к различным языкам описания аппаратуры, которые могут быть преобразованы в основные (core) диалекты (HW, Comb, Seq, Interop) или же могут быть подвергнуты анализу в рамках конкретного диалекта. CIRCT содержит как промежуточные представления, так и различные вспомогательные абстракции, такие как граф инстанциации модулей, что позволяет проводить межмодульный анализ. Поскольку CIRCT основывается на инфраструктуре MLIR, в настоящее время проводятся работы по эмуляции аппаратуры с помощью трансляции описаний в LLVM. На данный момент активно осуществляется интеграция инструмента slang в качестве парсера для SystemVerilog в рамках CIRCT. Следует отметить, что данный инструмент считается экспериментальным и пока не используется широко с другими инструментами, однако у него есть активное сообщество разработчиков. Лицензия инструмента Apache 2.0.

slang [27] – это инструмент для статического анализа и компиляции описаний аппаратуры на языке SystemVerilog, обладающий практически полным покрытием стандарта IEEE 1800-2017, а также стандарта IEEE 1800-2023. slang строит АСД и предоставляет инфраструктуру для реализации статических детекторов, включая интерфейс для обхода АСД. Помимо

компилятора и статического анализатора инфраструктура slang включает несколько дополнительных инструментов:

- Rewriter: инструмент, позволяющий автоматически вносить изменения в файлы описаний аппаратуры посредством модификации АСД.
- Netlist: инструмент для выполнения структурных проверок, строящий связи и соединения между различными модулями.
- slang-tidy [28]: инструмент статического анализа описаний на языках SystemVerilog и Verilog, использующих сопоставление синтаксических конструкций.

slang активно используется в других проектах в качестве фронтенда, например, в проекте CIRCT и в утилите Yosys+slang [29]. Инструмент распространяется под лицензией MIT, что способствует его интеграции и расширению в рамках других решений.

3.1.3 Сравнение инструментов с открытым исходным кодом

В табл. 1 представлено сравнение инструментов с открытым исходным кодом с точки зрения возможностей статического анализа SystemVerilog.

Табл. 1. Сравнение анализаторов с открытым кодом.
Table 1. Open-source analyzers comparison results.

Инструмент	**Поддержка SystemVerilog**	**Внутренние представления**	**Виды анализа**	**Интерфейс для создания детекторов**
Verible	частичная	Дерево разбора	Сопоставление синтаксических конструкций	+
svlint	практически полная	Дерево разбора	Сопоставление синтаксических конструкций	+
Verilator	частичная	АСД, C++, SystemC, граф потока данных	Сопоставление синтаксических конструкций; анализ потока данных	–
Surelog	практически полная	АСД, UHDM	Сопоставление синтаксических конструкций	+ (Python)
CIRCT	частичная	MLIR	Анализ потока данных на SSA	–
slang	практически полная	Дерево разбора, АСД	Сопоставление синтаксических конструкций	+

Как видно из таблицы, практически полную поддержку стандарта IEEE 1800-2017 предоставляют только такие инструменты, как svlint, Surelog и slang.

Инструменты Verible и svlint не позволяют проводить статический анализ с помощью сопоставления синтаксических конструкций, так как не способны построить АСД по описанию аппаратуры. Важно учитывать возможность расширения функциональности статического анализатора путем добавления других форм анализа или его интеграции с инструментами, поддерживающими такие методы.

Verilator является достаточно удобным и мощным инструментом для статического анализа с точки зрения инфраструктуры. Анализ производится в нем поэтапно в процессе трансляции

и построения модели на C++. Но данный инструмент создавался в первую очередь для языка Verilog, соответственно, поддержка стандарта языка SystemVerilog на данный момент достаточно ограничена. Также, немаловажным аспектом, повлиявшим на исключение этого инструмента из списка рассматриваемых, стала ограничивающая лицензия.

При выборе инструмента для реализации статического анализа описаний на SystemVerilog важным фактором стало наличие интерфейса для создания статических детекторов и интерфейса для выдачи диагностических предупреждений. В данном контексте наиболее подходящим решением оказался инструмент slang, так как он предоставляет широкофункциональные интерфейсы для разработки статических детекторов (через отдельный инструмент статического анализа – slang-tidy), а также интерфейсы для создания и выдачи диагностических предупреждений.

Также slang активно развивается сообществом, практически полностью поддерживает стандарт IEEE 1800-2017, демонстрирует широкую функциональность для выполнения углубленного анализа описаний и интегрирован в другие средства в качестве фронтенда, что обеспечивает возможность взаимодействия в экосистеме открытых инструментов для анализа описания аппаратуры.

С учетом перечисленных преимуществ, именно slang был выбран в качестве основной инфраструктуры для реализации системы статического анализа.

4. Правила проверки описаний на языке SystemVerilog

Несмотря на мощность и гибкость языка описания и верификации аппаратуры SystemVerilog, отсутствие четких руководящих принципов и стандартов проектирования может привести к разнообразным проблемам при разработке, таким как снижение читаемости кода, увеличение количества ошибок и усложнение процесса симуляции (моделирования), верификации и синтеза.

В этой главе рассматривается процесс и мотивация создания списка правил, разработанного на основе опыта инженеров российских дизайн-центров электроники и анализа зарубежных коммерческих анализаторов описаний на языке SystemVerilog. Составленный список направлен на унификацию подходов к написанию кода и повышение общего качества проектов и содержит как правила из стандартов (например, STARC [30]), так и пользовательские правила, отражающие опыт использования коммерческих инструментов.

Разработка правил, направленных на выявление и предотвращение неточностей и ошибок на ранних этапах разработки, значительно сокращает риск возникновения проблем в последующих этапах, что ведет к более надежным и устойчивым проектам с точки зрения их разработки и сопровождения. Стандартизация подходов к написанию кода способствует повышению совместимости кода между различными инструментами и средами разработки. Это особенно важно в условиях быстро меняющейся технологической базы и разнообразия используемых САПР.

В процессе сбора и анализа опыта инженеров, работающих с Verilog и SystemVerilog, были выявлены наиболее частые проблемы и ошибки, возникающие в процессе проектирования, реализации, верификации и синтеза проектов. Для каждой проблемы было сформулировано конкретное правило, направленное на ее предотвращение и соответствующее улучшение качества кода. Также были проанализированы правила из таких анализаторов, как SpyGlass и VCS, реализующие известные стандарты кодирования (например, OpenMore [31], DO-254 [32] и STARC), с целью уточнения сформулированных ранее правил, используя опыт коммерческих решений анализа описаний схем. Итоговый список правил был классифицирован по **18** категориям, соответствующим различным аспектам языка и процесса разработки. Такая структура позволяет легко ориентироваться в правилах и применять их к соответствующим областям кода. Полная классификация правил по категориям представлена в табл. 2.

Табл. 2. Классификация выделенных правил.
Table 2. Rules classification.

Категория	**Описание**
arith	проверки арифметических операций
assign	проверки операций присваивания
case	проверки операторов case
design	проверки логики и семантики описания схем
fsm	проверки конечных автоматов в дизайне
latch	проверки использования защелок
loop	проверки операторов цикла
port	проверки использования портов
pp	проверки использования директив препроцессора
range	проверки выходов за границы
reset	проверки конфликтов, связанных с сигналами сброса
signal	проверки корректности использования сигналов
synth	проверки синтезируемости описания схем
style	проверки оформления описания схем
timing	проверки конфликтов, связанных с временными задержками
type	проверки конфликтов, связанных с типами
variable	проверки использования переменных
width	проверки конфликтов, связанных с размером битовых векторов

Правила попадающие в определенную категорию, обладают общей характеристикой: например, правила категории `synth` направлены на проверку корректности синтезируемости описания аппаратуры с точки зрения определенных конструкций или семантических свойств (например, к данной категории относится правило, когда выходной регистр модуля обновляется на разных фазах сигнала тактовой частоты), правила из категории `signal` проверяют корректность описаний на SystemVerilog при работе с сигналами (например, к данной категории относится правило, когда используется сигнал, которому никогда не устанавливается значение).

При разработке правил учитывались:

- Соответствие стандартам: правила должны соответствовать официальным спецификациям языка SystemVerilog и общепринятым практикам.
- Практическая применимость: правила должны быть применимы к реальному коду и не создавать чрезмерную нагрузку на разработчиков.
- Ясность и однозначность: формулировки правил должны быть понятными и не допускать неоднозначных трактовок.

В результате был сформулирован список из **113** правил для языков описания аппаратуры Verilog и SystemVerilog. Для реализации статических детекторов для этих правил была разработана система для статического анализа SVAN, описание которой дано в следующем разделе.

Отметим, что отдельно были выделены правила, связанные с корректностью синхронизации (CDC и RDC) и корректностью использования доменов питания, однако рассмотрение этих правил выходит за рамки работы.

5. Система статического анализа SystemVerilog

На рис. 1 представлена структурная схема разработанной системы статического анализа описаний аппаратуры на языке SystemVerilog – SVAN.

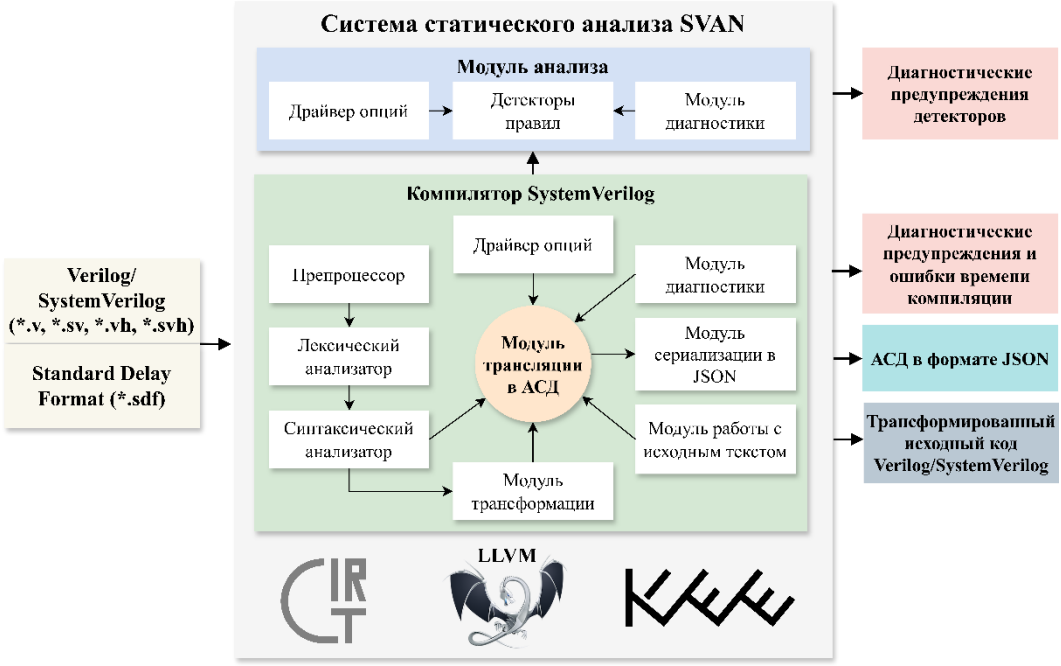

Рис. 1. Структурная схема системы статического анализа SVAN.
Fig. 1. Structural diagram of the SVAN static analysis system.

На верхнем уровне система статического анализа состоит из следующих компонентов:

- компилятор SystemVerilog;
- модуль анализа;
- подсистема анализа на основе инфраструктур CIRCT, LLVM и KLEE.

На вход системе поступают файлы с исходным текстом описания аппаратуры на языках Verilog или SystemVerilog, файлы со спецификацией задержек в формате SDF (Standart Delay Format), а также опции для компилятора и модуля анализа.

Модуль работы с исходным текстом анализирует пути к директориям с файлами и библиотеками с исходным кодом описаний аппаратуры на языках Verilog и SystemVerilog и составляет один или несколько единиц трансляции, в зависимости от переданных опций.

Затем файлы с описанием аппаратуры в виде одной или нескольких единиц трансляции обрабатываются препроцессором, который отвечает за раскрытие макроопределений в зависимости от переданных опций, а также за разрешение зависимостей включаемых файлов и подключаемых библиотек. Предобработанные единицы трансляции разбиваются на лексемы лексическим анализатором для их дальнейшей синтаксически управляемой трансляцией методом рекурсивного спуска, которая реализована в модуле синтаксического анализатора. В процессе синтаксически управляемой трансляции для каждой единицы

трансляции строится синтаксическое дерево разбора, по которому, затем, строится абстрактное синтаксическое дерево модулем трансляции в АСД.

Синтаксическое дерево разбора единицы трансляции можно трансформировать с помощью программного интерфейса модуля трансформации, который позволяет описывать обходчики синтаксического дерева и преобразовывать его узлы. Модуль также предоставляет возможность трансляции преобразованного синтаксического дерева разбора в Verilog и SystemVerilog, а также генерации АСД для дерева разбора с помощью модуля трансляции в АСД.

Модуль трансляции АСД рекурсивным обходом в глубину преобразует узлы синтаксического дерева разбора в узлы АСД, в результате строится предварительная модель каждой единицы трансляции. Затем по предварительной модели с помощью дополнительного семантического анализа зависимостей данных строится развернутая модель для каждой единицы трансляции. Ошибки и предупреждения, собранные в процессе лексического, синтаксического и семантического анализа обрабатываются и выдаются в виде предупреждений в файл или стандартный поток вывода модулем диагностики.

Построенное АСД подается на вход модулю анализа, который запускает наборы детекторов правил по категориям, специфицированным в конфигурационном файле. Детекторы в процессе обхода и анализа АСД выдают диагностические предупреждения с привязкой к позиции в файлах с исходным кодом. Предупреждения обрабатываются и выводятся в файл или стандартный поток вывода модулем диагностики.

АСД можно сохранить для обработки сторонними инструментами с помощью модуля сериализации в JSON, который получает на вход построенный АСД от модуля трансляции в АСД, затем рекурсивным обходом в глубину преобразует его узлы вместе с атрибутами в формат JSON, а также выводит его в файл или в стандартный поток вывода в зависимости от переданной опции.

5.1 Компилятор SystemVerilog

Компилятор основан на программной библиотеке с открытым исходным кодом slang [27], которая предоставляет различные компоненты для лексического анализа, синтаксического анализа, проверки типов и элаборации описаний на SystemVerilog. Библиотека slang предоставляет инструментарий для трансляции и анализа проектов на SystemVerilog, а также API для использования в качестве внешнего интерфейса для инструментов синтеза, симуляторов, линтеров, редакторов кода и инструментов рефакторинга.

Компилятор решает задачи, связанные с трансляцией описаний на языке SystemVerilog: непосредственно трансляцию текстового описания (препроцессирование, лексический и синтаксический анализ); построение дерева разбора; построение абстрактного синтаксического дерева; построение предварительной модели; построение развернутой модели.

Синтаксический анализ реализован методом рекурсивного спуска, который обеспечивает устойчивость к ошибкам в исходном тексте. Этот подход позволяет продолжить анализ даже при возникновении синтаксических ошибок. Результатом работы синтаксического анализатора является построенное дерево разбора, состоящее из синтаксических узлов разных типов. АСД находится на более высоком уровне абстракции и строится на отдельном этапе.

Построение дерева разбора, абстрактного синтаксического дерева, предварительной и развернутой модели реализовано с помощью модуля трансляции в АСД.

В процессе компиляции строится развернутая модель описания путем элаборации предварительной модели. Эта модель представлена в виде АСД. В процессе элаборации происходит, в том числе создание экземпляров модулей, соединение портов, подстановка параметров. В каждом проекте на языке SystemVerilog есть модуль верхнего уровня,

описывающий схему целиком, в то время как экземпляры остальных модулей создаются внутри модуля верхнего уровня. Такое представление задает иерархию соединения модулей итоговой схемы. Создание экземпляров модулей позволяет вычислять значения параметров для каждого экземпляра, и, как следствие, провести межмодульный анализ.

В рамках данной работы парсер slang был доработан в целях обеспечения полной поддержки соответствия стандарту SystemVerilog IEEE 1800-2017, в частности:

- Реализована поддержка статических проверок определений и разрешений множественных сигналов тактовой частоты (разделы стандарта 16.13 и 16.16);
- Реализована поддержка статической проверки невырожденности последовательностей (раздел стандарта 16.12.22) и доработка механизма проверок последовательностей, возвращающих только пустые сопоставления;
- Добавлен парсер файлов со спецификацией задержек в формате SDF и реализован механизм сопоставления и аннотации соответствующих конструкций, описывающих задержки и временные ограничения в SystemVerilog (раздел стандарта 32);
- Реализована механизм статической проверки покрытия строк в таблице состояний всех возможных состояний чувствительного к фазе последовательностного UDP (раздел стандарта 29.6), а также доработан механизм проверок перекрытия строк таблицы состояний UDP;
- Реализован механизм статической проверки перекрытия псевдонимов (алиасов) сигналов (раздел стандарта 10.11);
- Доработан механизм статических проверок виртуальных интерфейсов (раздел стандарта 25.9);
- Доработан механизм проверки ограничений на использование методов последовательностей `triggered` и `matched` в SystemVerilog;
- Другие доработки и исправления, связанные с лексическим и синтаксическим анализатором, PLA, рекурсивными свойствами, арифметикой целых чисел, отложенными операторами `assert` и т.д.

5.2 Модуль анализа

Модуль анализа основан на инструменте статического анализа описаний на языках SystemVerilog и Verilog slang-tidy из библиотеки slang. Инструмент предназначен для статического анализа описаний на языках SystemVerilog и Verilog, в том числе для реализации статических детекторов, использующих сопоставление синтаксических конструкций с помощью механизма обходчиков АСД.

Все детекторы этого инструмента реализуют общий интерфейс – `TidyCheck`, используемый для регистрации и вывода диагностики. Интерфейс предоставляет метод `check`, который принимает в качестве входного параметра корень АСД и является входной точкой для запуска одного или нескольких обходчиков, анализирующих АСД. Также интерфейс предоставляет методы для описания информации о детекторе, такие как название детектора, короткое описание, уникальный код диагностики и другие (листинг 2).

Большая часть детекторов требует реализации анализа с помощью обхода синтаксического дерева. Для этой цели в библиотеке slang предусмотрен интерфейс обходчика АСД. Обход производится от корневой вершины, которой обычно является вершина с типом – «единица компиляции» (`CompilationUnit`) к листовым. Пользователь может определить обработчик (`handler`) для каждого типа вершины АСД и собирать необходимую информацию во время обхода. Стоит отметить, что каждый узел АСД компилятора SVAN наследует один из четырех базовых типов: утверждение (`Statement`), выражение (`Expression`), символ

(`Symbol`) или тип (`Type`), подобно тому, как это реализовано в инфраструктуре компилятора Clang [33]. При определении метода обработчика определенного типа узла АСД этот обработчик вызывается обходчиком после выбора наиболее подходящего варианта с точки зрения иерархии типов.

```
class TidyCheck {
public:
    virtual bool check(const slang::ast::RootSymbol& root) = 0; // Точка входа анализа

    virtual std::string name() const = 0;              // Наименование детектора
    virtual std::string description () const = 0;      // Описание
    virtual std::string shortDescription() const = 0; // Краткое описание

    virtual slang::DiagCode diagCode() const = 0; // Целочисленный уникальный код
                                                  // диагностики, выдаваемый детектором
    virtual slang::DiagnosticSeverity diagSeverity() const = 0;
    virtual std::string diagString() const = 0;  // Строковое представление
                                                 // диагностического предупреждения
protected:
    slang::Diagnostics diagnostics;
    slang::TidyKind kind;
}
```

Листинг 2. Интерфейс класса TidyCheck.
Listing 2. TidyCheck interface.

Изначально инструмент slang не поддерживал автоматический обход дочерних узлов, что существенно усложняло логику кода детекторов. Интерфейс детекторов был доработан для поддержки автоматического определения узлов-потомков и их обхода.

В некоторых случаях при реализации детекторов правил может потребоваться классифицировать и отфильтровать отдельные файлы или участки кода на Verilog и SystemVerilog, которые относятся к тестовой среде (testbench), чтобы они не попадали под проверку свойств синтезируемого описания. Для этого реализован механизм указания пользователем того набора файлов и конструкций языка, которые необходимо отбросить при анализе.

Для упрощения разработки детекторов, анализирующих свойства тактовых сигналов и сигналов сброса, разработаны эвристические алгоритмы поиска таких сигналов в модулях, основанные на анализе их использований в процедурных блоках. Кроме того, поддерживается возможность пометки тактовых сигналов и сигналов сброса по именам через опции slang-tidy. Среди других добавленных возможностей стоит упомянуть эвристический механизм анализа и распространения значений сигналов в многозначных логиках, а также механизм определения комбинационной и последовательностной логики в процедурных блоках.

5.3 Метод анализа описаний с использованием инфраструктуры CIRCT и KLEE

Детекторы, для которых требуется анализ на уровне графа потока управления или графа потока данных, могут быть реализованы с использованием инфраструктуры CIRCT, возможности которой в совокупности покрывают виды анализа, сложно реализуемые с достаточной степенью точности на уровне структурного анализа на АСД. Примером такого анализа может служить распространение сигналов, содержащих `X и `Z биты.

Также существует возможность реализации детекторов, требующих статический анализ с использованием механизма символьного выполнения. Такой механизм реализован в инструменте с открытым исходным кодом KLEE [22]. KLEE является инструментом

динамического символьного выполнения для анализа всех возможных путей выполнения программы, построенным на базе LLVM. Инструмент заменяет конкретные входные данные символьными переменными, что позволяет исследовать множество путей выполнения, которые могли бы возникнуть при различных комбинациях входных данных. В ходе символьного выполнения KLEE осуществляет проверку состояний программы на наличие ошибок, включая переполнения, деление на ноль и нарушения свойств безопасности.

На рис. 2 представлена предварительная схема возможных путей трансляции SystemVerilog во внутреннее представление LLVM IR [34], на основе которого осуществляется символьное исполнение в KLEE.

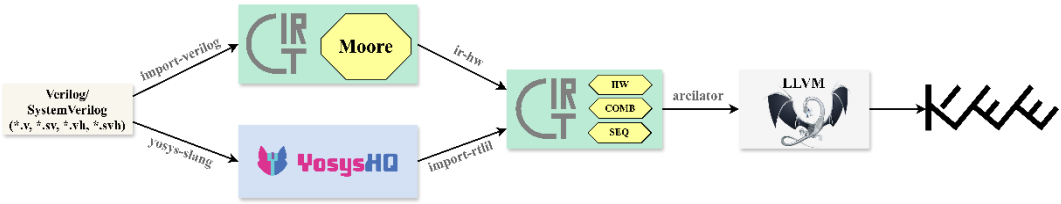

Рис. 2. Схема возможных путей трансляции в LLVM IR.
Fig. 2. LLVM IR translation paths.

Один из путей подразумевает трансляцию АСД представления описания, полученного Slang, в промежуточное представление CIRCT, используя диалект Moore [35]. Полученное представление в диалекте Moore преобразуется в представление на множестве основных низкоуровневых диалектов CIRCT – таких, как Comb, Seq, HW, – или других, подходящих для моделирования аппаратуры и оптимизации. После этого, с использованием симулятора arcilator [36], разработанного в рамках проекта CIRCT, выполняется трансляция в LLVM IR. Данный путь не предполагает использования промежуточных инструментов для получения LLVM IR, что является существенным преимуществом.

Второй путь подразумевает трансляцию АСД-представления в промежуточное представление Yosys RTLIL [37] при помощи расширения для инструмента Yosys (yosys-slang [29]) с дальнейшей конвертацией в CIRCT и LLVM IR. Данный путь предполагает использование дополнительного инструмента, но предоставляет возможность для анализа описаний с помощью средств Yosys, например, обнаружение защелок и выделение и анализ детерминированных конечных автоматов (FSM, Finite State Machine).

После трансляции описания аппаратуры на SystemVerilog в LLVM IR полученное представление передается на вход инструменту KLEE для символьного выполнения.

Выбор окончательного набора инструментов для реализации детекторов проверки правил, требующих анализ потока данных и механизма символьного выполнения, будет произведен в рамках дальнейшего исследования.

6. Разработка и реализация детекторов

В данном разделе рассмотрена реализация двух детекторов. За основу реализации детекторов для разработанных правил была выбрана инфраструктура анализатора slang-tidy, интерфейс и основные принципы работы которого были разобраны в разделе 5.

6.1 Детектор правила PORT10

Рассмотрим реализацию простого детектора для проверки правила PORT10, которое звучит следующим образом:

> «Шины инстанциаций модуля не должны быть присоединены в обратном порядке»

Использование обратных подключений (например: `[15:0]->[0:15]`) является синтаксически корректной, но плохой практикой процесса разработки, так как может

приводить к ошибкам чтения неинициализированных битов.

В листинге 3 приведен пример описания на языке SystemVerilog с конструкциями как нарушающими данное правило, так и соответствующими ему.

```
module test (input clk, input[0:5] in, inout [1:0] io);
endmodule


module top (input clk);
  wire clk;
  wire [0:5] sig1, sig2;
  wire [5:0] sig3, sig4;
  test t1(clk, sig3, sig4[1:0]);   // Нарушение – порядок битов sig3
                                   // не совпадает с in
  test t2(clk, sig1, sig2[0:1]);   // Нарушение – порядок битов sig2
                                   // не совпадает с io
  test t3(clk, sig1, sig3[1:0]);   // Нет нарушения
endmodule
```

*Листинг 3. Иллюстрация р*азличных аспектов правила PORT10.
Listing 3. Various aspects of the PORT10 rule.

Рассмотрим возможную реализацию детектора для данного правила с использованием slang-tidy. Для реализации данного правила достаточно анализа АСД представления для поиска узлов с типом «инстанциация модуля» (`InstanceSymbol`).

В разделе 5.2 был описан интерфейс детекторов slang-tidy. Реализация метода `check` и обходчика `InstanceVisitor` для данного правила представлена в листинге 4.

```
1  bool InstanceVisitor::handle(const InstanceSymbol& instanceSymbol) {
2   for (auto pConn : instanceSymbol.getPortConnections()) {
3     const auto* conn = pConn->getExpression(); // Извлечение подключаемого к порту сигнала
4     const auto* port = &pConn->port;           // Извлечение символа порта
5     const auto& cType = conn->getType();
6     const auto& pType = port->getType();
7     // Проверка, что типы являются упакованными массивами
8     if (!cType.isPackedArray() || !pType.isPackedArray())
9       continue;
10    // Проверка, что размер упакованных массивов больше 1.
11    // В этом случае не нужно выдавать предупреждение
12    if (pType.range.left == pType.range.right || cType.range.left == cType.range.right)
13      continue;
14
15    // Если у подключаемого сигнала левая граница меньше (больше) правой,
16    // а в определении порта левая граница больше (меньше) правой, то выдать диагностику.
17    if ((pType.range.left > pType.range.right) != (cType.range.left > cType.range.right))
18      diag(conn->location, "Шины присоединены в обратном порядке");
19  }
20  return true; // Обход дочерних узлов по умолчанию
21 }
22
23 // Регистрация детектора
24 class RevConnBuses : public TidyCheck {
25 public:
26    bool check(const RootSymbol& root) override {
27      // Вызов основного обходчика
28      InstanceVisitor visitor(diagnostics, root);
29      root.visit(visitor);
30      return diagnostics.empty();
31    }
32 };
```

Листинг 4. Реализация обходчика инициализаций модулей детектора правила PORT10.
Listing 4. Implementation of the PORT10 rule module initialization detector visitor.

Листинг описывает следующие шаги анализа:

1. Обходчик `InstanceVisitor` посещает все узлы АСД, начиная с корня, пока не встретит узел с типом `InstanceSymbol`, для которого описан обработчик в методе `handle` (строки 1–21).
2. Обработчик в цикле перебирает все соединения (строка 2).
3. Обработчик извлекает из соединений подсоединяемый сигнал (строка 3) и определение порта (строка 4), к которому этот сигнал подключается.
4. После извлечения типов соединения и порта (строки 5–6), происходит проверка, что они являются упакованными массивами (строка 8) (другими словами – проверяется, что их можно рассматривать, как единые сигналы, а не массивы сигналов).
5. Фильтруются случаи (строка 12), когда диапазоны длин порта и подключаемого сигнала больше 1; в противном случае прямой и обратный порядок подключения битов совпадает.
6. Происходит итоговая проверка (строки 17–18) того, что порядок подключения битов совпадает; в ином случае выдается предупреждение.

Стоит отметить, что приведенная реализация детектора правила PORT10 представлена в упрощенном виде для репрезентативности, реализация детектора в модуле анализа несущественно отличается.

6.2 Детектор правила ASSIGN04

Рассмотрим реализацию более сложного детектора для проверки правила ASSIGN04, которое звучит следующим образом:

> «Неблокирующее присваивание не должно использоваться в комбинационной логике, если следом идет блокирующее»

В данном разделе также рассмотрена методика эвристического анализа возможных путей выполнения логических блоков. В листинге 5 приведен пример описания на языке SystemVerilog с конструкциями как нарушающими данное правило, так и соответствующими ему.

Язык SystemVerilog предоставляет три основных типа присваиваний: непрерывное (`assign`), блокирующее (=) и неблокирующее (<=), каждое из которых реализует свое поведение при выполнении операций присваивания. Из них блокирующие и неблокирующие присваивания можно использовать в процедурных блоках.

Неблокирующие присваивания выполняются одновременно в рамках процедурного блока и завершаются к концу его выполнения. Они аналогичны работе регистров в аппаратной реализации схемы. Блокирующие присваивания, напротив, работают по принципу последовательного выполнения, схожему с присваиваниями в языке программирования C.

На практике при проектировании схем блокирующие присваивания должны использоваться в комбинационной логике, а неблокирующие присваивания – в последовательностной логике (глава *Gotcha28: Blocking assignments in sequential procedural blocks* из книги [38]).

Смешение двух типов присваиваний в одном и том же блоке `always` может привести к несоответствиям в результатах симуляции исходной RTL-модели и синтезированной логической схемы. Также стоит отметить, что использование блокирующего присваивания после неблокирующего может привести к неожиданному поведению при симуляции; например, в модуле `top` из листинга 5 на строке 5 инженер-разработчик может ожидать, что в «`c`» запишется значение «`0`» после присваивания на строке 4 (начальное значение переменной «`b`»), но на самом деле это не так: в «`c`» запишется значение «`1`», так как

неблокирующие присваивания параллельно выполняются в самом конце после выполнения всех блокирующих.

```
1  module top ();
2    logic a = 1, b = 0, c = 1;
3    always_comb begin
4      a <= b;
5      c = a; // Нарушение - блокирующее присваивание после неблокирующего
6      b <= b + 1;
7    end
8
9
10   always_comb begin
11     a <= b;
12     if (b == c)
13       c = a;  // Нарушение - блокирующее присваивание после неблокирующего
14     else
15       c = 1;  // Нарушение - блокирующее присваивание после неблокирующего
16   end
17
18   always_comb begin
19     a = b;
20     c <= a; // Нарушения нет
21   end
22 endmodule
```

Листинг 5. Иллюстрация различных аспектов правила ASSIGN04.
Listing 5. Various aspects of the ASSIGN04 rule.

Правило рассматривает процедурные блоки, реализующие *комбинационную* логику, которая описывает схемы без памяти и инерции. В отличие от нее, *последовательностная* логика имеет память, синхронизирована с тактовым сигналом и зависит от предыдущего состояния, а логика *защелок* позволяет сохранять старые значения или обновлять память при определенных условиях.

Для точного разграничения типов логики в процедурных блоках были сформулированы критерии, которые учитывают используемые в них операции. Особое внимание уделено процедурным блокам `always`, которые относятся к тестовой среде. Введем несколько ключевых понятий:

Вход логики: набор символов, используемых в списке чувствительности процедурного блока;

Путь: последовательность операторов в порядке их описания при определенных значениях входов логики;

Выход логики: набор символов, значения которых изменяются внутри процедурного блока хотя бы по одному пути.

К тестовой среде относятся те блоки, которые не содержат входов (обладающий пустым списком чувствительности) или выходов, так как такой тип логики обычно является несинтезируемым большинством инструментов. Синтезируемая логика обычно имеет как входы, так и выходы. Последовательностная логика определяется наличием входа, представляющего фронт тактового сигнала или сброса, а ее выходы находятся в регистрах. Комбинационная логика характеризуется тем, что ее выходы обновляются по всем путям, но входы не содержат фронта тактового сигнала. Логика защелок определяется отсутствием фронта для входных сигналов и тем, что ее выходы обновляются лишь на части путей.

Эти критерии позволяют классифицировать и анализировать различные типы логики в контексте синтезируемых и симуляционных конструкций на языке SystemVerilog, примеры процедурных блоков, каждого типа логики представлены в листинге 6.

`always @(in, en)`
`begin`
`  if (en)`
`  out = in;`
`  else`
`  out = ~in;`
`end`
	`always @(posedge clk)`
	`begin`
	`  count <= count + 1;`
	`  r1 <= r1 - 1;`
	`  r2 <= r2 * 1;`
	`  r3 <= r3 + P;`
	`end`
		`always @(in, en)`
		`begin`
		`  if (en)`
		`  out = in;`
		`  if (en)`
		`  out1 = in;`
		`end`
			`always begin`
			`  clk = ~clk;`
			`  #5`
			`end`
			
			`always @(en)`
			`  $display(en);`

Комбинационная	**Последовательностная**	**Защелка**	**Тестовая среда**

Листинг 6. Примеры типов логики процедурных блоков.
Listing 6. Procedural logic examples.

Несмотря на то, что в стандарте языка SystemVerilog предусмотрены специальные синтаксические конструкции, обозначающие тип логики, такие как `always_comb`, `always_ff` и `always_latch`, семантические ограничения на реализацию логики не определены стандартом в полной мере. Это является лишь маркером для инструментов, работающих с описанием RTL-модели. Соответственно, реализация логики защелки в блоке `always_comb` (например, вследствие ошибки проектирования или опечатки) не является ошибочной с точки зрения стандарта. В случае отсутствия инструмента линтинга и статического анализа, который мог бы проверить такие конструкции по схожим правилам с обычными блоками `always`, недочет может быть обнаружен только на этапе логического синтеза и только в том случае, если инструмент синтеза имеет соответствующую диагностику.

В процессе проектирования детектора данного правила была разработана эвристика, определяющая достижимость путей выполнения кода в процедурном блоке. Порядок выполнения операций при вычислении пути соответствует процедурному стилю, что означает выполнение операций одна за одной в той последовательности, в которой они располагаются в исходном описании на SystemVerilog. Для решения задачи обработки путей выполнения внутри процедурного блока необходимо было решить проблему ветвления путей. Для этого был применен эвристический подход, учитывающий операторы `if`, `case` и `for`, поскольку они наиболее часто используются для ветвления в синтезируемой логике. При синтезе все условные переходы преобразуются в набор логических элементов или мультиплексоров, а циклы, обладающие конечным количеством итераций, полностью раскручиваются.

Для реализации этого детектора был создан простой менеджер ограничений, накладываемых на символы, названный `ConstraintManager`. Этот класс связывает каждый символ, используемый в выражении, с набором линейных ограничений. Линейные ограничения представлены как объединение отрезков значений, которые может принимать число. Полное множество значений, которое может принимать данный символ, определяется его типом. Например, символу `'A'` в листинге 7 соответствует линейное ограничение вида `[-32, -3]`, `[10, 31]`.

```
logic[5:0] A;
if (A <= -3 || A > 9) begin
  // -32 <= A <= -3 или 9 < A <= 31
end;
```

Листинг 7. Пример линейного ограничения внутри оператора `if`.
Listing 7. An example of a linear constraint inside an `if` statement.

`ConstraintManager` используется для определения всех возможных путей выполнения, что усложняет его использование для нескольких символов одновременно. Это происходит

потому, что линейные ограничения должны преобразовываться в декартовы произведения множеств значений.

Для определения путей выполнения кода в процедурном блоке используется обходчик областей видимости программы, представленных в АСД, и собирает информацию о видимых в ней символах и накладываемых на них новых ограничениях. Области видимости образуются модулями, функциями, задачами, блочными операторами, операторами `if`, `case` и циклов. Для модулей, функций и задач происходит сопоставление соединений, фактических аргументов – портам и формальным аргументам для передачи информации о текущих вычисленных линейных ограничениях. Операторы `if`, `case` и операторы циклов обрабатываются особым образом.

Оператор `case`: Каждая ветка оператора `case` формирует отдельную область видимости. Для каждой ветки составляется список символов, обновленных в этой ветке, включая ветку `default`. Если символ не обновлен в одной ветке, но обновлен в другой, это может указывать на логику защелки.

Оператор `if`: Накладывает линейные ограничения на символ, используемый в условии, и создает комплементарные условия для ветки `else`. Для каждой области видимости (ветки `if` и `else`) собирается информация о символах, обновленных при всех возможных значениях символа условия.

Операторы циклов: Для синтезируемости цикла он должен быть раскручиваемым. Циклы `generate for` раскручивается на этапе построения развернутой модели, то есть во время построения АСД. Раскручивание обычных циклов for является трудной задачей для инструментов логического синтеза, так как условия циклов могут быть произвольной сложности, например, использовать выражения, не зависящие от параметров цикла, и другое.

Критерии раскрутки цикла `for` включают необходимость наличия константного инициализатора для счетчика, константного шага, сравнения с константой в условии останова и отсутствие управляющих конструкций `continue` и `break`, приводящим к множественным выходам, а также отсутствие изменений счетчика в теле цикла. Если цикл не раскручиваемый, то предполагается, что символы, изменяемые внутри цикла, обновляются не по всем путям. Это связано с невозможностью гарантирования выполнения цикла конечное постоянное число раз. Для определения того, что цикл является раскручиваемым необходимо, чтобы он содержал инвариантную инициализацию и граничное условие цикла, а также имел только индуктивные переменные в выражении шага. Гнезда циклов должны раскручиваться снизу-вверх. Примеры раскручиваемых и нераскручиваемых циклов можно наблюдать в листинге 8.

```
always @(in, en) begin
  // возможно развернуть цикл
  for (int i = 0; i < 10; i ++)
    ...
  // невозможно развернуть цикл
  for (int i = in; i < 10; i ++)
    ...
  // невозможно развернуть цикл
  for (int i = 0; i < in; i ++)
    ...
end
```

Листинг 8. Пример разворачиваемых и не разворачиваемых циклов.

Listing 8. Examples of loops which can and can't be unrolled.

В SystemVerilog поддерживается множество вариантов циклических управляющих конструкций, из которых на данный момент анализируются лишь `for`, `while`, `do-while` и repeat. Циклы `forever` и `foreach` на данный момент в реализованном инструменте считаются нераскручиваемыми и отнесены к несинтезируемой логике.

Таким образом, если удается вывести ограничения на переменные по всем путям каждого выхода процедурного блока, то становится возможным определить реализуемый в блоке тип логики, и остается лишь проверить присутствие спецификации фронтов сигналов в списке чувствительности.

Для процедурных блоков, реализующих комбинационную логику, детектору нужно проверить, что на каждом пути в процедурном блоке нет блокирующего присваивания, если ранее на том же пути было встречено неблокирующее. Стоит отметить, что при рассмотрении циклов в детекторе используется эвристика, согласно которой каждое тело цикла обходится ограниченное число раз, задаваемое опцией детектора. Эта эвристика помогает отследить комбинации присваиваний, выполняемых на разных итерациях цикла. В листинге 9 представлена упрощенная реализация детектора, реализующая описанные подходы.

В этом листинге представлены обработчики двух обходчиков – `ProceduralBlockVisitor` (строки 2–9) и `AssignmentVisitor` (строки 12–51). Первый обходчик предназначен для фильтрации тех процедурных блоков, которые не реализуют комбинационную логику (строки 4–5), и для инициализации второго обходчика (строки 6–7). `AssignmentVisitor` обрабатывает основные типы управляющих конструкций в процедурных блоках комбинационной логики и выдает диагностику при нахождении блокирующего присваивания, расположенного за неблокирующим на одном пути. Путь задается как набор непротиворечивых ограничений на переменные, используемые в управляющих конструкциях. Ограничения накапливаются в структуре данных `constraintsStack`, реализованной в виде стека, которые затем проверяются на консистентность (строки 37–38) для исключения потенциально недостижимых путей. Структура данных `firstNonBlockingAssignMap` (строка 39) содержит позицию первого неблокирующего присваивания для каждого пути. Первое встреченное на пути неблокирующее присваивание помечается (строки 42–43); при обработке блокирующего присваивания проверяется, встречалось ли на текущем пути неблокирующее присваивание (строка 47), и в таком случае выдается диагностическое сообщение (строки 48–51). Обработчик на строках 12-31 предназначен для учета и распространения ограничений на диапазоны значений переменных, используемых в условных выражениях веток оператора `if`, с учетом вложенности операторов. Стоит отметить, что рассмотрение веток `else-if` было упущено в приведенном листинге с целью упрощения, рассмотрена лишь ветка `else`. Ограничения на переменные ветки `else` вычисляются как отрицание ограничений предшествующих веток оператора (строка 21). После вычисления ограничений рекурсивно вызывается обходчик `AssignmentVisitor` для каждого из путей, передавая вычисленные ограничения дальше (строки 17–18 и 23–25). После обхода всех веток оператора `if` найденная на путях информация о неблокирующих присваиваниях сохраняется в структуре данных `firstNonBlockingAssignMap`, что помогает при анализе присваиваний в управляющих операторах, обладающих зависимыми условиями на одном уровне вложенности, как в листинге 10.

Из листинга видно, что первый процедурный блок не содержит нарушений правила, так как путь, проходящий через два оператора `if`, невозможен. Второй, напротив, содержит два оператора `if`, которые, при определенных значениях `b`, могут посещаться одновременно.

В листинге 9 не представлен обходчик оператора `case`, который собирает информацию по путям каждой из его меток, включая метку по умолчанию, а также не представлен обходчик циклов. Тело каждого цикла посещается несколько раз, так как это помогает уточнить те

границы значений переменных, которые изменяются в цикле и используются вне его пределов.

Стоит отметить, что представленная выше реализация детектора является достаточно упрощенной в целях репрезентативности. Различные аспекты детектора рассматриваемого правила реализованы в инструменте существенно сложнее, например, для многобитных сигналов учитываются также присваивания значений отдельным битам сигнала.

```
// Обход по всем процедурным блокам, реализующим комбинационную логику
bool ProceduralBlockVisitor::handle(const ProceduralBlockSymbol& symbol) {
    // Фильтрация процедурных блоков
    if (!procBlocksKinds.Comb.contains(&symbol))
        return false;
    AssignmentVisitor visitor({}, {});
    symbol.getBody().visit(visitor);
    return true;
}


// Обработка условных операторов if
bool AssignmentVisitor::handle(const ConditionalStatement& cond) {
    ConstraintManager manager(context);
    Constraint ifConstraints = manager.getConstraints(cond);
    auto& newConstraintsStack = pushConstraints(constraintsStack, ifConstraints);


    AssignmentVisitor trueBranch(firstNonBlockingAssignMap, newConstraintsStack);
    cond.ifTrue.visit(trueBranch);


    // Обход else ветки
    Constraint elseConstraint = ifConstraints.getComplementConstraints();
    newConstraintsStack = pushConstraints(constraintsStack, elseConstraints);
    AssignmentVisitor falseBranch(firstNonBlockingAssignMap, newConstraintsStack);
    if (cond.ifFalse != nullptr)
        cond.ifFalse->visit(falseBranch);


    // Анализ результатов обхода и добавление новых ограничений пути
    if (!trueBranch.firstNonBlockingAssignMap.empty())
      mergeAssignMaps(firstNonBlockingAssignMap, rueBranch.firstNonBlockingAssignMap);
    return false; // Аналогично для falseBranch
}


// Выдача диагностики на операторах присваивания
bool AssignmentVisitor::handle(const AssignmentExpression& assign) {
    auto constraints = getConstraints(constraintsStack);
    // Проверка, что текущий путь достижим с точки зрения ограничений
    if (!pathConstraintCheckConsistentency(constraints))
        return false;
    auto& firstNonBlockingAssign = firstNonBlockingAssignMap[constraints];


    // Обнаружено первое неблокирующее присваивание в процедурном блоке
    if (!firstNonBlockingAssign && assign.isNonBlocking())
        firstNonBlockingAssignMap[constraints] = &assign;


    // Обнаружено блокирующее присваивание следом за неблокирующим на этом пути
    if (firstNonBlockingAssign && assign.isBlocking())
        diag(assign->location,
            "Неблокирующее присваивание не должно использоваться"
            " в комбинационной логике, если следом идет блокирующее.");
    return false;
}
```

Листинг 9. Реализация обходчиков детектора правила ASSIGN04.
Listing 9. Implementation of the ASSIGN04 rule detector visitors.

```
always @(*) begin
  a = 0;
  if (b > 5) begin
    a <= 1;
  end
  if (b < 5) begin
    a = 1; // Нарушения нет – ограничения на
           // условия путей противоположны
  end
end

always @(*) begin
  a = 0;
  if (b < 10) begin
    a <= 1;
  end
  if (b < 20) begin
    a = 1; // Нарушение – существует путь,
           // включающий оба присваивания
  end
end
```

Листинг 10. Пример нетривиальных путей в процедурных блоках.
Listing 10. An example of non-trivial paths in procedural blocks.

7. Тестирование на открытых проектах

В рамках данной работы были реализованы детекторы для 38 правил из составленного списка, описанного в разделе 4. Еще 28 покрываются уже имеющимися в slang детекторами или диагностиками, некоторые из которых требуют доработки для покрытия более сложных случаев нарушений правил. Для оценки качества реализованных детекторов использовались составленные вручную синтетические примеры как нарушающие данные правила, так и соответствующие им, а также проекты с открытым исходным кодом:

- aes [39] – реализация на Verilog симметричного блочного шифра AES – NIST FIPS 197 (Йоахим Стрембергсон);
- PicoRV32 [40] – оптимизированный по размеру процессор RISC-V (компания YosysHQ);
- black-parrot [41] – многоядерный процессор RISC-V с поддержкой Linux (компания Black Parrot);
- CVA6 [42] – 64-битный RISC-V процессор, реализующий конвейер с 6 стадиями, обладающий поддержкой Linux (компания OpenHW);
- SCR1 [43] – ядро микроконтроллера RISC-V (компания Syntacore);
- Ibex [44] – небольшое 32-битное ядро процессора RISC-V (компания lowRISC);
- VeeR EH1 [45] – 32-битное ядро RISC-V с расширениями (компания Western Digital, CHIPS Alliance);
- openC910 [46] – высокопроизводительный 64-битный многоядерный процессор, построенный на архитектуре RISC-V (компания XUANTIE-RV, Alibaba);
- ASIC bitcoin miner [47-48] – описание дизайна для майнинга биткоина (Мичиганский университет, США).

- OpenTitan [49] – платформа для создания заслуживающих доверия аппаратных компонентов (RoT, Root of Trust) (компания Google).

Тестирование осуществлялось на сервере с четырьмя 64-ядерными процессорами AMD EPYC 7662 с ограничением тактовой частоты в 2.9 ГГц под управлением операционной системы openSUSE Tumbleweed и объемом оперативной памяти 512GB.

Качество анализа оценивалось путем ручной разметки результатов анализа и подсчета точности, которая вычислялась как отношение истинных срабатываний к сумме истинных (TP, True Positive) и ложных (FP, False Positive) рассмотренных срабатываний. Детекторы с большим числом срабатываний оценивались по 100 случайным срабатываниям в различных файлах проекта. Результаты всех разработанных детекторов представлены в табл. 3.

Табл. 3. Результаты анализа разработанных детекторов системы статического анализа SVAN на проектах с открытым исходным кодом.
Table 3. SVAN static analysis system detectors open-source projects analysis results.

Проект	**Срабатываний**			**Точность**	**Время анализа, сек.**	**Количество строк, тыс.**
	Всего	Рассмотренных	FP			
aes	0	0	0	**100%**	3	3,5
PicoRV32	207	207	1	**99,5%**	4,8	7,5
black-parrot	342	174	1	**99,4%**	390	128,5
CVA6	926	536	6	**98,9%**	683	225,2
SCR1	64	64	1	**100%**	15,6	13,7
Ibex	88	84	1	**98,8%**	43,6	66,5
VeeR EH1	252	252	4	**98,4%**	137	18,3
openC910	1421	221	5	**97,7%**	4260	351,5
ASIC miner	1	1	0	**100%**	0,3	0,7
OpenTitan	670	264	5	**98,1%**	3605	587,2

Из результатов анализа следует, что разработанные детекторы демонстрируют высокую точность (около 98%) и приемлемую скорость работы (корреляция времени анализа и количества строк не является линейной, так как время анализа того или иного детектора больше зависит от сложности исходного кода схемы, чем от его размера). Основная часть ложных срабатываний связана с детекторами правил CASE01 («Каждый оператор case должен иметь метку по умолчанию») и SIGNAL07 («Не должно быть использований неинициализированного сигнала или переменной»).

В детекторе правила CASE01 был реализован подсчет перекрытия значениями меток всего диапазона значений селектора оператора `case`. В случае, когда метки полностью покрывают все значения селектора, метка по умолчанию не требуется, и, следовательно, диагностика является излишней. На момент тестирования не была учтена обработка do-not-care значений («?») в метках, что привело к некорректной диагностике ситуаций полного перекрытия в операторах `casex`/`casez`.

Детектор правила SIGNAL07 выдает ложные срабатывания при использовании переменных, которые определены как `typedef struct` и применяются в коде тестовой среды. Для этих

переменных может потребоваться нетривиальная инициализация, что не было учтено при первоначальной разработке детектора.

Все ложноположительные срабатывания были учтены и будут исправлены в рамках дальнейшей работы.

Компилятор SystemVerilog и все реализованные детекторы тестировались на разработанных вручную синтетических тестах (парсер SystemVerilog был также протестирован на автоматически сгенерированных тестах) и тестах из набора `sv-tests` [10].

Также проведено сравнительное тестирование реализованных детекторов с аналогичными детекторами встроенного линтера инструмента Verilator (см. раздел 3.2) на открытых проектах aes и PicoRV32. Следует отметить, что количество совпадающих детекторов в разработанной инфраструктуре статического анализа SVAN и Verilator невелико. Однако на совпадающих детекторах количество истинных срабатываний у SVAN выше, чем у Verilator.

В листинге 11 представлен пример нарушения правила ASSIGN03 («*Не должно быть блокирующих присваиваний в последовательностной логике*») в проекте PicoRV32 [40], которое не было обнаружено инструментом Verilator.

```
// picorv32.v
reg irq_delay;
reg irq_active;
reg [31:0] irq_mask;
reg [31:0] irq_pending;
reg [31:0] timer;
reg [31:0] next_irq_pending;
...
always @(posedge clk) begin
  ...
  if (!resetn) begin
    ...
    irq_active <= 0;
    irq_delay <= 0;
    irq_mask <= ~0;
    next_irq_pending = 0; // Нарушение правила ASSIGN03
    irq_state <= 0;
    timer <= 0;
    ...
  end
  ...
end
```

Листинг 11. Пример описания, нарушающего правило ASSIGN03 из проекта PicoRV32.
Listing 11. An example of a design violating the ASSIGN03 rule from the PicoRV32 project.

Использование блокирующих присваиваний в последовательностной логике приводит к избыточному усложнению логического синтеза из-за установления зависимостей триггеров и генерации дополнительной логики. В листинге 11 представлен процедурный блок, который был определен детектором, как реализующий последовательностную логику. Данный факт не вызывает сомнений, так как он реализует логику только на регистрах и привязан к положительному фронту тактового сигнала. В проекте PicoRV32 установлено два фрагмента описания, нарушающих правило ASSIGN03, которые не удалось обнаружить с помощью модуля линтинга инструмента Verilator, но обнаруженных детектором из инструмента SVAN. Все остальные срабатывания детекторов данного правила в этом проекте у инструментов совпадают.

Далее рассмотрены примеры нарушений других правил, найденные инструментом SVAN в проектах с открытым исходным кодом.

В листинге 12 представлен пример части описания схемы ASIC из проекта [47], который нарушает правило PORT10 (см. раздел 6.1): сигнал присоединен в обратном порядке.

В проекте black-parrot [41] также был найден интересный случай, нарушающий правило SIGNAL07 («*Не должно быть использований неинициализированного сигнала или переменной*»). Правило является достаточно актуальным, так как использование неинициализированных значений может привести к непредсказуемым результатам симуляции (моделирования). Рассмотрим пример кода в листинге 13.

```
// src/naive_design.sv
module naive_design(
    input clk,
    input reset,
    output logic done,
    output logic [255:0] H_out
);
...
endmodule


// testbench/sha_tb.sv
logic [0:255] result_test;
...
naive_design tested_module (
        .clk(clk),
        .reset(reset),
        .done(done_out_test),
        .H_out(result_test)
        // Нарушение правила PORT10:
        // подключение сигнала с обратным порядком
    );
...
```

Листинг 12. Пример описания, нарушающего правило PORT10 из проекта [47].
Listing 12. An example of a design violating the PORT10 rule from the [47] project.

```
// bp_common_rv64_pkgdef.svh
localparam dword_width_gp = 64;
// bp_be_pipe_int.sv
localparam num_bytes_lp = dword_width_gp >> 8;
...
logic [num_bytes_lp-1:0][7:0] orcb;
for (genvar i = 0; i < num_bytes_lp; i++)
  assign orcb[i] = {8{|rs1[8*i+:8]}};
  ...


always_comb
  unique case (decode.fu_op)
    // Нарушение правила SIGNAL07
    e_int_op_orcb : alu_result = orcb;
```

Листинг 13. Пример описания, нарушающего правило SIGNAL07 из проекта black-parrot.
Listing 13. An example of a design violating the SIGNAL07 rule from the black-parrot project.

В листинге 13 переменная `orcb` типа `logic` инициализируется побитово в цикле непосредственно после объявления, но, если заметить, что количество итераций цикла зависит от переменной `num_bytes_lp`, значение которой равно `64` деленное на `256` или `0`, что эквивалентно битовому сдвигу вправо на `8`. Это значит, что цикл, зависимый от генерируемой переменной `i`, будет развернут нулевое количество раз. Следовательно, ни один бит переменной `orcb` инициализирован не будет.

Данная ошибка была найдена разработчиками независимо от авторов статьи и была исправлена в недавнем изменении [50].

Также в этом проекте было найдено нарушение правила CASE03 («*Ширина выражения выбора в case не соответствует ширине вариантов выбора*»). Несовпадение битовой ширины селектора и перечисления может привести к отклонению от ожидаемых результатов симуляции или синтеза. Рассмотрим его подробнее в листинге 14.

```
// bp_common_rv64_pkgdef.svh
module bp_fe_icache
  always_comb
    for (integer i = 0; i < assoc_p; i++)
      case (tag_mem_pkt_cast_i.opcode)
        e_cache_tag_mem_set_tag:
          ...
        e_cache_tag_mem_set_state:
          ...
        {1'b0, e_cache_tag_mem_set_inval}: // Нарушение правила CASE03
          ...
      endcase


// bp_common_cache_engine_pkgdef.svh
typedef enum logic [2:0]
{
  e_cache_tag_mem_set_clear, e_cache_tag_mem_set_tag,
  e_cache_tag_mem_set_state, e_cache_tag_mem_set_inval,
  e_cache_tag_mem_read
} bp_cache_tag_mem_opcode_e;
```

Листинг 14. Пример описания, нарушающего правило CASE03 из проекта black-parrot.
Listing 14. An example of a design violating the CASE03 rule from the black-parrot project.

В листинге 14 выражение селектора оператора `case` является полем `opcode` переменной `tag_mem_pkt_cast_i` и имеет тип `bp_cache_tag_mem_opcode_e`, который соответствует перечисляемому типу с трехбитными значениями. Все, кроме одной, метки являются элементами данного перечисления. В последней метке используется выражение – конкатенация элемента перечисления и однобитного нуля. Соответственно, ширина последней метки равна четырем, что не соответствует ширине выражения селектора, которое равное трем.

О данном нарушении было сообщено разработчикам [51], а также было опубликовано его исправление [52].

При анализе проекта CVA6 в одном из используемых им компонентов, а именно в компоненте, реализующем контроллер прерываний `rv_plic` [53], было найдено нарушение правила LATCH01 («*Логику защелки не следует использовать в описании, кроме тех случаев, где это действительно необходимо*»). Появление непреднамеренной защелки обычно вызвано ошибками проектирования, например, когда отсутствует значение по умолчанию для переменной, переопределяемой в условном операторе (за исключением случаев, когда это явно подразумевается). Рекомендуется избегать появления защелок (кроме тех случаев, где это действительно необходимо) в описании схем, так как они не синхронизированы с тактовыми сигналами, что влечет к излишнему усложнению схемы. Рассмотрим листинг 15.

В листинге 15 приведен пример описания из автоматически генерируемого файла, в котором инженером-проектировщиком явно используется процедурный блок для описания комбинационной логики `always_comb`, но один из выходов, а именно однобитный `ip_re_o`, не получает начальное значение в процедурном блоке, но существует условное присваивание в этот выход по одной из меток оператора `case`. Для данного примера инструмент

логического синтеза сгенерирует защелку, то есть элемент последовательностной логики, что, вероятно, не соответствует намерению проектировщика.

Данная ошибка была найдена разработчиками независимо от авторов статьи и была исправлена в изменении [54].

```
// plic_regmap.sv
module plic_regs (
  output logic [30:0] prio_re_o,
  output logic [0:0] ip_re_o,
  output logic [1:0][30:0] ie_o)
  // предполагалась комбинационная логика
  always_comb begin
    prio_re_o = '0;
    ie_o = '0;
    ... // нет предварительной инициализации ip_re_o
    unique case(req_i.addr)
      ...
      32'hc000078: begin
        resp_o.rdata[2:0] = prio_i[30][2:0];
        prio_re_o[30] = 1'b1;
      end
      32'hc001000: begin
        resp_o.rdata[30:0] = ip_i[0][30:0];
        ip_re_o[0] = 1'b1; // Нарушение правила LATCH01
      end
...
```

Листинг 15. Пример описания, нарушающего правило LATCH01 из проекта CVA6.
Listing 15. An example of a design violating the LATCH01 rule from the CVA6 project.

8. Заключение

В данной работе представлена разработанная система статистического анализа SVAN, которая состоит из компилятора SystemVerilog, предназначенного для построения развернутого представления цифровых описаний аппаратуры на языках Verilog и SystemVerilog, и модуля анализа для проверки структурных и семантических свойств описаний. Также на основании опыта инженеров российских дизайн-центров электроники, был составлен список проблем, возникающих в ходе проектирования и разработки цифровых схем. Для каждой проблемы было разработано соответствующее правило. Эти правила затем были классифицированы по 18 категориям. Для 38 правил были разработаны детекторы с помощью инфраструктуры статического анализа SVAN, которые затем были протестированы на синтетических тестах и проектах описаний дизайнов с открытым исходным кодом. На синтетических примерах разработанные детекторы не имели ложных срабатываний, а на представленных проектах с открытым исходным кодом точность анализа составила порядка 98%. По сравнению с инструментом Verilator разработанные детекторы дают более точные и полные результаты.

В рамках дальнейшей работы планируется продолжить реализацию детекторов для разработанных правил, а также реализовать инфраструктуру для разработки детекторов, требующих анализ потока данных и символьное выполнение, с использованием инструментов CIRCT и KLEE.

Также планируется провести сравнительное тестирование разработанных детекторов с коммерческими анализаторами, обладающими детекторами аналогичных правил.

Список литературы / References

[1]. Riesgo T., Torroja Y., de la Torre E. Design Methodologies Based on Hardware Description Languages. IEEE Transactions on Industrial Electronics, vol. 46, no. 1, 1999. pp. 3-12. DOI: 10.1109/41.744370.

[2]. Earlham College. – Intel Pentium fdiv error, https://www.cs.earlham.edu/~dusko/cs63/fdiv.html, accessed 01.12.2024

[3]. Mcilroy R., Sevcik J., Tebbi T., Titzer B.L., Verwaest T. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv:1902.05178, 2019. https://arxiv.org/abs/1902.05178.

[4]. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017. 1315 p. DOI: 10.1109/IEEESTD.2018.8299595.

[5]. IEEE Standard for VHDL Language Reference Manual. IEEE Std 1076-2019. 673 p. DOI: 10.1109/IEEESTD.2019.8938196.

[6]. Смолов С.А. Обзор методов извлечения моделей из HDL-описаний. Труды ИСП РАН, том 27, вып. 1, 2015. с. 97-124. DOI: 10.15514/ISPRAS-2015-27(1)-6.

[7]. "IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[8]. EBMC – model checker for hardware designs, accessed 12.04.2024. https://www.cprover.org/ebmc, accessed 01.12.2024

[9]. Zhang R., Deutschbein C., Huang P., Sturton C. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018. pp. 815-827. DOI: 10.1109/MICRO.2018.00071.

[10]. sv-tests – SystemVerilog parsing synthetic test suit. https://github.com/chipsalliance/sv-tests, accessed 01.12.2024

[11]. SpyGlass: Early Design Analysis Tools for SoCs. https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html, accessed 01.12.2024

[12]. VCS: Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html, accessed 01.12.2024

[13]. IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE Std 1800.2-2020. 458 p. DOI: 10.1109/IEEESTD.2020.9195920.

[14]. Jasper Formal Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, accessed 01.12.2024

[15]. Xcelium logic simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, accessed 01.12.2024

[16]. Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/, accessed 01.12.2024

[17]. Icarus Verilog, https://steveicarus.github.io/iverilog/, accessed 01.12.2024

[18]. Verible – SystemVerilog parser and linter, https://github.com/chipsalliance/verible, accessed 01.12.2024

[19]. svlint – SystemVerilog linter, https://github.com/dalance/svlint, accessed 01.12.2024

[20]. SystemVerilog parser library, https://github.com/dalance/sv-parser, accessed 01.12.2024

[21]. Verilator – SystemVerilog simulation tool, https://github.com/verilator/verilator, accessed 01.12.2024

[22]. KLEE – symbolic virtual machine built on top of the LLVM compiler infrastructure, https://github.com/klee, accessed 01.12.2024

[23]. Surelog – SystemVerilog 2017 pre-processor, parser, elaborator, UHDM compiler, https://github.com/chipsalliance/Surelog, accessed 01.12.2024

[24]. Dargelas A., Zeller H. Universal Hardware Data Model. Workshop on Open-Source EDA Technology (WOSET), 2020. https://woset-workshop.github.io/PDFs/2020/a10.pdf

[25]. CIRCT – Circuit IR Compilers and Tools, https://github.com/llvm/circt, accessed 01.12.2024

[26]. Lattner C. et al. MLIR: A compiler infrastructure for the end of Moore's law // arXiv preprint arXiv:2002.11054. – 2020.

[27]. Mike Popoloski, slang - SystemVerilog Language Services, https://github.com/MikePopoloski/slang, accessed 01.12.2024

[28]. Mike Popoloski, slang-tidy - SystemVerilog linter, https://github.com/MikePopoloski/slang/tree/master/tools/tidy, accessed 01.12.2024

[29]. slang based frontend for Yosys, https://github.com/povik/yosys-slang, accessed 01.12.2024

[30]. Takemoto T. Joint activity for semiconductor R&D and role of Semiconductor Technology Academic Research Center (STARC). International Symposium on VLSI Technology, Systems, and Applications, 2001. pp. 7-10. DOI: 10.1109/VTSA.2001.934468.
[31]. Synopsis Inc., Mentor Graphics Corporation (2001). OpenMORE Assessment Program for Hard/Soft IP Version 1.0, http://www.openmore.com, accessed 01.12.2024
[32]. Hilderman V., Baghi T. Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware). Avionics Communications, 2007. 244 p.
[33]. Clang: a C language family frontend for LLVM, https://clang.llvm.org, accessed 01.12.2024
[34]. LLVM Language Reference Manual, https://llvm.org/docs/LangRef.html, accessed 01.12.2024
[35]. Moore Dialect, https://circt.llvm.org/docs/Dialects/Moore/, accessed 01.12.2024
[36]. Erhart M., Schuiki F., Yedidia Z., Healy B., Grosser T. Arcilator: Fast and cycle-accurate hardware simulation in CIRCT. LLVM Developers Meeting, 2023. https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf
[37]. The RTL Intermediate Language (RTLIL), https://yosyshq.readthedocs.io/projects/yosys/en/stable/yosys_internals/formats/rtlil_rep.html, accessed 01.12.2024
[38]. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them. Springer, 2010. 218 p.
[39]. AES – Hardware implementation of AES encryption algorithm, https://github.com/secworks/aes, accessed 01.12.2024
[40]. PicoRV32 – A Size-Optimized RISC-V CPU, https://github.com/YosysHQ/picorv32, accessed 01.12.2024
[41]. Black Parrot – A Linux-Capable Accelerator Host RISC-V Multicore, https://github.com/black-parrot/black-parrot, accessed 01.12.2024
[42]. The CORE-V CVA6 is an Application class 6-stage RISC-V CPU capable of booting Linux, https://github.com/openhwgroup/cva6, accessed 01.12.2024
[43]. SCR1 – RISC-V Core, accessed 10.06.2024. https://github.com/syntacore/scr1, accessed 01.12.2024
[44]. Ibex – RISC V Core, https://github.com/lowRISC/ibex, accessed 01.12.2024
[45]. VeeR EH1 Core, https://github.com/chipsalliance/Cores-VeeR-EH1, accessed 01.12.2024
[46]. OpenXuantie – OpenC910 Core, https://github.com/XUANTIE-RV/openc910, accessed 01.12.2024
[47]. ASIC Design for Bitcoin Mining, https://github.com/susansun1999/eecs570_final_project, accessed 01.12.2024
[48]. Sun Y., Yang H., Zhang W., Gu Y. ASIC Design for Bitcoin Mining. University of Michigan, 2021. https://zwtaoumich.github.io/paper/EECS570_Final_Report.pdf
[49]. OpenTitan: Open source silicon root of trust (RoT). https://opentitan.org/, accessed 01.12.2024
[50]. black-parrot, Hotfix: orbc typo, https://github.com/black-parrot/black-parrot/pull/1228/commits/87866ddfa37ab9d5df8e5951d3b7865b23676c77, accessed 01.12.2024
[51]. Case label selector mismatch. https://github.com/black-parrot/black-parrot/issues/1230, accessed 01.12.2024
[52]. Fix case label mismatch. https://github.com/black-parrot/black-parrot/pull/1232, accessed 01.12.2024
[53]. RISC-V Platform-Level Interrupt Controller. https://github.com/pulp-platform/rv_plic/, accessed 01.12.2024
[54]. Adjust codegen to fix synthesis latch. https://github.com/pulp-platform/rv_plic/pull/7, accessed 01.12.2024

Информация об авторах / Information about authors

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

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

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

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

Константин Николаевич КИТАЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Konstantin Nikolaevich KITAEV – student at MIPT, laboratory assistant at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Алексей Георгиевич ВОЛОХОВ – ведущий инженер отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Aleksey Georgievich VOLOKHOV – leading engineer at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Егор Викторович ДОЛГОДВОРОВ – студент МФТИ, Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Egor Viktorovich DOLGODVOROV – student at MIPT. Research interests: static analysis, compiler technologies, optimizations.

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

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

Артем Михайлович КОЦЫНЯК – научный сотрудник отдела технологий программирования ИСП РАН. Научные интересы: формальные методы, компиляторные технологии, модели вычислений, языки программирования.

Artem Mikhailovich KOTSYNYAK – researcher at Software Engineering department of ISP RAS. Research interests: formal methods, compiler technologies, models of computation, programming languages.

Дмитрий Олегович САМОВАРОВ – стажер-исследователь отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации, языки описания аппаратуры.

Dmitry Olegovich SAMOVAROV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations, hardware description languages.