

Федеральное государственное бюджетное учреждение науки Институт
системного программирования им. В. П. Иванникова Российской академии наук

На правах рукописи

Чуркин Ян Андреевич

**Метод и программные средства статического анализа
описаний цифровой аппаратуры для выявления дефектов
проектирования**

Специальность 2.3.5 —

«Математическое и программное обеспечение вычислительных систем,
комплексов и компьютерных сетей»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
кандидат технических наук
Бучацкий Рубен Артурович

Москва — 2026

Оглавление

	Стр.
Введение	6
Глава 1. Обзор методов и средств статического анализа описаний	
цифровой аппаратуры	14
1.1 Уровень регистровых передач как объект статического анализа . . .	14
1.2 Классификация выявляемых дефектов	16
1.3 Методы статического анализа RTL-описаний	17
1.3.1 Представления описания, используемые при анализе	18
1.3.2 Обход абстрактного синтаксического дерева	19
1.3.3 Анализ потоков данных	19
1.3.4 Абстрактная интерпретация и символьное выполнение . . .	21
1.3.5 Извлечение и анализ конечных автоматов	22
1.4 Анализ с привлечением сопутствующих проектных форматов . . .	23
1.5 Анализ корректности синхронизации	24
1.6 Методы на основе машинного обучения	24
1.7 Обзор инструментов статического анализа RTL-описаний	25
1.7.1 Коммерческие инструменты	25
1.7.2 Инструменты с открытым исходным кодом	26
1.7.3 Сравнительный анализ открытых инструментов	27
1.8 Требования к методу статического анализа и постановка задачи . .	29
1.9 Выводы по главе	31
Глава 2. Метод статического анализа описаний цифровой	
аппаратуры на основе внутреннего представления	32
2.1 Внутреннее представление RTL-описания, правила и детекторы . .	32
2.1.1 Уровни внутреннего представления	33
2.1.2 Правила, нарушения и детекторы	35
2.2 Набор правил статического анализа	37
2.3 Анализ на основе обхода абстрактного синтаксического дерева . .	39
2.4 Анализ путей в процедурных блоках	40
2.4.1 Классификация типов логики в процедурных блоках	41
2.4.2 Обход путей выполнения с интервальными ограничениями .	42
2.5 Анализ синтезируемости пользовательских примитивов	46

	Стр.
2.6 Анализ на основе логической сети	48
2.7 Анализ потоков данных	50
2.7.1 Учет установленных разрядов и источников сигналов	52
2.8 Предметно-ориентированный язык сопоставления узлов абстрактного синтаксического дерева	54
2.8.1 Мотивация и обзор существующих решений	54
2.8.2 Синтаксис языка сопоставителей	56
2.8.3 Семантика сопоставления и организация анализа	59
2.9 Расширяемость и поддержка стандартов	62
2.10 Выводы по главе	63
Глава 3. Метод извлечения и статического анализа конечных автоматов из RTL-описаний	65
3.1 Основные определения	65
3.2 Обзор подходов к извлечению и обоснование выбора	66
3.3 Метод извлечения конечных автоматов из сети логических элементов	68
3.4 Графовый анализ извлеченных автоматов	71
3.4.1 Анализ мертвых переходов	72
3.4.2 Анализ недостижимых состояний	73
3.4.3 Анализ тупиковых состояний и взаимоблокировок	73
3.5 Сопоставление результатов с исходным кодом	74
3.6 Обнаружение взаимодействующих автоматов	76
3.7 Обработка параметризованных автоматов	77
3.8 Выводы по главе	78
Глава 4. Метод интеграции сопутствующих проектных форматов . . .	79
4.1 Формат описания подсистемы питания UPF	79
4.1.1 Основные компоненты подсистемы питания	80
4.1.2 Эволюция стандарта	82
4.1.3 Аппаратная реализация и типы ошибок управления питанием	83
4.2 Построение единой информационной модели	84
4.2.1 Обработка библиотек стандартных ячеек Liberty	85
4.2.2 Интеграция формата задержек SDF	88

	Стр.
4.3 Интерпретация UPF-описания	90
4.4 Статический анализ модели подсистемы питания	91
4.5 Анализ RTL-описаний с привлечением библиотек Liberty	94
4.6 Выводы по главе	97
Глава 5. Программная реализация системы SVAN	99
5.1 Архитектура системы SVAN	99
5.1.1 Компилятор SystemVerilog	100
5.1.2 Интерпретатор UPF	102
5.1.3 Разбор библиотек Liberty	104
5.1.4 Анализатор формата SDF	105
5.1.5 Сериализация АСД в формат JSON	106
5.1.6 Фронтенд языка VHDL	106
5.1.7 Модуль анализа и интерфейс детекторов	108
5.1.8 Реализация подсистемы анализа конечных автоматов	116
5.2 Вклад в развитие открытых инструментов	116
5.2.1 Доработки компилятора SystemVerilog	117
5.2.2 Доработки фронтенда yosys-slang и инфраструктуры Yosys	118
5.3 Технические средства, вызовы и данные	118
5.4 Выводы по главе	119
Глава 6. Экспериментальная оценка системы SVAN	121
6.1 Методика экспериментальной оценки	121
6.2 Результаты анализа детекторов правил	123
6.3 Результаты анализа конечных автоматов	127
6.4 Оценка анализа с привлечением сопутствующих форматов	130
6.5 Оценка механизма сопоставителей узлов АСД	131
6.6 Сравнение с коммерческим аналогом	133
6.6.1 Применимость и ресурсоемкость	134
6.6.2 Точность и полнота	135
6.6.3 Разбор находок	138
6.6.4 Анализ конечных автоматов	139
6.7 Сравнение с открытыми средствами статического анализа	141
6.7.1 Условия и методика сравнения	142
6.7.2 Сравнение с Verilator	143

	Стр.
6.7.3 Сравнение со svlint	146
6.7.4 Сравнение с Verible	148
6.7.5 Сравнение с Surelog	150
6.7.6 Сравнение с Yosys	151
6.7.7 Сводные результаты	154
6.8 Внедрение результатов	155
6.9 Выводы по главе	156
Заключение	158
Список сокращений и условных обозначений	160
Словарь терминов	162
Список литературы	164
Список рисунков	172
Список таблиц	173
Приложение А. Каталог правил статического анализа	174
Приложение Б. Примеры дефектов, выявленных в проектах с открытым исходным кодом	183
Приложение В. Материалы сравнительного исследования с открытыми средствами	189
Приложение Г. Сведения о внедрении и регистрации результатов . . .	195

Введение

Современные цифровые интегральные схемы характеризуются высокой сложностью, при этом к ним предъявляются жесткие требования по корректности, надежности и безопасности функционирования. На ранних этапах проектирования одним из основных уровней описания цифровых устройств является уровень регистровых передач (Register Transfer Level, RTL), на котором формализуются структура и поведение аппаратуры с использованием языков описания аппаратуры (Hardware Description Language, HDL), таких как Verilog, SystemVerilog, VHDL и других. Создание RTL-описания является этапом логического проектирования цифровой схемы. Под дефектом проектирования в работе понимается свойство RTL-описания, которое приводит или может привести к некорректному функционированию схемы, к расхождению ее поведения при имитационном моделировании и логическом синтезе либо к нарушению свойств безопасности. Корректность RTL-описаний во многом определяет качество и надежность конечной аппаратной реализации. Дефекты, внесенные на этом этапе, нередко выявляются лишь на более поздних этапах проектирования: в процессе функциональной верификации, логического синтеза или даже после изготовления микросхемы. Исправление таких дефектов сопровождается значительными временными и финансовыми затратами. При этом широко применяемые методы верификации на основе имитационного моделирования (simulation-based verification) не обеспечивают покрытия всех возможных состояний системы, особенно в условиях роста объема RTL-описаний и усложнения взаимодействий между компонентами цифровых систем.

В этих условиях особую значимость приобретает статический анализ RTL-описаний, позволяющий выявлять дефекты проектирования без имитационного моделирования и логического синтеза. Статический анализ должен выявлять на ранних стадиях разработки структурные и семантические дефекты, а также нарушения свойств безопасности. Последнее особенно важно с учетом возрастающего внимания к безопасности цифровой аппаратуры, нарушение которой может приводить к некорректной работе вычислительных систем или к возникновению аппаратных уязвимостей.

Методы статического анализа программ (построение абстрактных синтаксических деревьев, анализ потоков данных, контроль типов) хорошо изучены

и широко применяются в компиляторах и инструментах анализа программного обеспечения. Однако их перенос на языки описания аппаратуры сопряжен с трудностями: такие языки описывают параллельно функционирующие компоненты с событийно-временной семантикой и четырехзначной логикой сигналов, сочетая поведенческие и структурные описания в одной модели, что усложняет построение внутренних представлений и моделей анализа.

Особую роль в RTL-описаниях играет управляющая логика, реализуемая в виде конечных автоматов. Недостижимые состояния и мертвые переходы, тупиковые состояния и взаимоблокировки не выявляются синтаксическими проверками и требуют анализа графа состояний и переходов, а в современных RTL-описаниях управляющая логика состоит из взаимодействующих автоматов. Автоматы записываются в различных стилях кодирования, и распознавание по синтаксическим шаблонам часть из них пропускает. Поэтому метод статического анализа должен обеспечивать извлечение конечных автоматов из RTL-описаний независимо от стиля кодирования, их проверку и поиск взаимодействующих автоматов.

RTL-описание не содержит части сведений, необходимых для анализа схемы: устройства библиотечных ячеек, распределения питания и задержек. Эти сведения задают сопутствующие проектные форматы Liberty, UPF и SDF, и без их привлечения не выявляются, в частности, ошибки подсистемы питания. Поэтому метод статического анализа должен использовать информацию сопутствующих проектных описаний.

Задачей статического анализа RTL-описаний занимаются как промышленные, так и академические коллективы. Коммерческие анализаторы компаний Synopsys (SpyGlass), Cadence (JasperGold), Siemens EDA (Questa One) и Aldec (ALINT-PRO) реализуют тысячи правил и применяются в промышленности, однако закрыты для расширения, неполно поддерживают стандарты SystemVerilog IEEE 1800-2017 и IEEE 1800-2023 и интегрируют сопутствующие форматы лишь частично. Открытые средства либо не поддерживают SystemVerilog в полном объеме, либо ограничиваются проверками по синтаксическому дереву и развернутой модели без анализа значений сигналов, и ни одно из них не сочетает анализ потоков данных с полной поддержкой SystemVerilog. К ним относятся Verilator (W. Snyder), Verible и Surelog (альянс CHIPS), svlint, компилятор slang (M. Popoloski), инфраструктура логического синтеза Yosys (C. Wolf), платформа Qihe (лаборатория PASCAL) и инфраструктура CIRCT

(проект LLVM). Академические группы Корнеллского университета (G. E. Suh, A. C. Myers, язык SecVerilog), Калифорнийского университета в Санта-Барбаре (T. Sherwood, B. Hardekopf, языки Caisson и Sapper), Калифорнийского университета в Сан-Диего (R. Kastner, Clepsydra) и Университета Флориды (F. Farahmandi, M. Tehranipoor, инструменты FSMx) сосредоточены на свойствах безопасности. Подходы групп Корнеллского и Калифорнийских университетов требуют аннотирования описания или перехода на расширенный язык, группа Университета Флориды извлекает автоматы из списков соединений вентильного уровня и RTL-описаний для оценки уязвимостей. Дефекты проектирования в стандартном SystemVerilog эти работы не затрагивают. В отечественных работах (С. А. Смоллов, А. С. Камкин, ИСП РАН) статический анализ RTL-описаний применялся для построения расширенных конечных автоматов в целях функциональной верификации. Таким образом, ни одно из существующих решений не сочетает полную поддержку SystemVerilog, расширяемый набор проверок, анализ конечных автоматов и интеграцию сопутствующих форматов.

В связи с этим актуальной является задача разработки и исследования метода статического анализа описаний цифровой аппаратуры, который опирается на набор правил, задающих проверяемые свойства описания, учитывает специфику языков описания аппаратуры при построении внутреннего представления, охватывает управляющую логику на уровне конечных автоматов и использует информацию сопутствующих проектных описаний. Метод должен быть реализован в виде расширяемой системы статического анализа, поддерживающей языки Verilog и SystemVerilog.

Целью диссертационной работы является разработка метода и программных средств статического анализа описаний цифровой аппаратуры на основе внутреннего представления проекта и набора правил, задающих проверяемые свойства описания, включая извлечение и анализ конечных автоматов и интеграцию сопутствующих проектных форматов, для выявления дефектов проектирования на ранних стадиях разработки цифровых интегральных схем.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Проанализировать существующие методы и программные средства статического анализа описаний цифровой аппаратуры и определить требования к методу анализа.

2. Разработать набор правил статического анализа, направленных на выявление структурных и семантических дефектов и нарушений свойств безопасности в описаниях цифровой аппаратуры, определив для каждого правила проверяемое свойство описания и сопоставив правилам техники анализа, достаточные для их проверки.
3. Разработать метод статического анализа описаний цифровой аппаратуры, основанный на внутреннем представлении проекта, обеспечивающий проверку разработанных правил без выполнения логического синтеза и имитационного моделирования.
4. Разработать метод извлечения и статического анализа конечных автоматов из RTL-описаний на основе сети логических элементов и графового представления для выявления независимо от стиля кодирования недостижимых состояний и мертвых переходов, тупиковых состояний и взаимоблокировок, а также взаимодействующих конечных автоматов с сопоставлением результатов анализа с исходным кодом.
5. Разработать метод интеграции сопутствующих проектных форматов (Liberty, UPF, SDF) в статический анализ на основе построения единой информационной модели.
6. Реализовать программные средства статического анализа в виде расширяемой системы в соответствии с разработанными методами и провести их экспериментальную оценку на открытых проектах цифровых СБИС, включая проверку корректности реализованных детекторов, анализ выявляемых дефектов, а также сравнение полученных результатов с существующими открытыми и коммерческими инструментами статического анализа цифровой аппаратуры.

Научная новизна:

1. Разработаны метод статического анализа описаний цифровой аппаратуры на основе внутреннего представления проекта, сочетающий обход абстрактного синтаксического дерева (в том числе с языком сопоставления узлов), классификацию типов логики и анализ путей в процедурных блоках с интервальными ограничениями, анализ потоков данных с отслеживанием диапазонов значений, учет установленных разрядов и источников сигналов, анализ логической сети и распознавание семантических шаблонов, и набор правил для выявления структурных и семантических дефектов и нарушений свойств безопасности, в котором правилам

сопоставлены техники анализа, достаточные для их проверки. Метод обеспечивает расширяемость набора проверок и раннее выявление дефектов без логического синтеза и имитационного моделирования.

2. Разработан метод извлечения и статического анализа конечных автоматов из RTL-описаний на основе сети логических элементов и графового представления, обеспечивающий независимо от стиля кодирования обнаружение недостижимых состояний и мертвых переходов, тупиковых состояний и взаимоблокировок, а также выявление взаимодействующих конечных автоматов с сопоставлением результатов анализа с исходным кодом.
3. Разработан метод интеграции сопутствующих проектных форматов (Liberty, UPF, SDF) в статический анализ описаний цифровой аппаратуры, основанный на построении единой информационной модели, связывающей развернутую модель описания с объектами сопутствующих описаний, включающий трансляцию библиотечных ячеек в поведенческое Verilog-представление и обеспечивающий статический анализ модели подсистемы питания (проверку необходимости, наличия и корректности стратегий изоляции и преобразования уровня напряжения), а также разбор и статическую проверку согласованности SDF-описания с развернутой моделью описания аппаратуры.

Теоретическая и практическая значимость. Теоретическая значимость работы заключается в разработке метода статического анализа описаний цифровой аппаратуры на основе внутреннего представления проекта, метода извлечения и статического анализа конечных автоматов из RTL-описаний на основе сети логических элементов и графового представления, а также метода интеграции сопутствующих проектных форматов в статический анализ. В работе определены понятия правила, нарушения и детектора, что позволяет сопоставить каждому правилу технику анализа, достаточную для его проверки. Тупиковые состояния и взаимоблокировки конечного автомата определены через терминальные сильно связанные компоненты его графа. Сформулирован критерий необходимости стратегий изоляции и преобразования уровня напряжения по состояниям наборов питания. Метод на основе внутреннего представления может быть распространен на другие языки описания аппаратуры.

Практическая значимость работы заключается в реализации расширяемой системы статического анализа описаний цифровой аппаратуры на языках Verilog

и SystemVerilog с поддержкой сопутствующих проектных форматов (Liberty, UPF, SDF), включающей набор детекторов правил, средства анализа конечных автоматов и анализатор модели подсистемы питания. Система позволяет выявлять дефекты проектирования в описаниях на языках Verilog и SystemVerilog на ранних стадиях разработки цифровых интегральных схем. Экспериментальная оценка системы проведена на открытых проектах цифровых СБИС в сравнении с коммерческим и открытыми инструментами, результаты подтверждают практическую применимость разработанных методов и их значимость для развития программного обеспечения САПР цифровых интегральных схем. Реализованная система статического анализа внедрена в научно-исследовательские и учебные проекты ИСП РАН, а также в опытно-конструкторские проекты АО «МНТЦ МИЭТ» для верификации описаний цифровой аппаратуры на ранних этапах проектирования. Переносимость архитектуры системы на другие языки описания аппаратуры подтверждена реализацией поддержки подмножества языка VHDL и построением единого абстрактного синтаксического дерева для описаний, сочетающих модули на языках Verilog и VHDL.

Методология и методы исследования. Результаты работы получены с использованием методов и моделей, применяемых при статическом анализе и верификации программ и аппаратных описаний. Математическую основу работы составляют теория графов, теория автоматов, теория формальных языков и теория множеств.

Соответствие паспорту научной специальности. Описания цифровой аппаратуры на языках Verilog, SystemVerilog и VHDL представляют собой программы на формальных языках со стандартизованной семантикой (IEEE 1364, IEEE 1800, IEEE 1076), поэтому разработанные методы являются методами статического анализа программ, учитывающими особенности этих языков. Область исследования соответствует паспорту научной специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей» по пункту 1 «Модели, методы и алгоритмы проектирования, анализа, трансформации, верификации и тестирования программ и программных систем» (методы статического анализа описаний, извлечения и анализа конечных автоматов и интеграции сопутствующих проектных форматов, алгоритмы детекторов правил и проверки стратегий подсистемы питания) и по пункту 2 «Языки программирования и системы программирования, семантика программ» (анализ на основе семантики языков Verilog и SystemVerilog, доработки компи-

лятора SystemVerilog, язык сопоставления узлов абстрактного синтаксического дерева, интерпретатор языка UPF и фронтенд языка VHDL).

Основные положения, выносимые на защиту:

1. Метод статического анализа описаний цифровой аппаратуры на основе внутреннего представления проекта и набора из 118 правил для раннего выявления структурных и семантических дефектов и нарушений свойств безопасности без логического синтеза и имитационного моделирования.
2. Метод извлечения и статического анализа конечных автоматов из RTL-описаний на основе сети логических элементов и графового представления, выявляющий независимо от стиля кодирования недостижимые и тупиковые состояния, мертвые переходы, взаимоблокировки и взаимодействующие автоматы.
3. Метод интеграции сопутствующих проектных форматов в статический анализ на основе единой информационной модели, связывающей описание аппаратуры с объектами Liberty, UPF и SDF, для анализа схем с библиотечными ячейками, модели подсистемы питания и SDF-описаний.
4. Программные средства статического анализа описаний цифровой аппаратуры, реализующие предложенные методы, и результаты их экспериментальной оценки на открытых проектах цифровых СБИС, включая сравнение с открытыми и коммерческими инструментами.

Апробация работы. Основные результаты работы докладывались и обсуждались на научных семинарах Института системного программирования им. В. П. Иванникова РАН, а также на следующих научных конференциях и семинарах:

- Открытая конференция ИСП РАН, Москва, 11–12 декабря 2024 г.
- Конференция EDA Connect 2025: Bridging Industry and Academia, Ереван, Армения, 13–14 марта 2025 г.
- 67-я Всероссийская научная конференция МФТИ, Долгопрудный, 31 марта – 5 апреля 2025 г.
- Programming Languages and Compiler Technologies Summit 2025, Сочи, 20–22 апреля 2025 г.
- Международная конференция «Иванниковские чтения», Иркутск, 26–27 июня 2025 г.
- Открытая конференция ИСП РАН, Москва, 9–10 декабря 2025 г.

- Заседание №3 экспертно-аналитической группы «Аппаратная безопасность», Москва, 16 декабря 2025 г.
- Международная конференция DVCon China 2026 (Design and Verification Conference and Exhibition), Шанхай, КНР, 12–13 мая 2026 г.
- Международная конференция USBEREIT 2026 (IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology), Зеленоград, 14–15 мая 2026 г.
- Коллоквиум молодых ученых SYRCoSE 2026 (Spring/Summer Young Researchers' Colloquium on Software Engineering), Этномир, 28–31 мая 2026 г.
- Международная конференция ITC-Asia 2026 (International Test Conference in Asia), Нинбо, КНР, 19–21 августа 2026 г.

Личный вклад. Все результаты работы получены лично автором.

Публикации. По теме диссертации опубликовано восемь научных работ. Работы [1–4] опубликованы в журналах, входящих в перечень рецензируемых научных изданий ВАК. Работы [5–8] опубликованы в сборниках трудов научных конференций, причем работы [5; 6] опубликованы в изданиях, индексируемых международной базой цитирования Scopus. Получены два свидетельства о государственной регистрации программ для ЭВМ [9; 10].

В работах [1; 5] автору принадлежат разработка набора правил, метода статического анализа, анализа потоков данных с отслеживанием диапазонов значений, метода анализа конечных автоматов и архитектуры разработанной системы статического анализа, а также экспериментальная оценка. В работе [4] автором выполнен обзор методов и средств анализа, в работе [6] автором разработан метод извлечения и статического анализа конечных автоматов. В работах [2; 7] автору принадлежат разработка единой информационной модели и анализа стратегий питания, в работе [3] автору принадлежат разработка правил проверки синтезируемости и метода проверки пользовательских примитивов, в работе [8] автору принадлежат постановка задачи и проектирование языка сопоставления узлов абстрактного синтаксического дерева.

Объем и структура работы. Диссертация состоит из введения, 6 глав, заключения, списка сокращений и условных обозначений, словаря терминов, списка литературы и 4 приложений. Полный объем диссертации составляет 197 страниц, включая 10 рисунков и 29 таблиц. Список литературы содержит 87 наименований.

Глава 1. Обзор методов и средств статического анализа описаний цифровой аппаратуры

В настоящей главе рассматривается уровень регистровых передач как объект статического анализа, систематизируются классы выявляемых дефектов и методы их поиска, выполняется обзор инструментов статического анализа и на основании их сравнения формулируются требования к методу и ставится задача исследования. Результаты главы опираются на выполненный автором систематический обзор предметной области.

1.1 Уровень регистровых передач как объект статического анализа

Современное проектирование цифровых интегральных схем осуществляется на нескольких уровнях абстракции, среди которых уровень регистровых передач (Register Transfer Level, RTL) занимает центральное положение: он является точкой сопряжения между функциональной спецификацией и конкретной аппаратной реализацией. На уровне регистровых передач цифровая схема описывается в терминах сигналов, регистров и логических операций между ними, без детализации реализации при помощи логических вентилей. Такое описание представляет передачу значений сигналов в рамках одного периода тактового сигнала. Основными языками RTL-описаний в промышленной практике являются Verilog (стандарт IEEE Std 1364-2005 [11]) и SystemVerilog (стандарты IEEE Std 1800-2017 [12] и IEEE Std 1800-2023 [13]).

Обработка RTL-описания начинается с элаборации (elaboration), то есть с построения развернутой (elaborated) модели, в которой вычислены значения параметров, созданы экземпляры модулей, разрешены иерархические имена и раскрыты блоки генерации (generate). Устройство развернутой модели описано в разделе 2.1.1.

Место уровня регистровых передач в маршруте проектирования цифровых интегральных схем и соотношение уровней абстракции показаны на рисунке 1.1.



Рисунок 1.1 — Маршрут проектирования цифровой интегральной схемы и уровни абстракции описания

С точки зрения статического анализа RTL-описания принципиально отличаются от программного кода по четырем ключевым аспектам.

1. *Параллелизм процедурных блоков.* Все процедурные блоки, непрерывные присваивания и экземпляры модулей семантически активны одновременно, что исключает применение классического анализа на основе единого графа потока управления.
2. *Событийно-временная семантика.* Семантика выполнения определяется событийной моделью: некорректное смешение блокирующих и неблокирующих присваиваний порождает труднообнаруживаемые функциональные ошибки, описанные, в частности, в [14].
3. *Четырехзначная логика сигналов.* Использование значений 0, 1, X, Z требует отслеживания распространения неопределенного значения X при анализе потоков данных.
4. *Совмещение поведенческого и структурного описания* существенно усложняет межмодульный анализ.

Перечисленные особенности порождают потребность в специализированных моделях анализа и отличают задачу анализа RTL-описаний от хорошо изученной задачи статического анализа программного обеспечения [15].

1.2 Классификация выявляемых дефектов

Под *дефектом проектирования* в работе понимается свойство RTL-описания, которое приводит или может привести к некорректному функционированию схемы, к расхождению ее поведения при имитационном моделировании и логическом синтезе либо к нарушению свойств безопасности. Дефекты, выявляемые при статическом анализе RTL-описаний, целесообразно разделить на три основные группы.

- **Структурные дефекты** — несоответствие разрядности сигналов, неподключенные порты, множественное управление сигналом. Такие дефекты выявляются на уровне абстрактного синтаксического дерева (АСД, Abstract Syntax Tree).
- **Семантические дефекты** — некорректные типы присваиваний, неполные списки чувствительности, ошибки пересечения доменов синхрони-

зации (Clock Domain Crossing, CDC), проблемы сброса. Их выявление требует анализа с учетом тактовых сигналов.

- **Нарушения свойств безопасности** — несанкционированные информационные потоки, отладочные механизмы в итоговой схеме, аппаратные побочные каналы, а также деление на ноль, выход за границы диапазонов и переполнения, приводящие в четырехзначной логике к появлению неопределенных значений либо к незаметному искажению результата. Подобные слабости цифровой аппаратуры систематизированы в аппаратном разделе каталога CWE [16].

Приведенная классификация напрямую соотносится с категориями правил, формулируемых в главе 2, и с применяемыми к ним техниками анализа. Структурным дефектам соответствуют, в частности, категории правил `port`, `case` и `width`, проверяемые обходом абстрактного синтаксического дерева. Семантическим дефектам соответствуют категории `assign`, `latch`, `reset`, `signal` и `fsm`, требующие анализа типов логики в процедурных блоках, анализа потоков данных либо извлечения и анализа графа состояний (глава 3). Нарушениям свойств безопасности соответствуют категории `arith` и `range` (деление на ноль, выход за границы, переполнения), требующие анализа распространения значений сигналов (интервальный анализ значений, раздел 2.7).

1.3 Методы статического анализа RTL-описаний

В разделе рассматриваются методы, применяемые в инструментах статического анализа RTL-описаний: обход АСД, анализ потоков данных, абстрактная интерпретация и символьное выполнение, извлечение и анализ конечных автоматов. Вначале сопоставляются представления описания, над которыми эти методы работают. Формальная верификация на основе модельной проверки (инструменты NuSMV [17] и EVMC) представляет собой самостоятельную область и выходит за рамки настоящего обзора. Отдельные элементы формальных методов рассматриваются здесь в той мере, в которой они применяются в инструментах статического анализа RTL.

1.3.1 Представления описания, используемые при анализе

Множество свойств, доступных проверке, определяется представлением описания, над которым работает средство анализа. Существующие средства, обзор которых приведен в разделе 1.7, используют представления четырех уровней (таблица 1). Чем ближе уровень к реализации схемы, тем больше сведений о ее структуре он содержит и тем слабее его связь с исходным текстом описания.

Таблица 1 — Уровни представления описания в средствах анализа

Уровень	Средства	Что содержит	Чего не содержит
Синтаксическое дерево	Verible, svlint, tree-sitter	конструкции языка и их расположение в тексте	значения параметров, иерархия экземпляров
Развернутая модель	slang, Surelog, Verilator	иерархия экземпляров, значения параметров, типы, соединения портов	значения сигналов, связи между сигналами через логику
Сеть логических элементов	Yosys, CIRCT	элементы схемы и связи между ними независимо от формы записи	несинтезируемые конструкции, соответствие элементов конструкциям исходного текста
Список соединений вентильного уровня	FSMx, FSMx-Ultra	реализация схемы на ячейках библиотеки	структура RTL-описания, связь с исходным текстом

Средства, работающие с синтаксическим деревом, ограничены проверками формы записи. Развернутая модель дает полную семантику описания, но сведения о значениях сигналов и связях между ними получают дополнительным анализом: базовый учет присвоенных разрядов и источников сигналов выполняет компилятор slang, а развитый анализ потоков данных реализован для языка Verilog в платформах Qihe [18] и Pyverilog, строящих для этого собственные промежуточные представления. Сеть логических элементов приводит различные формы записи одной и той же логики к единому виду, что делает ее удобной для извлечения конечных автоматов, однако строится только для синтезируемой части описания.

Общий недостаток перечисленных представлений состоит в том, что в них не интегрированы сведения сопутствующих проектных форматов (раздел 1.4). Библиотечные ячейки, подсистема питания и задержки не связаны в них с конструкциями RTL-описания, поэтому дефекты, для выявления которых нужны

эти сведения (например, отсутствие ячейки изоляции на пути данных через границу доменов питания), по таким представлениям выявить нельзя.

1.3.2 Обход абстрактного синтаксического дерева

Обход АСД составляет основу большинства существующих инструментов и позволяет выявлять структурные дефекты и нарушения стилевых соглашений. Синтаксические методы основаны на поиске в АСД характерных шаблонов. Их основным ограничением является то, что грамматика SystemVerilog 2017 и 2023 содержит более 1300 синтаксических правил с контекстно-зависимой семантикой, что объясняет ограниченную языковую поддержку в большинстве открытых инструментов.

1.3.3 Анализ потоков данных

Анализ потоков данных (Data Flow Analysis, DFA) вычисляет для каждой точки программы сведения о возможных состояниях вычисления: факты распространяются вдоль путей выполнения и объединяются в точках слияния, а решение находится итерациями до неподвижной точки [15; 19; 20]. Виды анализа различаются по следующим признакам.

- *Направление*. Прямой анализ распространяет факты по ходу выполнения (достигающие определения, распространение констант, диапазоны значений), обратный — против хода выполнения (живые переменные).
- *Объединение фактов*. Анализ вида «может» (may) считает факт установленным, если он верен хотя бы на одном пути к точке, анализ вида «должен» (must) — если он верен на всех путях.
- *Чувствительность*. Анализ, чувствительный к потоку управления, учитывает порядок операторов. Анализ, чувствительный к путям, различает факты, полученные при разных исходах условий ветвления. Анализ, чувствительный к контексту, различает вызовы одной подпрограммы из разных мест.

- *Область значений.* В классических задачах факт принадлежит конечному множеству (например, достигает ли определение точки). Анализ значений использует области абстрактной интерпретации [21]: константы, интервалы, отношения между переменными. Если область, как, например, интервальная, содержит бесконечные возрастающие цепочки, завершение итераций обеспечивается операцией расширения (widening).

Применение анализа потоков данных к RTL-описаниям имеет ряд особенностей. Описание состоит из параллельно работающих процессов и не имеет единого графа потока управления, поэтому анализ выполняется либо в границах отдельного процедурного блока, либо над графом зависимостей по данным (Data Dependence Graph, DDG) [22], охватывающим все блоки модуля и расширяемым до иерархического графа раскрытием экземпляров подмодулей. На графе зависимостей основаны, в частности, слайсинг HDL-описаний [23] и анализ пересечения доменов синхронизации. Неблокирующее присваивание обновляет значение в конце шага имитационного моделирования, поэтому вид присваивания и порядок операторов входят в анализируемое состояние. Сигналы многоразрядны и допускают присваивание отдельных разрядов, что требует учета фактов с точностью до разряда. Наконец, значения сигналов принадлежат четырехзначной логике, и отсутствие присваивания означает неопределенное значение X, а не ошибку времени выполнения.

В открытых средствах анализ потоков данных представлен ограниченно (раздел 1.7.2): компилятор slang отслеживает присвоенные разряды и источники сигналов, Verilator выявляет сигналы без источника и неиспользуемые сигналы, а платформы Qihe и Pyverilog реализуют анализ потоков данных только для Verilog 2001/2005. Диапазоны значений сигналов в описаниях на SystemVerilog ни одно из рассмотренных открытых средств не вычисляет. Разработанный в настоящей работе анализ (раздел 2.7) в приведенной классификации является прямым анализом в границах процедурного блока, чувствительным к потоку управления и, в пределах ограниченного числа хранимых альтернатив, к путям, с входом в вызываемые подпрограммы до заданной глубины. Его область значений — интервалы значений сигналов с учетом присвоенных разрядов.

1.3.4 Абстрактная интерпретация и символьное выполнение

Абстрактная интерпретация [21] в контексте RTL реализуется преимущественно в облегченных вариантах (анализ X-распространения, свертка констант, побитовая аппроксимация) ввиду высокой вычислительной стоимости полноценных абстрактных доменов. Немногочисленные применения абстрактной интерпретации непосредственно к HDL ориентированы на язык VHDL и иные задачи: вычисление надмножества достижимых состояний поведенческих VHDL-описаний [24; 25] и полуавтоматический вывод временных моделей процессоров для анализа наихудшего времени выполнения [26]. Выявление дефектов проектирования в описаниях на SystemVerilog эти работы не затрагивают. Символьное выполнение [27] применяется для потактового анализа путей с использованием решателей SMT (Satisfiability Modulo Theories), однако его масштабируемость ограничена взрывом числа путей в глубоких иерархиях [28]. Развитием данного направления являются работы по кэшированию SMT-запросов [29]. Отслеживание потоков информации (Information Flow Tracking, IFT) [30; 31] направлено на верификацию политик безопасности: каждому сигналу присваивается метка домена безопасности, а анализ верифицирует отсутствие запрещенных потоков данных между доменами. Развитую ветвь этого направления составляют языковые расширения и системы типов: SecVerilog [32] расширяет Verilog аннотациями типов безопасности со статической проверкой невмешательства с учетом временных каналов. Языки Caisson [33] и Sapper [34] обеспечивают соблюдение политик безопасности конструкциями языка с трансляцией в Verilog. Clepsydra [35] и VeriSketch [36] отслеживают временные потоки, а QIF-Verilog [37] применяет количественные метрики информационного потока. Обобщение направления отслеживания информационных потоков на уровне RTL дано в работе [38]. Все перечисленные подходы требуют аннотирования описания или перехода на расширенный язык и направлены на проверку свойств безопасности, тогда как разработанный в настоящей работе метод анализирует стандартный SystemVerilog без аннотаций и направлен на выявление дефектов проектирования.

1.3.5 Извлечение и анализ конечных автоматов

Методы извлечения и анализа конечных автоматов (Finite State Machine, FSM) относятся к фундаментальным задачам анализа управляющей логики, не имеющим прямых аналогов в программной инженерии [39; 40]. В отечественных работах предложен метод построения расширенных конечных автоматов по RTL-описанию средствами статического анализа [41], ориентированный на задачи функциональной верификации. В RTL-описаниях FSM реализуется совокупностью регистра состояния, логики вычисления следующего состояния и логики выходных сигналов (автоматы Мили или Мура). Некорректная реализация FSM порождает функциональные ошибки, не выявляемые синтаксическими проверками: недостижимые состояния и тупиковые состояния, нарушения полноты оператора выбора. Методы анализа ошибок в FSM условно делятся на графовые (строят явное или неявное представление графа состояний и проверяют его на достижимость и блокировки), структурные и формальные. В сложных RTL-описаниях управляющая логика реализована совокупностью взаимодействующих конечных автоматов, где выходные сигналы одного автомата управляют переходами другого. Анализ таких структур позволяет выявлять зависимости управления между автоматами. Отдельное направление составляет извлечение FSM для задач безопасности: инструменты FSMx [42] и FSMx-Ultra [43] извлекают автоматы из списков соединений вентильного уровня (gate-level netlist) для оценки уязвимостей управляющей логики, а работа [44] той же группы выполняет быстрое извлечение на уровне RTL также в целях оценки безопасности. Смежное направление составляет восстановление автоматов из списков соединений вентильного уровня при обратном проектировании и анализе стойкости запутывания (обфускации) управляющей логики [45; 46]. В отличие от них, метод, разработанный в настоящей работе (глава 3), ориентирован на качество проектирования (обнаружение недостижимых состояний, мертвых переходов, тупиковых состояний и взаимоблокировок) и работает на уровне RTL с сопоставлением результатов анализа с исходным кодом.

1.4 Анализ с привлечением сопутствующих проектных форматов

RTL-описание не содержит части сведений, необходимых для анализа схемы: устройства библиотечных ячеек, распределения питания и задержек реализованной схемы. Эти сведения задаются сопутствующими проектными форматами. Формат **Liberty** является промышленным стандартом описания технологических библиотек стандартных ячеек и содержит функциональное описание, временные характеристики и атрибуты тактовых сигналов каждой ячейки. Без этих сведений экземпляры библиотечных ячеек остаются для анализатора «черными ящиками», а с ними становится возможным, в частности, распознавание буферов тактовых цепей. Стандарт **UPF** (Unified Power Format, IEEE Std 1801 [47]) определяет топологию доменов питания, условия переключения и стратегии изоляции. Его интеграция позволяет верифицировать наличие ячеек изоляции на путях данных через границы доменов и корректность логики управления питанием. Формат не зависит от логики RTL-описания и связан с ним только иерархией объектов, поэтому ошибки задания подсистемы питания анализом одного RTL-описания не выявляются. Методы статической и формальной верификации подсистемы питания по UPF-описанию рассматриваются в [48; 49]. Стандарт **SDF** (Standard Delay Format, IEEE Std 1497 [50]) передает задержки, рассчитанные после синтеза или трассировки, в средства имитационного моделирования. Его записи адресуются экземплярам, путям задержек и временным проверкам описания, поэтому согласованность SDF-описания с HDL-описанием может быть проверена статически, до имитационного моделирования.

В существующих средствах каждое сопутствующее описание обрабатывается отдельно и для собственной задачи: Yosys разбирает Liberty для технологического отображения, Icarus Verilog аннотирует задержки SDF при имитационном моделировании, платформа SpyGlass содержит отдельный модуль анализа подсистемы питания по UPF (раздел 1.7). Ни один из рассмотренных открытых инструментов не реализует интеграцию перечисленных форматов на уровне RTL-анализа. В коммерческих инструментах она выполнена лишь частично.

1.5 Анализ корректности синхронизации

Отдельный класс задач анализа RTL-описаний составляет проверка корректности синхронизации, то есть анализ пересечений доменов синхронизации (Clock Domain Crossing, CDC) и доменов сброса (Reset Domain Crossing, RDC). Разработка соответствующих детекторов выходит за рамки защищаемых положений настоящей работы, однако методы анализа синхронизации составляют существенную часть области статического анализа цифровой аппаратуры и реализованы в рамках более широкой системы статического анализа.

Пересечением доменов синхронизации называется ситуация, когда связанные трактом данных регистр-источник и регистр-приемник принадлежат разным (асинхронным) доменам синхронизации. Основная проблема при этом — метастабильность триггера приемника, вызванная нарушением времен предустановки и удержания, и связанные с ней искажение и потеря данных. Пересечение защищается синхронизатором, вид которого зависит от типа передаваемых данных: цепочка из двух и более триггеров для псевдостатического одноразрядного сигнала, двусторонний синхронизатор (запрос и подтверждение) для управляющих сигналов и многоразрядных псевдостатических данных, двухпортовая память FIFO с указателями в коде Грея для потока данных.

Пересечением доменов сброса называется ситуация, когда связанные трактом данных регистры относятся к одному домену синхронизации, но к разным доменам сброса. RDC-анализ, в отличие от CDC-анализа, рассматривает все пути от источника к приемнику, включая проходящие через промежуточные триггеры.

1.6 Методы на основе машинного обучения

В 2023–2026 годах сформировалось самостоятельное направление применения методов машинного обучения (Machine Learning, ML) и больших языковых моделей (Large Language Model, LLM) к задачам анализа RTL-кода: обнаружению ошибок [51], диагностике кодирования [52], генерации верификационных утверждений [53], автоматическому исправлению кода [54] и отладке [55]. К преимуществам подходов на основе ML и LLM относятся способность выявлять

семантические дефекты без явного задания правил (закономерности извлекаются из обучающих данных) и потенциально низкая стоимость. Принципиальным ограничением, отличающим их от классического анализа, является отсутствие формальных гарантий полноты и корректности, недетерминизм и зависимость от обучающих данных. Таким образом, методы машинного обучения на текущем уровне развития следует рассматривать как перспективное дополнение к классическому детерминированному статическому анализу, а не как его замену.

1.7 Обзор инструментов статического анализа RTL-описаний

1.7.1 Коммерческие инструменты

SpyGlass [56] является де-факто промышленным стандартом статического анализа RTL. Платформа включает модули SpyGlass Lint для диагностики кодирования, SpyGlass CDC/RDC для анализа пересечения доменов синхронизации и доменов сброса, SpyGlass Power для анализа подсистемы питания с интеграцией UPF. Совокупное число предопределенных правил превышает 2000. К основным ограничениям относятся неполная поддержка ряда конструкций стандарта SystemVerilog IEEE 1800-2017 и IEEE 1800-2023, закрытость архитектуры и исходного кода, высокая стоимость лицензии.

JasperGold [57] делает формальную верификацию основным механизмом: модуль Superlint автоматически генерирует SVA-утверждения по RTL. Принципиальным преимуществом является наличие приложений верификации путей безопасности и корректности межмодульных соединений на формальной основе.

ALINT-PRO [58] наиболее близок к SpyGlass по целевому назначению и реализует правила каталогов STARC, RMM и DO-254. Существенным ограничением является отсутствие поддержки стандартов SystemVerilog IEEE 1800-2017 и IEEE 1800-2023.

Questa One [59] компании Siemens EDA разделяет функции между модулями Questa Lint (диагностика кодирования) и Questa Inspect (автоматические формальные проверки: мертвый код, тупиковые состояния конечных автоматов, арифметическое переполнение, выход за границы индексов).

1.7.2 Инструменты с открытым исходным кодом

Verilator [60] — открытый высокопроизводительный транслятор описаний на Verilog и SystemVerilog в код на C++/SystemC. В режиме проверки - `lint-only` выявляются несоответствия разрядности, сигналы без источника и неиспользуемые сигналы, комбинационные петли. Набор правил ограничен задачами компиляции и не допускает расширения без модификации исходного кода.

svlint [61] — инструмент диагностики кодирования для SystemVerilog на базе библиотеки `sv-parser`. Архитектура основана на шаблоне «обходчик АСД + набор правил». Анализ ограничен стилевыми и синтаксическими проверками.

slang [62] — библиотека для лексического разбора, синтаксического анализа, проверки типов и компиляции кода SystemVerilog, на текущий момент являющаяся единственной проходящей тесты из набора `sv-tests` и демонстрирующей наиболее полную среди открытых инструментов поддержку стандартов SystemVerilog. Ключевой особенностью является построение полностью аннотированного АСД и развернутой модели. В рамках инфраструктуры `slang` реализован инструмент `slang-tidy` для специализированных проверок.

Qihe [63] — программная платформа статического анализа Verilog, реализующая 22 базовых вида анализа (включая анализ потоков данных, помеченных данных (taint analysis) и достижимости) и 20 клиентских анализаторов. Ключевое ограничение — поддержка только Verilog 2001/2005.

Pyverilog [64] — инструмент на Python для анализа Verilog, включающий построение DDG и экспериментальный модуль идентификации FSM, но ограниченный поддержкой Verilog 2001/2005.

Surelog [65] — компилятор SystemVerilog 2017, реализующий полный цикл препроцессирования, синтаксического разбора и построения развернутой модели с сохранением результата в формате UHDM [66].

CIRCT [67] — компиляторная инфраструктура на основе MLIR/LLVM с набором специализированных диалектов для представления аппаратных описаний. Представление в форме SSA обеспечивает анализ потоков данных, готовых проходов диагностики RTL-описаний инфраструктура не содержит.

Verible [68] — открытый набор инструментов для SystemVerilog (средства проверки стиля и форматирования, языковой сервер), работающий без построения развернутой модели и ограниченный синтаксическими проверками.

Icarus Verilog [69] — открытый компилятор и система событийного моделирования Verilog 2005 с частичной поддержкой SystemVerilog. Ориентирован на имитационное моделирование и статических проверок качества RTL не выполняет, ограничиваясь диагностикой, которые выдаются при компиляции и построении развернутой модели. Поддерживает обратную аннотацию задержек в формате SDF при имитационном моделировании (системная задача `$sdf_annotate`).

Yosys [70] — открытая инфраструктура логического синтеза с внутренним представлением RTLIL. Собственные диагностики ограничены базовыми проверками целостности сети логических элементов (комбинационные петли, конфликтующие и отсутствующие источники, команда `check`). Специализированного анализа качества RTL не выполняет. Включает проход `fsm_detect` для обнаружения конечных автоматов в сети логических элементов, а также проход `dft_tag` для отслеживания потоков данных на основе распространения меток (анализ помеченных данных, применяемый в задачах верификации безопасности и обнаружения переполнений). Инфраструктура Yosys используется в настоящей работе как основа подсистемы извлечения автоматов (глава 3).

Отдельную категорию составляют инкрементальные синтаксические анализаторы, не выполняющие статического анализа, но участвующие в сравнении покрытия стандарта на наборе тестов `sv-tests`: **tree-sitter-verilog** [71] и **tree-sitter-systemverilog** [72] — грамматики для библиотеки инкрементального разбора `tree-sitter`, применяемые в редакторах кода для подсветки синтаксиса и навигации. Их возможности ограничены синтаксическим разбором без построения развернутой модели, проверки типов и семантического анализа, а покрытие конструкций SystemVerilog 2017 неполно.

1.7.3 Сравнительный анализ открытых инструментов

В таблице 2 приведено сравнение открытых инструментов анализа и обработки RTL-описаний по единой системе критериев (столбцы таблицы). В сравнение включены также Icarus Verilog, Yosys и tree-sitter, не выполняющие полноценного статического анализа (раздел 1.7.2). Последняя строка таблицы

приведена для сопоставления разработанной в настоящей работе системы с требованиями раздела 1.8.

Таблица 2 — Сравнение открытых инструментов анализа и обработки RTL-описаний с разработанной системой

Инструмент	Поддержка SV 2017, 2023	Анализ потоков данных	Анализ FSM	Расширяемость (интерфейс правил)	Интеграция форматов
Verible	частичная	нет	нет	нет	нет
svlint	почти полная	нет	нет	да	нет
Verilator	частичная	частично	нет	нет	нет
Surelog	почти полная	нет	нет	нет	нет
CIRCT	частичная	да (SSA)	нет	да	нет
Qihe	нет (Verilog)	да	нет	да	нет
Pyverilog	нет (Verilog)	да (DDG)	частично	нет	нет
slang	почти полная	частично	нет	да	нет
Icarus Verilog	частичная	нет	нет	нет	частично
Yosys	частичная	да	частично	да	частично
tree-sitter (Verilog)	нет (Verilog)	нет	нет	нет	нет
tree-sitter (SV)	частичная	нет	нет	нет	нет
SVAN (настоящая работа)	полная	да	да	да	да*

* Для Liberty и UPF реализована полная интеграция, для SDF — разбор и проверка согласованности с моделью. Значение «частично» в столбце анализа автоматов для Yosys и Pyverilog — извлечение автоматов без анализа дефектов. Значение «частично» в столбце интеграции форматов для Yosys — разбор Liberty (read_liberty) для технологического отображения, для Icarus Verilog — обратная аннотация задержек в формате SDF при имитационном моделировании (\$sdf_annotate). Ни тот, ни другой не поддерживают полного набора Liberty, UPF и SDF для целей анализа.

Столбец расширяемости отражает наличие интерфейса добавления пользовательских правил без изменения исходного кода инструмента. Расширяемость обеспечивается у CIRCT компиляторной инфраструктурой MLIR, у Qihe механизмом клиентских анализаторов, у slang инструментом slang-tidy, у svlint механизмом подключаемых правил (загружаемых в виде отдельных библиотек), у Yosys системой проходов и подключаемых модулей.

Сравнительный анализ позволяет сформулировать пять ключевых выводов.

1. Полная поддержка SystemVerilog 2017 и 2023 отсутствует у большинства инструментов. Единственным открытым синтаксическим анализатором, проходящим тесты набора sv-tests и обеспечивающим почти полную обработку синтаксиса SystemVerilog 2017 и 2023, является slang.

2. Открытые инструменты в подавляющем большинстве ограничиваются обходом АСД без анализа потоков данных и межмодульного анализа. Исключениями являются Qihe и Pyverilog (только для Verilog), инфраструктура CIRCT (представление SSA) и Yosys (отслеживание потоков данных на сети логических элементов средствами анализа помеченных данных). При этом ни один из них не сочетает анализ потоков данных с полной поддержкой SystemVerilog 2017.
3. Открытые средства выявляют лишь отдельные семантические дефекты, систематическое выявление семантических дефектов и нарушений безопасности реализовано в коммерческих инструментах и отдельных академических прототипах. Ни один открытый инструмент не обеспечивает одновременного покрытия всех трех классов дефектов для SystemVerilog.
4. Ни один открытый инструмент не реализует полноценного анализа конечных автоматов (распознавание, построение графа состояний, анализ достижимости, обнаружение тупиковых состояний и анализ взаимодействующих автоматов).
5. Поддержка стандартов кодирования RTL (STARC [73], OpenMORE [74]) и наборов правил для процессов сертификации авиационной аппаратуры по DO-254 [75], а также совокупная интеграция форматов Liberty, UPF и SDF для целей анализа реализованы преимущественно в коммерческих инструментах. Из открытых средств лишь Yosys разбирает Liberty, причем для технологического отображения при синтезе, а не для анализа качества RTL. Поскольку такие процессы требуют проверяемого соблюдения стандартов кодирования, отсутствие их поддержки ограничивает применимость открытых решений в сертифицируемых разработках.

1.8 Требования к методу статического анализа и постановка задачи

На основании выявленных ограничений сформулированы требования к перспективному методу статического анализа RTL-описаний.

- Поддержка Verilog и SystemVerilog в полном объеме синтезируемого подмножества, включая поддержку препроцессора, параметрических конструкций, интерфейсов и пакетов.

- Выявление структурных и семантических дефектов на основе анализа внутреннего представления без выполнения синтеза или имитационного моделирования.
- Поддержка проверок нарушений свойств безопасности с возможностью расширения отслеживанием информационных потоков.
- Извлечение конечных автоматов из RTL-описаний и анализ графа переходов: проверка достижимости, обнаружение тупиковых состояний, идентификация и анализ взаимодействующих автоматов.
- Интеграция информации из форматов Liberty, UPF и SDF для повышения точности анализа.
- Высокая расширяемость: добавление новых правил без модификации ядра системы.
- Масштабируемость: работоспособность на промышленных RTL-описаниях объемом до миллиона строк.

Из обзора следуют и требования к внутреннему представлению, над которым выполняется анализ. Ни одно из представлений раздела 1.3.1 в отдельности им не удовлетворяет: синтаксическое дерево не содержит значений параметров и иерархии экземпляров, развернутая модель не содержит сведений о значениях сигналов и связях между ними, сеть логических элементов и список соединений вентильного уровня не охватывают несинтезируемые конструкции и теряют соответствие исходному тексту, а сведения сопутствующих форматов хранятся отдельно от представления RTL-описания. Поэтому в работе внутреннее представление строится как совокупность связанных уровней: развернутая модель RTL-описания, строящиеся над ней внутренние структуры (глава 2), граф конечного автомата (глава 3) и единая информационная модель, связывающая развернутую модель с объектами сопутствующих описаний (глава 4). *Внутренними структурами* в работе называются вспомогательные структуры данных, которые система строит над развернутой моделью для техник анализа и которые сохраняют привязку к ее узлам, например классификация типов логики процедурных блоков и логическая сеть. Все уровни привязаны к узлам развернутой модели, что позволяет локализовать нарушение любого правила в исходных текстах описаний.

Сформулированные требования определяют направление диссертационного исследования. В последующих главах представлены: набор правил, задающих проверяемые свойства описания, метод статического анализа на основе внут-

ренного представления и предметно-ориентированный язык сопоставления узлов абстрактного синтаксического дерева для разработки детекторов (глава 2), метод извлечения и статического анализа конечных автоматов (глава 3), метод интеграции сопутствующих проектных форматов (глава 4), а также программная реализация системы статического анализа SVAN (глава 5) и ее экспериментальная оценка (глава 6).

1.9 Выводы по главе

В главе выполнен обзор методов и средств статического анализа описаний цифровой аппаратуры. Выделены четыре особенности RTL-описаний как объекта анализа: параллелизм процедурных блоков, событийно-временная семантика, четырехзначная логика сигналов и совмещение поведенческого и структурного описания в одной модели. Систематизированы классы выявляемых дефектов, и каждому классу сопоставлены категории правил и техники анализа. Показано, что представления описаний, используемые существующими средствами, не интегрируют сведения сопутствующих проектных форматов, а коммерческие инструменты закрыты для расширения и неполно поддерживают стандарты SystemVerilog. Ни одно открытое средство не сочетает анализ потоков данных с полной поддержкой SystemVerilog, не выявляет дефекты конечных автоматов и не использует форматы Liberty, UPF и SDF для целей анализа. На основании обзора сформулированы требования к методу статического анализа и к его внутреннему представлению, которое строится как совокупность связанных уровней, привязанных к узлам развернутой модели. Результаты главы соответствуют первой задаче диссертационной работы.

Глава 2. Метод статического анализа описаний цифровой аппаратуры на основе внутреннего представления

В настоящей главе представлен метод статического анализа описаний цифровой аппаратуры, основанный на внутреннем представлении и правилах, задающих проверяемые свойства описания, без выполнения логического синтеза и имитационного моделирования. Вначале описывается внутреннее представление, над которым выполняется анализ (раздел 2.1), и вводятся понятия правила, нарушения и детектора, обеспечивающие расширяемость набора проверок (раздел 2.1.2). Далее описывается разработанный набор правил, его классификация по категориям и по технике анализа, достаточной для проверки правила (раздел 2.2). В разделах 2.3–2.7 рассматриваются техники проверки правил, различающиеся вычислительной стоимостью и множеством выявляемых свойств. Каждая техника описывается на уровне алгоритма и иллюстрируется представительным правилом. Затем представлен предметно-ориентированный язык сопоставления узлов АСД, предназначенный для разработки детекторов (раздел 2.8), и рассмотрены расширяемость метода и поддержка стандартов языка (раздел 2.9).

2.1 Внутреннее представление RTL-описания, правила и детекторы

Цифровая схема может быть описана с помощью логических вентилях и соединений между ними (gate-level netlist). Учитывая высокую сложность современных интегральных схем (10^7 – 10^9 вентилях), такое описание практически невозможно выполнить вручную, поэтому в проектировании цифровых схем получил распространение подход к описанию модели аппаратуры на уровне регистровых передач. Языки описания аппаратуры, такие как Verilog и SystemVerilog, используют синтаксис, схожий с синтаксисом процедурных языков программирования, однако главное отличие состоит в модели вычислений: в RTL-модели операции, которые должны выполняться одновременно в конце такта, описываются в императивном стиле наряду с последовательными операциями.

Основной структурной единицей описания является модуль (module). Каждый модуль имеет интерфейс, включающий входные (input) и выходные (output) сигналы, и тело, в котором описывается функциональная логика. Тело модуля может содержать экземпляры других модулей и фиксированное число процессов (always), выполняющихся параллельно в рамках одного модуля. Каждый процесс имеет список чувствительности (sensitivity list) и тело, содержащее последовательность операторов. Важной особенностью является специфический режим выполнения процессов: во время имитационного моделирования каждый процесс ожидает наступления события, и при наступлении события, перечисленного в списке чувствительности процесса, выполняется его тело. Пример простого описания схемы на языке SystemVerilog приведен в листинге 2.1.

Листинг 2.1: Пример простого описания схемы на SystemVerilog

```

1 module SAMPLE (
2     input logic RST,    // входной сигнал сброса
3     input logic CLK,    // входной тактовый сигнал
4     input logic D,      // входной одноразрядный сигнал
5     output logic OUT    // выходной одноразрядный сигнал
6 );
7     always @(posedge CLK or negedge RST) begin
8         if (!RST)      OUT <= 1'b0;    // параллельное присваивание
9         else if (D)     OUT <= ~OUT;
10        else           OUT <= OUT;
11    end
12 endmodule

```

2.1.1 Уровни внутреннего представления

Внутреннее представление, над которым выполняется анализ, строится в несколько этапов. После препроцессирования и лексического анализа синтаксический анализатор, работающий методом рекурсивного спуска, строит *дерево разбора* (parse tree), отражающее вывод исходного текста по грамматике языка. На отдельном этапе по дереву разбора строится *абстрактное синтаксическое дерево* (АСД), в котором отсутствуют узлы, связанные только с правилами вывода грамматики. Узлы АСД соответствуют конструкциям языка описания (модулям, переменным, процедурным блокам и т.п.) и содержат атрибуты, специфичные

для конструкции, включая диапазон в исходном тексте (имя файла, строку и столбец начала и конца соответствующего конструкции фрагмента). Для единообразия анализа схожие по семантике, но различающиеся синтаксически конструкции (например, объявления переменных и портов, различные виды процедурных блоков) представляются узлами одного уровня абстракции. Каждый узел АСД относится к одной из базовых категорий. Основными являются символ (*symbol*), оператор (*statement*) и выражение (*expression*), причем типы данных (*type*) представлены как частный вид символов. Всего компилятор выделяет восемь базовых категорий узлов, перечисленных в разделе 2.8.

После построения АСД строится развернутая модель. Сначала строится *предварительная* (pre-elaborated) модель — совокупность определений модулей, интерфейсов и пакетов, извлеченных из разобранных исходных файлов и связанных по именам, но без создания экземпляров верхнего уровня, без вычисленных значений параметров и конструкций `defparam`, без раскрытых блоков генерации и установленных соединений портов. На основе предварительной модели строится *развернутая* (elaborated) модель — целостное описание, в котором вычислены все значения параметров (в том числе зависящих от других параметров и иерархических имен), раскрыты блоки генерации, подставлены тела вложенных модулей, соединены порты и разрешены иерархические имена. Глубина рекурсивной подстановки контролируется для корректной обработки рекурсивно вложенных модулей. В развернутой модели присутствует ровно одно определение каждого элемента. Детекторы получают на вход развернутую модель и обходят тела модулей через их экземпляры, при необходимости обращаясь к синтаксису конструкций, поэтому модули, не имеющие экземпляров в иерархии модуля верхнего уровня, не анализируются (исключение составляют проверки текста комментариев). Развернутая модель необходима и для анализа свойств, затрагивающих несколько модулей (межмодульного анализа).

Перечисленные уровни строит компилятор SystemVerilog (раздел 5.1.1), и сами по себе они не отличаются от представлений других средств, строящих развернутую модель (раздел 1.3.1). Разработанное в работе внутреннее представление включает, помимо них, внутренние структуры, то есть вспомогательные структуры данных, которые строятся над развернутой моделью по заявкам детекторов и сохраняют привязку к ее узлам: классификацию типов логики процедурных блоков (раздел 2.4.1), логическую сеть (раздел 2.6), состояния анализа потоков данных с диапазонами значений и маски установленных

разрядов (раздел 2.7), а также граф конечного автомата (глава 3) и единую информационную модель (глава 4). Уровни внутреннего представления и внутренние структуры приведены на рисунке 2.1.

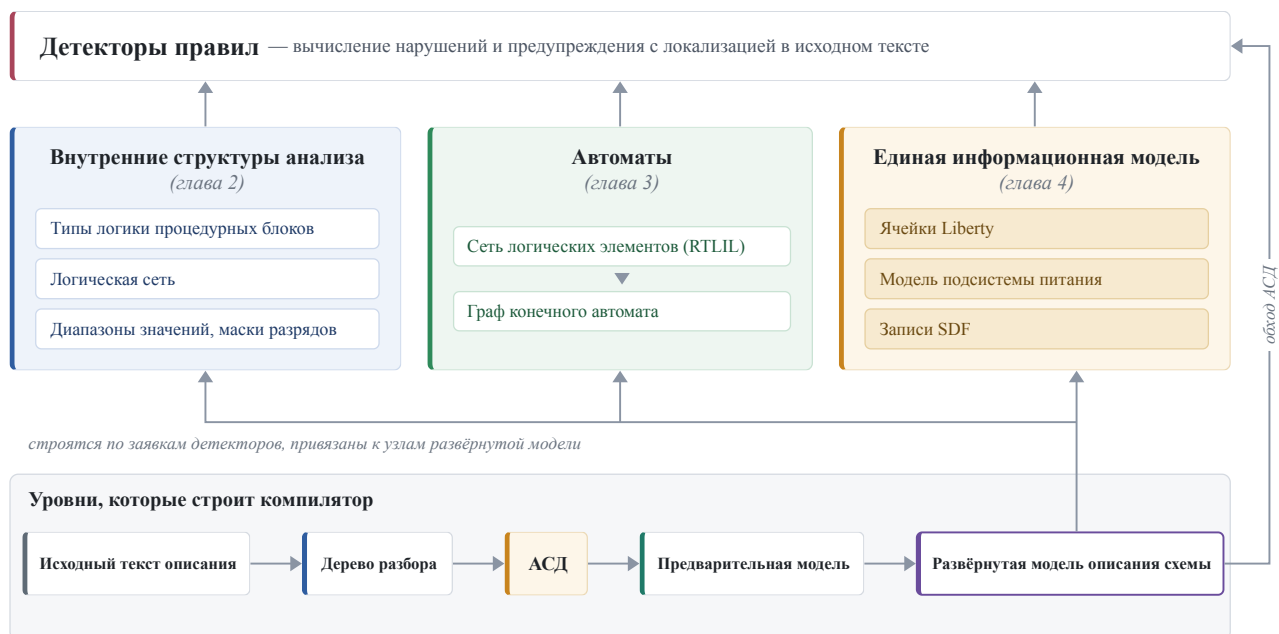


Рисунок 2.1 — Внутреннее представление: уровни, которые строит компилятор, и внутренние структуры

2.1.2 Правила, нарушения и детекторы

Введем понятия, на которых строится метод. *Конструкцией описания* будем называть синтаксически законченный фрагмент описания, которому соответствует узел (поддерево) внутреннего представления (например, объявление сигнала, создание экземпляра модуля, процедурный блок или отдельный оператор).

Правило r задает проверяемое свойство описания — условие, которому должно удовлетворять описание или его части. *Нарушением* правила r называется конкретное несоответствие описания этому свойству. Нарушение может относиться к отдельной конструкции (например, использование несинтезируемого оператора), к отсутствию ожидаемой конструкции (например, отсутствие ветви `default` в операторе `case`) или к отношению между несколькими конструкциями (например, наличие нескольких источников у сигнала без третьего состояния). Нарушение правила указывает на потенциальный дефект либо на нежелательную с точки зрения проектирования, верификации или синтеза конструкцию. При этом

оно не означает обязательной функциональной некорректности описания, поскольку часть правил относится к стилю кодирования и устойчивым инженерным практикам, а не к функциональной семантике схемы.

Обозначим через P анализируемое описание, то есть RTL-описание вместе с сопутствующими описаниями, если они заданы (глава 4), а через $T(P)$ — совокупность построенных по нему внутренних представлений: представления раздела 2.1, а для отдельных групп правил также построенные по ним модели: граф конечного автомата (глава 3) и единая информационная модель, связывающая развернутую модель с объектами сопутствующих описаний (глава 4). Через $V_r(P)$ обозначим множество всех нарушений правила r в описании P . С каждым нарушением $v \in V_r(P)$ связана локализация $\text{loc}(v)$ — диапазон в исходном тексте (имя файла, строка и столбец начала и конца фрагмента), в котором нарушение диагностируется.

Детектором правила r называется алгоритм d_r , который по внутреннему представлению $T(P)$ вычисляет множество $\hat{V}_r(P)$ обнаруживаемых нарушений правила r и формирует для каждого из них диагностическое предупреждение

$$d_r(T(P)) = \{ w(v) \mid v \in \hat{V}_r(P) \}, \quad w(v) = (\text{loc}(v), r, m_r(v)),$$

где предупреждение $w(v)$ содержит локализацию нарушения $\text{loc}(v)$, идентификатор нарушенного правила r и диагностическое сообщение $m_r(v)$.

Проверка ряда свойств алгоритмически неразрешима или вычислительно неприемлема в точной постановке, поэтому вычисляемое детектором множество $\hat{V}_r(P)$ в общем случае лишь аппроксимирует $V_r(P)$: элементы множества $\hat{V}_r(P) \setminus V_r(P)$ являются ложными срабатываниями (предупреждениями о несуществующих нарушениях), а элементы множества $V_r(P) \setminus \hat{V}_r(P)$ — пропусками (необнаруженными нарушениями). Точность детекторов оценивается экспериментально (глава 6).

Каждому правилу набора $R = \{r_1, \dots, r_n\}$ сопоставляется реализующий его детектор, что дает множество детекторов $D = \{d_{r_1}, \dots, d_{r_n}\}$. Роль детектора в смысле этого определения выполняет детектор модуля анализа, диагностика компилятора или специализированный анализатор (раздел 2.2). Результатом статического анализа описания P является объединение предупреждений активированных детекторов

$$\text{Diag}(P) = \bigcup_{d \in D'} d(T(P)), \quad D' \subseteq D,$$

где D' — подмножество детекторов, активированных в конфигурации анализа. Такая постановка обеспечивает *расширяемость* метода: добавление нового правила сводится к реализации соответствующего детектора d_r без модификации ядра анализа. Техника, которой реализуется детектор, выбирается по характеру проверяемого свойства (раздел 2.2). Проверка отдельных правил состоит в распознавании *семантического шаблона*, то есть заданной структуры конструкции, соответствие которой определяет ее функциональное назначение (например, строк таблицы состояний пользовательского примитива, задающих переходы защелки или D-триггера). Для таких правил применяется распознавание шаблонов (раздел 2.5).

2.2 Набор правил статического анализа

Набор правил является инженерным обобщением практики потребителей системы и каталогов коммерческих анализаторов и формировался в несколько этапов.

1. *Сбор свойств.* От инженеров дизайн-центров, потенциальных потребителей системы (АО «НТЦ «Атлас», АО «МНТЦ МИЭТ»), и из опыта разработчиков, работающих с Verilog и SystemVerilog, собраны типовые проблемы, возникающие при проектировании, верификации и синтезе, в виде неформальных требований к описанию.
2. *Уточнение до проверяемого свойства.* Каждое требование проанализировано с точки зрения семантики языка: выделены случаи, в которых конструкция допустима стандартом, и случаи, в которых она приводит к дефекту. При необходимости требование разделено на несколько свойств с явными исключениями.
3. *Формулировка правила.* Свойство записано как правило в смысле раздела 2.1.2, снабжено идентификатором и отнесено к категории по аспекту языка или процесса разработки.
4. *Сверка с существующими каталогами.* Формулировки сопоставлены с правилами стандарта кодирования STARC [73] и с наборами проверок коммерческих анализаторов. По результатам сверки формулировки уточнены, а дублирующие правила объединены.

5. *Тесты.* Для каждого правила подготовлены примеры описаний, нарушающих его и соответствующих ему. Они образуют набор синтетических тестов детекторов (глава 6).

Путь от неформального требования до правила иллюстрирует требование разработчиков «сигнал не должен управляться из нескольких мест». Анализ семантики показал, что несколько источников допустимы для цепей с третьим состоянием, что повторное присваивание внутри одного процедурного блока на взаимоисключающих путях (задание значения по умолчанию с последующим переопределением) является принятой практикой и что опасность представляют следующие ситуации: конфликт источников, гонка присваиваний в одном такте и зависимость результата от порядка присваиваний триггеру. В итоге требование распалось на четыре правила с собственными исключениями: SIGNAL09 (сигнал без третьего состояния должен иметь единственный источник), SIGNAL04 (сигналу не должно выполняться несколько присваиваний в пределах одного такта имитационного моделирования), ASSIGN02 (триггеру в последовательностном блоке не должно присваиваться значение несколько раз, кроме асинхронного сброса) и ASSIGN05 (присваивание не должно переопределять предыдущее в той же области видимости, кроме задания значения по умолчанию).

В результате сформулирован список из 118 правил: 114 правил для языков описания аппаратуры Verilog и SystemVerilog, 2 правила корректности модели подсистемы питания и 2 правила корректности пересечения доменов синхронизации и сброса. Итоговый список правил классифицирован по 19 категориям, соответствующим различным аспектам языка и процесса разработки (в скобках указано число правил): arith (2), assign (6), case (10), design (5), fsm (3), latch (5), loop (3), port (14), power (2), pp (2), range (1), reset (8), signal (19), sync (2), synth (13), timing (2), type (4), variable (8) и width (9). Назначение категорий раскрыто в заголовках каталога правил (приложение А). Большинство правил набора имеет аналоги в коммерческих анализаторах (глава 6).

Для всех 114 правил Verilog/SystemVerilog в рамках работы реализована проверка (полный каталог приведен в приложении А): 89 правил проверяются детекторами модуля анализа, 22 правила — диагностиками используемого компилятора (глава 5). Правила категории fsm (3 правила) проверяются отдельной подсистемой анализа конечных автоматов (глава 3), правила категории power — анализатором модели подсистемы питания (глава 4). Правила категории sync проверяются отдельным анализатором пересечений доменов синхронизации и сбро-

са, входящим в состав системы SVAN (раздел 5.1). Разработка этого анализатора в настоящую работу не входит, поэтому он далее не рассматривается.

По характеру проверяемого свойства правила разделяются на группы, что определяет применимую к ним технику анализа. *Структурными* будем называть правила, нарушение которых определяется по структуре описания (наличию, расположению, атрибутам и взаимосвязям конструкций) без вычисления значений сигналов и анализа достижимости путей (например, обратный порядок подключения шин или отсутствие ветви `default` в операторе `case`). Проверка остальных правил требует дополнительной информации: о реализуемом в процедурном блоке типе логики и порядке операторов на путях выполнения (раздел 2.4), о замкнутых путях передачи значений между сигналами (раздел 2.6), о диапазонах значений сигналов и достижимости операторов (раздел 2.7) либо об источниках разрядов сигналов в экземпляре и по иерархии (раздел 2.7.1). Правила, проверка которых сводится к распознаванию семантического шаблона (на примере таблиц состояний пользовательских примитивов), рассмотрены в разделе 2.5. Техника, достаточная для проверки каждого правила, указана в каталоге (приложение А). Обхода АСД достаточно для 65 правил, классификация типов логики процедурных блоков требуется для 22 правил, анализ путей выполнения — для пяти, анализ потоков данных — для семи, учет установленных разрядов и источников сигналов — для 21, анализ логической сети (раздел 2.6) и распознавание шаблона — для двух правил каждый. Часть правил требует нескольких техник совместно, два правила дополнительно используют сведения библиотек ячеек Liberty (глава 4).

2.3 Анализ на основе обхода абстрактного синтаксического дерева

Структурные правила проверяются обходом АСД, базовой техникой анализа, обладающей наименьшей вычислительной стоимостью, поскольку сложность обхода пропорциональна размеру дерева. Большая часть детекторов реализуется обходом АСД от корневого узла (единица компиляции) к листовым. Для каждого типа узла определяется обработчик (`handler`), вызываемый обходчиком при посещении узла соответствующего типа. Обработчик собирает необходимую информацию во время обхода и при обнаружении нарушения формирует

диагностическое предупреждение с его локализацией. Общий программный интерфейс детекторов, механизм их регистрации и формирование кода диагностики описаны в главе 5.

Рассмотрим технику на примере правила **PORT10**: «шины при создании экземпляра модуля не должны подключаться в обратном порядке разрядов». Использование обратных подключений (например, [15:0] к [0:15]) является синтаксически корректным, но плохой практикой, так как может приводить к ошибкам чтения неинициализированных разрядов. В листинге 2.2 приведен пример описания с конструкциями, как нарушающими данное правило, так и соответствующими ему.

Листинг 2.2: Иллюстрация различных аспектов правила PORT10

```

1 module test (input clk, input [0:5] in, inout [1:0] io);
2 endmodule
3
4 module top (input clk);
5     wire [0:5] sig1, sig2;
6     wire [5:0] sig3, sig4;
7     test t1 (clk, sig3, sig4[1:0]); // нарушение: порядок битов sig3 != in
8     test t2 (clk, sig1, sig2[0:1]); // нарушение: порядок битов sig2 != io
9     test t3 (clk, sig1, sig3[1:0]); // нет нарушения
10 endmodule

```

Для проверки правила достаточно обхода АСД с обработчиком узлов типа «экземпляр модуля». Обработчик последовательно перебирает все соединения портов экземпляра, отбирает соединения, тип которых является упакованным массивом, исключает случаи единичной разрядности и проверяет совпадение порядка следования разрядов: если у подключаемого сигнала левая граница диапазона меньше (больше) правой, а у соответствующего порта — наоборот, то порядки противоположны и выдается диагностическое предупреждение с указанием порта.

2.4 Анализ путей в процедурных блоках

Ряд правил требует анализа возможных путей выполнения внутри процедурного блока, для которого обхода АСД недостаточно: необходимо учитывать ветвления и отбрасывать недостижимые пути. Для проверки таких правил

в работе применяется техника анализа путей с интервальными ограничениями: операторы процедурного блока обходятся рекурсивно в порядке выполнения, сведения о пройденных операторах переносятся вдоль путей и объединяются в точках слияния ветвей, а интервальные ограничения на значения выражений в условиях ветвлений позволяют отбрасывать несовместимые сочетания ветвей (раздел 2.4.2). Дополнительно техника опирается на классификацию типов логики, реализуемой процедурными блоками (раздел 2.4.1). Рассмотрим технику на примере правила **ASSIGN04**: «в комбинационном блоке за неблокирующим присваиванием не должно следовать блокирующее». Блокирующее присваивание, следующее на одном пути комбинационного блока за неблокирующим, использует еще не обновленное значение, поэтому результат имитационного моделирования зависит от порядка операторов и расходится с синтезированной логической схемой [14].

2.4.1 Классификация типов логики в процедурных блоках

Для точного разграничения типов логики в процедурных блоках сформулированы критерии, учитывающие используемые в них операции. Введем ключевые понятия:

- *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.

Процедурные блоки *always* разделяются на четыре класса, проверяемых в следующем порядке. Блок без управления событиями относится к *тестовому окружению* (несинтезируемому коду). Блок относится к *последовательной* логике, если среди его входов есть событие по фронту сигнала (тактового сигнала или сброса). К логике *защелок* относится блок, у которого существует выход (с точностью до диапазона разрядов), обновляемый не на всех путях, в том числе из-за условия *iff* в списке чувствительности или цикла, число итераций которого не удастся установить (раздел 2.4.2). Блок, обновляющий все выходы

на каждом пути, считается *комбинационной* логикой, если он читает только входы и локальные переменные, использует неявный список чувствительности `@*` либо объявлен как `always_comb`. Остальные блоки, читающие сигналы, не входящие в список чувствительности, относятся к тестовому окружению. Блоки `initial` и `final` и блоки, порожденные утверждениями, не классифицируются. Задержки внутри блока критерием не являются: блок с задержками и без списка чувствительности попадает в тестовое окружение по отсутствию управления событиями. В строгом режиме неполнота списка чувствительности не учитывается, и блок, обновляющий все выходы на каждом пути, считается комбинационным. Примеры процедурных блоков каждого типа логики приведены в листинге 2.3.

Листинг 2.3: Примеры типов логики процедурных блоков

```

1 // комбинационная логика
2 always @(in, en) begin
3     if (en) out = in;
4     else    out = ~in;
5 end
6
7 // последовательностная логика
8 always @(posedge clk) begin
9     count <= count + 1;
10 end
11
12 // логика защелки
13 always @(in, en) begin
14     if (en) out = in;
15 end

```

2.4.2 Обход путей выполнения с интервальными ограничениями

Каждому выражению, используемому в условиях ветвлений, сопоставляется *интервальное ограничение* на его значения: упорядоченный список непересекающихся отрезков. Над таким представлением определены операции добавления отрезка (со слиянием перекрывающихся отрезков), вычитания, пересечения и объединения. Полное множество значений выражения определяется его типом. Ограничения выводятся из сравнений с константами, унарных операций и их конъюнкций и дизъюнкций над одним и тем же выражением.

Например, символу A типа `logic signed [5:0]` в условии $A \leq -3 \ || \ A > 9$ соответствует ограничение вида $[-32, -3] \cup [10, 31]$. Для ветви `else` берется дополнение ограничения до полного диапазона типа.

Операторы процедурного блока обходятся рекурсивно. Каждая ветвь условного оператора и каждая альтернатива оператора выбора обходится один раз со сведениями, накопленными до ветвления, а после оператора сведения ветвей объединяются. Пути при этом не перечисляются: вдоль них переносится проверяемое правило сведения (для правила `ASSIGN04` — неблокирующее присваивание, которое могло встретиться до текущей точки). Управляющие конструкции обрабатываются следующим образом:

- сведения ветвей оператора `case` объединяются по принципу «хотя бы в одной ветви», а обновление символа не во всех ветвях указывает на логику защелки (раздел 2.4.1);
- оператор `if` накладывает интервальное ограничение на выражение условия и дополняющее ограничение на ветвь `else`;
- циклы при проверке порядка присваиваний аппроксимируются двукратным обходом тела, что позволяет обнаруживать чередование видов присваивания между итерациями. Критерий гарантированного числа итераций (константные начальное значение и шаг счетчика цикла `for`, сравнение счетчика с вычислимым выражением в условии завершения, отсутствие оператора `continue`) применяется при классификации типов логики: цикл, не удовлетворяющий критерию, обходится однократно, а присваиваемые в его теле сигналы считаются обновленными не на всех путях. Циклы `while`, `do-while`, `repeat` и `foreach` критерию не удовлетворяют.

По схеме вычисления техника является прямым анализом потоков данных, а от анализа потоков данных раздела 2.7 отличается абстрактным доменом. Анализ потоков данных вычисляет для каждой точки блока диапазоны значений сигналов, тогда как анализ путей — сведения о видах и порядке выполненных операторов, а интервальные ограничения служат лишь для отбрасывания несовместимых сочетаний ветвей. Поэтому правила о порядке операторов проверяются анализом путей, а правила о значениях сигналов — анализом потоков данных.

Для процедурных блоков, реализующих комбинационную логику, правило `ASSIGN04` нарушается, если существует путь, на котором за неблокирующим присваиванием следует блокирующее. Состояние анализа $\sigma = (f, p, C)$ опи-

сывает неблокирующие присваивания, которые могли встретиться до текущей точки. Здесь f — первое неблокирующее присваивание текущей последовательности операторов, p — неблокирующее присваивание, встреченное на пути, который ограничениями не различается (под составным условием или в теле цикла), а отображение C сопоставляет выражению e условий предшествующих операторов if объединение $U[e]$ ограничений ветвей, на которых встречено неблокирующее присваивание, и первое такое присваивание $N[e]$. Блокирующее присваивание считается нарушением, если задано f или p либо отображение C непусто. Ветвь условного оператора по выражению e наследует присваивание $N[e]$ только при непустом пересечении ее ограничения с $U[e]$, а запись для e в ее состояние не переносится, поэтому взаимоисключающие ветви (например, по условиям $a == 0$ и $a == 1$) нарушения не дают. Остальные сведения передаются во вложенные операторы и возвращаются из них с объединением, так что неблокирующее присваивание под условием учитывается и для следующих операторов, в том числе для ветвей оператора if по другому выражению. Тело цикла обходится дважды с сохранением накопленных сведений, что позволяет обнаружить неблокирующее присваивание в конце итерации и блокирующее в начале следующей, а неблокирующее присваивание в теле учитывается и после цикла. Обход приведен в алгоритме 2.1: функция «Обход» возвращает состояние после оператора, запись $x \sqcup y$ обозначает x , если оно задано, и y иначе, а для отображений — объединение ограничений по каждому выражению, запись $C \setminus e$ обозначает отображение C без записи для выражения e . Программная реализация менеджера интервальных ограничений и обработчиков детектора описана в главе 5.

Алгоритм 2.1 — Обход путей процедурного блока с проверкой правила ASSIGN04

Вход: процедурный блок комбинационной логики (оператор) s

Выход: предупреждения о нарушениях правила ASSIGN04

1 **функция** Обход(s, σ):

 // $\sigma = (f, p, C)$, вначале f и p не заданы, $C = \emptyset$

2 **если** s — присваивание **то**

3 **если** s — неблокирующее и f не задано **то** $f \leftarrow s$;

4 **иначе если** s — блокирующее, и f или p задано либо $C \neq \emptyset$ **то** выдать
 предупреждение ASSIGN04(loc(s));

5 **иначе если** s — условный оператор с условием c и ветвями s_1, s_2 **то**

6 **если** c задает интервальное ограничение k на выражение e **то**

7 $k_1 \leftarrow k, k_2 \leftarrow \neg k$;

8 **для** $i \leftarrow 1$ **до** 2 :

9 **если** $U[e] \cap k_i \neq \emptyset$ **то** $g \leftarrow N[e]$ **иначе** $g \leftarrow f$;

10 $(f_i, p_i, C_i) \leftarrow \text{Обход}(s_i, (g, p, C \setminus e))$;

11 **для** $i \leftarrow 1$ **до** 2 :

12 $C \leftarrow C \sqcup C_i, p \leftarrow p \sqcup p_i$;

13 **если** f_i задано **то** $U[e] \leftarrow U[e] \cup k_i, N[e] \leftarrow N[e] \sqcup f_i$;

14 **если** $U[e]$ покрывает весь диапазон значений e **то** $f \leftarrow N[e]$;

15 **иначе**

16 **для** $i \leftarrow 1$ **до** 2 : $(f_i, p_i, C_i) \leftarrow \text{Обход}(s_i, \sigma)$;

17 $C \leftarrow C \sqcup C_1 \sqcup C_2, p \leftarrow p \sqcup p_1 \sqcup p_2 \sqcup f_2$;

18 **иначе если** s — оператор выбора с ветвями $s_1, \dots, s_m$ **то**

19 **для** $i \leftarrow 1$ **до** m : $(f_i, p_i, C_i) \leftarrow \text{Обход}(s_i, \sigma)$;

20 $f \leftarrow f_1 \sqcup \dots \sqcup f_m, p \leftarrow p \sqcup p_1 \sqcup \dots \sqcup p_m, C \leftarrow C \sqcup C_1 \sqcup \dots \sqcup C_m$;

21 **иначе если** s — цикл с телом s_b **то**

22 $(f_b, p_b, C_b) \leftarrow \text{Обход}(s_b, \text{Обход}(s_b, \sigma))$;

// две итерации

23 $C \leftarrow C \sqcup C_b, p \leftarrow p \sqcup p_b \sqcup f_b$;

24 **иначе если** s — блок операторов $s_1; \dots; s_n$ **то**

25 **для** $j \leftarrow 1$ **до** n : $\sigma \leftarrow \text{Обход}(s_j, \sigma)$;

26 **вернуть** σ ;

Каждая ветвь обходится однократно, а состояние содержит по одной записи на каждое выражение в условиях операторов if, поэтому время обхода составляет $O(n \cdot q)$, где n обозначает число операторов блока, а q обозначает число выражений в условиях. Исключение составляют циклы: тело цикла обходится дважды, и при вложенности циклов глубины d число обходов внутреннего тела растёт как 2^d .

2.5 Анализ синтезируемости пользовательских примитивов

Отдельную группу составляют правила проверки синтезируемости конструкций языка. Рассмотрим в качестве представительного примера распознавания семантического шаблона технику проверки правила **SYNTH11**: «пользовательские примитивы (User Defined Primitive, UDP) должны быть синтезируемыми». Язык Verilog позволяет описывать примитивы поведенчески, то есть таблицей состояний, строки которой задают соответствие между значениями входов (а для последовательностных примитивов также текущим состоянием) и значением выхода. Не всякая такая таблица соответствует синтезируемому элементу: синтезируемыми считаются комбинационные примитивы и последовательностные примитивы, реализующие защелку или D-триггер с однозначно заданной активной фазой тактового сигнала. Произвольная таблица состояний может не отображаться ни на один стандартный библиотечный элемент и потому не подлежит логическому синтезу.

Техника анализа основана на распознавании в таблице состояний UDP шаблона триггера или защелки. Анализатор обрабатывает таблицу следующим образом (алгоритм 2.2): для каждого входа примитива по строкам таблицы определяется, выступает ли он в роли тактового сигнала, и если да, то какого типа его активная фаза (передний фронт, задний фронт, высокий или низкий уровень). Распознавание тактового сигнала выполняется сопоставлением строк таблицы, описывающих реакцию выхода на изменение входа, с эталонными шаблонами переходов триггера (для пар входных комбинаций вида $0 \rightarrow 1$ и $1 \rightarrow 0$ при фиксированных значениях остальных входов и состояния). Для эффективного поиска строк таблицы используется префиксное дерево (trie) комбинаций «входы – состояние – выход». Если в таблице обнаруживается допустимый тактовый вход с единственной активной фазой и допустимым числом входов (от двух до пяти вместе с тактовым), примитив классифицируется как синтезируемый триггер или защелка. В противном случае выдается диагностическое предупреждение о несинтезируемом примитиве.

Алгоритм 2.2 — Распознавание тактового входа пользовательского примитива

Вход: таблица состояний UDP T со множеством входов I

Выход: классификация примитива: синтезируемый (комбинационный примитив, триггер или защелка) или несинтезируемый примитив

- 1 **если** примитив комбинационный **то вернуть** «синтезируемый комбинационный примитив»;
 - 2 построить префиксное дерево D строк таблицы T по ключу «входы — состояние — выход»;
 - 3 $c \leftarrow$ не задан; $p \leftarrow$ не задана ; // тактовый вход и его активная фаза
 - 4 **для каждого** входа $i \in I$:

- 5 **для каждого** фазы q из множества {передний фронт, задний фронт, высокий уровень, низкий уровень} :

- 6 **если** строки T , описывающие изменение входа i по фазе q , совпадают с шаблоном перехода триггера при всех фиксированных значениях прочих входов и состояния (поиск строк по дереву D) **то**
 - 7 **если** c не задан **то** $c \leftarrow i$; $p \leftarrow q$;
 - 8 **иначе вернуть** «несинтезируемый примитив» (несколько тактовых входов или фаз);
 - 9 **если** c задан, фаза p определена однозначно и $2 \leq |I| \leq 5$ (вместе с тактовым входом) **то вернуть** «синтезируемый триггер или защелка» (по фронту или уровню p) **иначе вернуть** «несинтезируемый примитив»;
-

Помимо распознавания тактового входа, детектор отвергает таблицы, в которых тактовый вход задан переходами вида ?0 или ?1, таблицы только из строк со значением x и слишком малые таблицы, а также примитивы, входы которых не удастся сопоставить сигналам данных, сброса, установки и разрешения (правило SYNTH11, приложение А). Для триггеров такое сопоставление выполняется только в строгом режиме детектора, который задается в конфигурации и по умолчанию выключен. Вычислительная стоимость распознавания линейна по числу строк таблицы состояний (с учетом числа входов примитива).

2.6 Анализ на основе логической сети

Правила, нарушение которых образуется замкнутым путем передачи значений между сигналами, не сводятся ни к обходу АСД, ни к анализу путей одного процедурного блока: путь может проходить через несколько блоков и непрерывных присваиваний. Рассмотрим технику на примере правила **DESIGN01**: «в описании не должно быть комбинационных петель». Комбинационная петля возникает, когда выход комбинационной логики через одну или несколько комбинационных конструкций возвращается на ее вход. Такая схема не имеет устойчивого состояния, а ее поведение зависит от задержек элементов.

Проверка выполняется на *логической сети* — ориентированном графе, строящемся обходом внутреннего представления. В отличие от сети логических элементов, используемой в главе 3, эта сеть строится без синтеза, на уровне объявлений и вхождений. Вершинами сети служат объявления портов и переменных, вхождения переменных в выражения и псевдонимы участков разрядов переменных, а ребра отражают передачу значения и помечены видом события присваивания. Чтобы путь через последовательностный элемент не принимался за петлю, в сеть включаются только процедурные блоки комбинационного вида (`always_comb` и `always @(*)`) и непрерывные присваивания. Для правила LATCN05 в сеть дополнительно включаются блоки, классифицированные как защелки (раздел 2.4.1). Ребра, помеченные фронтом тактового события, при поиске петель не учитываются. Если в описании используются разные участки разрядов одной переменной (например, `x[3:0]` и `x[7:4]`), каждому участку соответствует отдельная вершина сети. Это устраняет ложные петли в случаях, когда значение одного участка вычисляется через другой участок того же сигнала. Внутри комбинационного блока чтение переменной после ее присваивания на том же пути получает значение этого присваивания, поэтому ребро из внешнего значения не создается. Циклы `for` с константными границами раскручиваются в пределах заданного числа итераций, чтобы обращения по переменной цикла разрешались в конкретные разряды.

Поиск петель двухэтапный. Сначала алгоритмом Тарьяна [76] выделяются сильно связные компоненты сети, среди которых отбираются содержащие хотя бы одно ребро между вхождениями переменных. Затем только из вершин отобранных компонент запускается поиск в глубину, перечисляющий элементарные циклы.

Такой порядок снимает квадратичную зависимость от размера сети, поскольку поиск циклов не запускается из вершин, заведомо не лежащих на циклах. Техника приведена в алгоритме 2.3.

Алгоритм 2.3 — Обнаружение комбинационных петель на логической сети

Вход: развернутая модель M ; предел глубины петли L (по умолчанию не ограничен)

Выход: предупреждения о комбинационных петлях с перечислением звеньев

- 1 $N \leftarrow$ логическая сеть, построенная обходом комбинационных блоков и непрерывных присваиваний модели M с раскруткой циклов с константными границами;
 - 2 расщепить вершины объявлений N на псевдонимы по используемым диапазонам разрядов;
 - 3 $N' \leftarrow$ подграф N без ребер присваиваний по фронту;
 - 4 вычислить сильно связные компоненты N' алгоритмом Тарьяна;
 - 5 $S \leftarrow$ вершины компонент, содержащих ребро между двумя вхождениями переменных;
 - 6 **для каждой** вершины $u \in S$:
 - 7 **для каждого** элементарного цикла c через u , найденного поиском в глубину в пределах глубины L :
 - 8 **если** c содержит только чтение и запись одной переменной **то перейти к следующей итерации;**
 - 9 выдать предупреждение DESIGN01 с локализацией первого вхождения переменной в c и примечаниями на каждое звено c ;
-

Стоимость построения сети линейна по размеру развернутой модели с учетом раскрутки циклов, вычисление сильно связных компонент линейно по числу вершин и ребер сети, а перечисление циклов ограничено вершинами отобранных компонент и пределом глубины. Родственное правило LATCH05 (комбинационная обратная связь с выхода защелки на ее вход данных или сигнал разрешения) проверяется на той же сети, дополненной блоками защелок по классификации раздела 2.4.1 и ребрами «сигнал разрешения — выход».

2.7 Анализ потоков данных

Правила, требующие знания о возможных значениях сигналов и достижимости операторов (например, деление на ноль, выход за границы массива, недостижимый код), проверяются анализом потоков данных: распространением фактов о состоянии вычисления вдоль путей выполнения с объединением в точках слияния (стандартная схема описана в [15]).

В работе анализ потоков данных построен как надстройка над обходом управляющей структуры процедурного блока. Базовый уровень, предоставляемый компилятором, отслеживает присвоенные разряды сигналов и достижимость операторов. Разработанное в работе расширение добавляет к состоянию значения и диапазоны значений сигналов с указанием оператора, ограничившего диапазон (присваивания или условия). Множества разрядов и диапазоны значений представлены интервальными отображениями, что позволяет учитывать частичные присваивания многоразрядных сигналов. Альтернативные пути хранятся как множество отображений «сигнал — диапазоны значений», объединяемых в точках слияния. Анализ выполняется в границах одного процедурного блока с входом в вызываемые подпрограммы до заданной глубины вызовов и параметризуется константами, распространенными в экземпляре непрерывными присваиваниями. Циклы раскручиваются, пока условие завершения вычислимо, в пределах заданного числа шагов, а нераскрученный цикл обходится однократно как абстракция произвольного числа итераций. Исчерпание предела числа хранимых путей отмечается признаком усечения, который проверки универсальных свойств (недостижимый код) трактуют как отказ от вердикта, а проверки существования нарушения — как возможный пропуск. Явный граф потока управления или граф зависимостей по данным не строится: обход идет по дереву структурированных операторов блока.

Факты анализа (выражения с возможным нулевым делителем, операнды операций деления и возведения в степень, обращения по индексу с возможным выходом за границы, недостижимые операторы, подпрограммы с неограниченной рекурсией) вычисляются за один обход блока и разделяются между детекторами. Детектор объявляет требуемые факты и пределы анализа, а система объединяет заявки активированных детекторов, выполняет один обход на блок и кэширует результат по блоку, пределам и распространенным в экземпляре константам. По-

вторной обход эквивалентных тел экземпляров (с теми же значениями констант) отсекается. Перед анализом применяется недорогой по вычислениям предварительный фильтр, отбирающий только блоки, содержащие конструкции, о которых анализ может что-то сообщить (операцию деления, обращение по невычислимому индексу, условие, зависящее от присвоенного в блоке значения). На фактах анализа реализованы детекторы правил о делении на ноль (ARITH02), выходе за границы (RANGE01), недостижимом коде (DESIGN02), полноте присваивания результата функции (VARIABLE05) и комбинационной обратной связи через защелку (LATCH05), а на базовых результатах — детекторы правил о регистровых портах (PORT04) и каскадах защелок (LATCH04).

Рассмотрим технику на примере правила **RANGE01**: «обращения к массивам и битовым векторам не должны выходить за их границы». Компилятор сообщает лишь об индексах, вычисляемых статически. Детектор правила использует диапазоны значений, выведенные анализом потоков данных из присваиваний, условий, границ циклов и аргументов вызовов подпрограмм, и поэтому выявляет обращения, индекс которых может выйти за границы хотя бы на одном пути. Для объекта фиксированного размера границы определяются типом, для динамического массива, очереди и строки — отслеживаемым по пути размером объекта, а при неизвестном размере проверяется только отрицательность индекса. Обращение с индексом, вычисляемым статически, на объекте фиксированного размера не рассматривается, поскольку о нем сообщает компилятор. Вердикт выносится по первому пути, на котором диапазон индекса выходит за границы, и сопровождается примечаниями о том, какие значения анализ вывел для индекса и откуда они получены. Если анализ не сумел сузить диапазон индекса по сравнению с диапазоном его типа, нарушение по умолчанию не сообщается, чтобы не порождать срабатывания на каждом неограниченном индексе. Вычислительная стоимость анализа определяется числом хранимых путей, которое в худшем случае растет экспоненциально с числом ветвлений блока, и ограничена заданными пределами. Техника приведена в алгоритме 2.4.

Алгоритм 2.4 — Проверка выхода за границы по диапазонам значений индексов (правило RANGE01)

Вход: процедурный блок b ; константы K , распространенные в экземпляр непрерывными присваиваниями; пределы анализа (число шагов раскрутки, число путей, глубина вызовов)

Выход: предупреждения о возможном выходе за границы

- 1 **если** b не содержит обращения по невычислимому индексу **то вернуть**;
 - 2 $\Pi \leftarrow$ множество путей блока b с отображениями «сигнал — диапазоны значений», полученное обходом управляющей структуры b из начального состояния K с раскруткой циклов и входом в подпрограммы в пределах заданных пределов;
 - 3 **если** обход прерван по достижению предела числа шагов **то** пометить результат как усеченный;
 - 4 **для каждого** обращения по индексу $e = a[i]$ или $a[l : r]$ в b и в вызванных из него подпрограммах :
 - 5 $V \leftarrow$ границы объекта a (по типу либо по отслеженному размеру);
 - 6 **если** V не определены, либо индекс вычислим статически и объект имеет фиксированный размер **то перейти к следующей итерации**;
 - 7 **для каждого** пути $\pi \in \Pi$:
 - 8 $I \leftarrow$ диапазоны значений индекса (индексов) на пути π ;
 - 9 **если** $I \not\subseteq V$ и (I сужен относительно диапазона типа индекса или включен режим предупреждений о неограниченных индексах) **то**
 - 10 выдать предупреждение RANGE01 с локализацией e , частью I вне V , значениями I и их происхождением на пути π ;
 - 11 **break** ; // один вердикт на обращение
-

2.7.1 Учет установленных разрядов и источников сигналов

Отдельно от анализа потоков данных в границах блока ведется учет установленных разрядов сигналов в пределах экземпляра модуля, необходимый правилам об инициализации сигналов. Специфику RTL-описаний здесь составляют четы-

рехзначная логика сигналов (значения 0, 1, X и Z) и частичная инициализация многоразрядных сигналов: разряд без источников не имеет определенного логического значения. У переменной это значение X, у цепи — значение Z (IEEE 1800, раздел 6.6), которое при чтении в выражения также дает X. Проверка правила **SIGNAL07** («используемый сигнал должен быть предварительно установлен») выполняется потоко-нечувствительно: за первый проход по телу экземпляра для каждого сигнала накапливается маска установленных разрядов по всем источникам, за второй проход выявляются чтения разрядов, не вошедших в маску (алгоритм 2.5). Сигналы, разрядность которых превышает заданный предел, считаются установленными целиком.

Алгоритм 2.5 — Учет установленных разрядов и обнаружение неинициализированных сигналов (правило SIGNAL07)

Вход: тело экземпляра модуля M

Выход: предупреждения о чтении неустановленных разрядов
(со значением X или Z)

- 1 **для каждого** сигнала v в M : $Driven[v] \leftarrow \emptyset$;
 // первый проход: накопление установленных разрядов
 - 2 **для каждого** источника d в M (присваивания, инициализаторы, входные порты, подключения выходных портов экземпляров, выходные аргументы вызовов) :
 - 3 $v \leftarrow$ приемник источника d ;
 - 4 $S \leftarrow$ срез разрядов сигнала v , устанавливаемый источником d ;
 - 5 $Driven[v] \leftarrow Driven[v] \cup S$; // объединение интервалов разрядов
 // второй проход: обнаружение чтения неустановленных разрядов
 - 6 **для каждого** чтения сигнала v на срезе разрядов R :
 - 7 $U \leftarrow R \setminus Driven[v]$; // разряды без единого источника
 - 8 **если** $U \neq \emptyset$ **то** выдать предупреждение о неинициализированных разрядах U с локализацией чтения;
-

Стоимость проверки линейна по числу источников и чтений сигналов в теле экземпляра. Для правил, требующих сведений об источниках сигнала по всей иерархии (неподключенные порты и порты без источника, несколько источников у сигнала без третьего состояния, тактовые сигналы в последовательностной логике), используется агрегирование источников сигналов средствами компилятора:

для каждого сигнала накапливается интервальная карта источников из процедурных блоков, непрерывных присваиваний, подключений портов и тактовых переменных по всей развернутой модели, что исключает повторный обход иерархии каждым детектором. Для правил о константных тактовых сигналах, сигналах сброса и разрешения, о подключении портов к сетям питания и о принудительном изменении направления портов по развернутой модели строятся цепочки соединений сигналов через порты экземпляров, по которым прослеживаются источники и приемники сигнала. Для правил о третьем состоянии и неопределенных значениях сигналов признаки возможных значений Z и X, заданные литералами, потоко-нечувствительно распространяются по присваиваниям в пределах развернутой модели.

2.8 Предметно-ориентированный язык сопоставления узлов абстрактного синтаксического дерева

Базовой техникой проверки структурных правил в разработанном методе является обход АСД (раздел 2.3): для каждого детектора вручную реализуется обходчик с обработчиками интересующих типов узлов. С ростом набора правил такой способ разработки детекторов становится узким местом: он требует глубокого знания внутреннего устройства компилятора и порождает объемный однотипный код. В настоящем разделе представлен разработанный в работе предметно-ориентированный язык сопоставления узлов АСД языка SystemVerilog, то есть механизм *сопоставителей* (matchers), выполненный по образцу сопоставителей АСД (AST matchers) инструмента clang-tidy [77] и встроенный в модуль анализа системы.

2.8.1 Мотивация и обзор существующих решений

Основная задача обходчика АСД — вернуть узел дерева, соответствующий заданным условиям. Практика разработки детекторов выявила следующие недостатки этого механизма.

- Задачи, выходящие за рамки базовой (сбор всех узлов заданного вида, отбор узлов по условиям на их поддеревья), не реализуются штатными средствами. Для них приходится писать объемный и сложный в сопровождении код, который затруднительно использовать повторно.
- Объем кода детектора на обходчиках определяется не сложностью правила, а устройством интерфейса обхода. Так, детектор правила PORT10 (подключение шин в обратном порядке разрядов) на обходчиках требует отдельного класса-обходчика, обработчик которого занимает 33 строки (листинг 5.4), тогда как тот же детектор на сопоставителях записывается одним выражением и занимает 16 строк без дополнительных классов (листинг 5.8), не считая общей функции сравнения направлений диапазонов разрядов.
- Обход всегда начинается с корневого узла и не предоставляет способа пропустить заведомо нерелевантные поддеревья, хотя зачастую поиск эффективнее начинать с узлов определенного типа (например, с определенных модулей).
- Узлы обрабатываются строго по одному. Получить список узлов, удовлетворяющих (или не удовлетворяющих) условию, штатным обходчиком нельзя, поэтому на каждый такой запрос требуется отдельный класс.

В области статического анализа программного обеспечения аналогичная задача решена языком сопоставителей АСД инструмента clang-tidy [77], встроенным в C++ языком вложенных вызовов, сочетающим сопоставители конкретных типов узлов, атрибутные сопоставители и сопоставители, управляющие обходом, а также агрегирующие конструкции `allOf` (истина, если выполнены все вложенные проверки), `anyOf` (истина, если выполнена хотя бы одна) и `unless` (инвертирует результат вложенной проверки). Дополнительно язык clang-tidy предоставляет *механизм захвата узлов* (метод `bind`), позволяющий в рамках одного прохода поиска пометить и получить сразу несколько различных узлов, сокращая тем самым число обходов дерева.

Для языка SystemVerilog аналогичный механизм предоставляет инструмент Verible [68]: в нем предусмотрены сопоставители для поиска шаблонов в дереве разбора, построенные по образцу сопоставителей clang-tidy. Однако Verible работает на уровне дерева разбора, сохраняющего исходное разбиение кода на лексемы до обработки препроцессором, обладает лишь частичной поддержкой стандарта языка и не строит развернутой модели. Поскольку ис-

пользуемый в работе компилятор slang [62] обеспечивает полную поддержку стандарта и построение развернутой модели (раздел 2.1), механизм сопоставителей был разработан именно для его АСД.

К разрабатываемому языку были предъявлены следующие требования: полнота относительно типов узлов АСД компилятора (сопоставитель для каждого типа узла и для каждого отношения между узлами), оптимизация производительности анализа за счет кэширования и унифицированного обхода АСД, совместимость с существующим интерфейсом детекторов (раздел 5.1.7) для упрощения интеграции.

2.8.2 Синтаксис языка сопоставителей

Сопоставители записываются как вложенные вызовы функций, образующие дерево условий. Подмножество формальной грамматики языка в нотации Бэкуса — Наура приведено в листинге 2.4.

Листинг 2.4: Грамматика языка сопоставителей (подмножество)

```

1 <matcher>      ::= <node-matcher> | <attribute-matcher>
2                | <traversal-matcher>
3 <node-matcher> ::= <node-type> [ "(" <matcher-args> ")" ]
4                [ ".bind(" <string-literal> ")" ]
5 <node-type>    ::= <concrete-type> | <base-type>
6 <concrete-type> ::= "PortSymbol" | "SequenceSymbol" | "SubroutineSymbol"
7                | "BinaryExpression" | "IntegerLiteral"
8                | другие< конкретные типы узлов АСД>
9 <base-type>     ::= "Symbol" | "Statement" | "Expression" | "Constraint"
10               | "AssertionExpr" | "BinsSelectExpr" | "TimingControl"
11               | "Pattern"
12 <matcher-args> ::= <matcher> | <matcher-list> | <aggregation-matcher>
13               | <predicate>
14 <matcher-list> ::= <matcher> "," <matcher-list> | <matcher>
15 <predicate>    ::= пользовательская< проверка узла>
16 <attribute-matcher> ::= <attribute-name> [ "(" [ <matcher> ] ")" ]
17 <attribute-name> ::= "hasRHS" | "hasLHS" | "hasCondition" | "hasBody"
18               | "hasConnectedPort" | другие< атрибуты>
19 <traversal-matcher> ::= <traversal-name> "(" <matcher> ")"
20 <traversal-name>  ::= "has" | "hasDescendant" | "hasParent"
21               | "hasAncestor"
22 <aggregation-matcher> ::= "allof(" <matcher-list> ")"

```

```

23 |                                     | "anyOf(" <matcher-list> ")"
24 |                                     | "unless(" <matcher> ")"
25 | <string-literal> ::= "'" <identifier> "'"

```

Язык различает три типа сопоставителей.

- *Сопоставители типов узлов* (конкретных типов, например `PortSymbol` и `BinaryExpression`, а также базовых категорий, описанных ниже) — единственный тип, который может появляться на верхнем уровне проверки. Все остальные сопоставители могут быть лишь вложены в него или друг в друга, причем количество вложенных сопоставителей и глубина вложенности не ограничены. Это же единственный тип, поддерживающий механизм захвата узлов (метод `bind`).
- *Атрибутные сопоставители* (например, `hasRHS`, `hasLHS`, `hasCondition`, `hasBody`, `hasConnectedPort`) проверяют атрибуты узла: операнды, условие и тело цикла, связанный с выражением-подключением порт, типы данных и т. п. Они всегда вложены в другой сопоставитель и возвращают логическое значение, определяющее продолжение или остановку обхода вглубь поддерева.
- *Сопоставители, управляющие обходом* (`has`, `hasDescendant`, `hasParent`, `hasAncestor`) выражают отношения «предок — потомок» и запускают вторичный поиск от текущего узла, в том числе вверх по дереву: `has` проверяет непосредственных потомков, `hasDescendant` проверяет потомков на любой глубине, а `hasParent` и `hasAncestor` проверяют соответственно непосредственного предка и предков на любой глубине.

Сопоставители конкретных типов узлов реализованы для всех типов узлов АСД используемого компилятора. Дополнительно предусмотрены *сопоставители базовых категорий* (`Symbol`, `Statement`, `Expression`, `Constraint`, `AssertionExpr`, `BinsSelectExpr`, `TimingControl`, `Pattern`), сопоставляющиеся с любым узлом соответствующей категории независимо от его конкретного типа. Это удобно для сквозных проверок, охватывающих целый класс конструкций (типы данных при этом представлены как частный вид символов и отдельной категории не образуют). Агрегирующие конструкции `allOf`, `anyOf` и `unless` комбинируют условия логическими операциями. При этом перечисление нескольких вложенных сопоставителей внутри одного узлового сопоставителя эквивалентно их объединению конструкцией `allOf` (все условия должны выполняться).

Наряду с перечисленными сопоставителями аргументом узлового сопоставителя может выступать *предикат* — произвольная пользовательская проверка узла, возвращающая логическое значение. Предикат применяется к узлу наравне с остальными вложенными условиями и служит точкой расширения языка: свойства, не выразимые имеющимися атрибутивными сопоставителями (например, сравнение направлений диапазонов разрядов у порта и подключенной к нему шины), задаются предикатом, тогда как структурный отбор узлов остается декларативным. Это позволяет не расширять язык новым атрибутивным сопоставителем ради единичного правила. Часть типовых предикатов предоставляется языком в готовом параметризуемом виде, например проверка вида операции бинарного или унарного выражения (`hasAnyOperator`).

Сопоставление выражения-сопоставителя с деревом иллюстрируют примеры в листингах 2.5–2.7. Выражение с атрибутивным сопоставителем (листинг 2.5) находит все бинарные выражения, правым операндом которых является целочисленный литерал. Выражение с сопоставителем, управляющим обходом (листинг 2.6), находит все целочисленные литералы, имеющие среди предков бинарное выражение.

Листинг 2.5: Атрибутивный сопоставитель: бинарные выражения с целочисленным литералом в правом операнде

```

1 | BinaryExpression(
2 |     hasRHS(
3 |         IntegerLiteral()
4 |     )
5 | ).bind("bin-exp")

```

Листинг 2.6: Сопоставитель со вторичным обходом: литералы внутри бинарных выражений

```

1 | IntegerLiteral(
2 |     hasAncestor(
3 |         BinaryExpression()
4 |     )
5 | ).bind("int-lit")

```

Строковый литерал в `bind` задает уникальный в рамках правила идентификатор захваченных узлов. Результаты доступны детектору в двух видах. Объединенный вид — ассоциативный контейнер, отображающий идентификатор в вектор всех узлов АСД, удовлетворивших условиям. Поэлементный вид сохраняет для каждого отдельного совпадения верхнеуровневого сопоставителя

собственный набор привязок. Первый вид удобен, когда достаточно перечня узлов одного идентификатора. Второй необходим, когда важно *соответствие* между узлами, захваченными разными идентификаторами в рамках одного совпадения (например, пара «выражение-подключение, порт»), поскольку в объединенном виде эта связь теряется. Захватывать можно и промежуточные узлы: в выражении из листинга 2.7 узлы типа `BinaryExpression` будут помечены только в том случае, если удовлетворено требование на узел типа `IntegerLiteral`: пометка промежуточных узлов откладывается до успеха внешнего сопоставителя.

Листинг 2.7: Захват промежуточных узлов

```

1 IntegerLiteral(
2     hasAncestor(
3         BinaryExpression().bind("bin-exp")
4     )
5 ).bind("int-lit")

```

2.8.3 Семантика сопоставления и организация анализа

Список включенных проверок известен при запуске анализатора, поэтому все сопоставители предварительно распределяются по типам узлов верхнего уровня сопоставления. Узлы каждого типа собираются в непрерывный контейнер и просматриваются каждым сопоставителем последовательно, что положительно сказывается на использовании аппаратного кэша данных и вычислительной стоимости работы детекторов. Сопоставители разных верхнеуровневых типов вызываются независимо, поэтому для описаний, в которых те или иные синтаксические конструкции не представлены вовсе, соответствующие проверки не вносят накладных расходов.

Семантика перехода к другим узлам различается по типам сопоставителей. Сопоставитель конкретного типа узла сам обхода не выполняет: он проверяет предъявленный ему узел на конъюнкцию вложенных условий. Узлы верхнего уровня поставляет легковесный кэш, а вглубь дерева анализ направляют вложенные сопоставители. Атрибутный сопоставитель переходит ровно к одному узлу, определяемому атрибутом (например, после `hasRHS` анализ продолжается только по правому операнду), и применяет к нему не более одного вложенного

сопоставителя. Сопоставители, управляющие обходом, запускают вторичный поиск и выражают *существование* узла с требуемыми свойствами: `has` перебирает непосредственных потомков, `hasDescendant` перебирает потомков рекурсивно, а `hasParent` и `hasAncestor` проверяют непосредственного предка и предков вверх до корня. Поиск завершается на первом подходящем узле, который и становится захваченным узлом.

Механизм захвата согласован с логикой ветвления: привязки, сделанные в неуспешных ветвях (не подошедший потомок при `has`, отвергнутая альтернатива при `anyOf`), в результат не попадают, а привязки, сделанные внутри `unless`, отбрасываются всегда, поскольку вложенный сопоставитель там либо не сработал, либо его срабатывание означает неуспех внешнего условия.

Анализ механизмом сопоставителей выполняется в два этапа (алгоритм 2.6). На первом этапе выполняется единственный полный обход дерева, заполняющий два кэша: *легковесный кэш* (отображение типа узла во множество всех узлов данного типа, заполняемое только для типов, на которых зарегистрированы сопоставители включенных проверок, с доступом к узлам типа за $O(1)$ и обходом всех узлов типа за $O(k)$, где k — число узлов данного типа) и *граф узлов*, то есть линейное представление дерева в порядке прямого обхода, хранящее для каждого узла индекс родителя и размер поддеревя. Поддерево узла занимает в таком представлении непрерывный диапазон, поэтому поиск потомка на любой глубине сводится к линейному просмотру диапазона, перебор непосредственных потомков выполняется переходами через поддеревья, а подъем к предкам — переходами по индексам родителей. Граф узлов строится *по требованию*: потребность в нем выводится автоматически из состава зарегистрированных сопоставителей (граф используют сопоставители, управляющие обходом, и атрибутивный сопоставитель `hasConnectedPort`), а детектор, не обращающийся к отношениям «предок — потомок», может явно объявить отказ от графа. Если так поступают все детекторы набора, построение графа пропускается, и подготовка сводится к заполнению легковесного кэша. Тела модулей, имеющих несколько экземпляров, обходятся однократно, через канонический экземпляр, согласованно с устранением повторов, выполняемым компилятором при построении развернутой модели (выражения-подключения портов, принадлежащие конкретному экземпляру, обходятся для каждого экземпляра). Код, не имеющий экземпляров в развернутой модели, из обхода исключается, поэтому сопоставителям предъявляются те же узлы, что и обходчикам.

Алгоритм 2.6 — Сопоставление набора правил с АСД

Вход: АСД развернутой модели; набор правил с зарегистрированными сопоставителями

Выход: для каждого правила — захваченные узлы по идентификаторам `bind`

// этап 1: единственный полный обход дерева

- 1 определить по составу сопоставителей, требуется ли граф узлов;
- 2 обойти АСД, пропуская код без экземпляров и повторные тела экземпляров; заполнить легковесный кэш: для каждого затребованного типа сохранить узлы этого типа; если граф узлов требуется — построить его: порядок прямого обхода, индексы родителей, размеры поддеревьев;

// этап 2: обработка правил по кэшам

- 3 **для каждого** правила r и **каждого его** *верхнеуровневого сопоставителя* m с *типом узла* T :
 - 4 **для каждого** узла n *типа* T (из легковесного кэша) :
 - 5 применить вложенные сопоставители m к узлу n : атрибутные — с продолжением анализа только по узлу-атрибуту; управляющие обходом — со вторичным поиском по графу узлов (просмотр диапазона поддерева или подъем по индексам родителей);
 - 6 **если** узел n *удовлетворяет всем условиям* **то** зафиксировать привязки `bind` для n и помеченных промежуточных узлов;
-

Единообразие обработки разнородных узлов обеспечивается обобщенным типонезависимым дескриптором узла. При заполнении легковесного кэша каждый узел дополнительно индексируется в группе своей базовой категории, что позволяет сопоставителям базовых категорий получать все узлы класса за то же время $O(1)$, что и сопоставителям конкретных типов. Связи «родитель — потомок», отсутствующие в структуре АСД компилятора, восстанавливаются в ходе того же единственного обхода при помощи стека открытых предков и сохраняются в графе узлов.

Результаты каждой проверки изолированы: для каждого правила выделяется отдельная ячейка внутреннего ассоциативного контейнера, поэтому работа различных проверок не влияет на результаты друг друга. Стоимость заполнения кэшей при запуске набора из N проверок амортизируется: N полных обходов дерева заменяются одним, а на каждую проверку приходится лишь просмотр

узлов ее верхнеуровневого типа. Благодаря построению графа узлов по требованию и однократному обходу тел модулей с несколькими экземплярами выигрыш по сравнению с обходчиками достигается уже при одной включенной проверке и растет при совместном запуске набора. Экспериментальная оценка приведена в главе 6 (раздел 6.5).

2.9 Расширяемость и поддержка стандартов

Расширяемость набора проверок следует из определений раздела 2.1.2: добавление правила сводится к реализации его детектора одной из техник разделов 2.3–2.7.

Детектор реализуется как отдельная единица, объявляющая проверяемое свойство, текст и уровень значимости диагностики, и регистрируется в реестре детекторов с указанием категории. Категории соответствуют аспектам описания (раздел 2.2) и служат единицей группового включения и отключения проверок.

Ресурсоемкие предварительные вычисления не дублируются детекторами. Детектор декларативно объявляет, какие общие данные ему требуются: классификация типов логики (раздел 2.4.1), агрегированные по иерархии источники сигналов, четырехзначная модель значений сигналов, константы непрерывных присваиваний, цепочки соединений сигналов через порты экземпляров и карта использований сигналов (раздел 2.7.1), сопоставители узлов АСД (раздел 2.8), факты анализа потоков данных с требуемыми пределами (раздел 2.7). Система объединяет заявки активированных детекторов, строит каждый анализ один раз на запуск и разделяет результаты между детекторами, а детекторы, не объявившие потребность, затрат на построение не несут.

Каждое правило настраивается независимо. Детектор может быть включен или отключен в конфигурации анализа (подмножество D' в разделе 2.1.2). Ему сопоставляется уровень значимости диагностики (от игнорирования до неустранимой ошибки), а поведение уточняется параметрами и исключениями, при которых нарушение не сообщается. Настройки делятся на общие для всех детекторов (имена тактовых сигналов, сбросов и сигналов разрешения, активный уровень сброса, соглашения об именовании портов, имена шин питания, списки файлов, исключаемых из анализа или относимых к тестовому окружению, строгий

режим) и индивидуальные (пределы раскрутки циклов, числа путей и глубины вызовов, пороги, исключение отдельных конструкций, например операторов case с полным покрытием значений селектора или экземпляров, помеченных как библиотечные ячейки). В исходном тексте поддерживается локальное подавление диагностик директивами и комментариями. Настраиваемость позволяет адаптировать набор проверок к соглашениям, принятым разработчиками, и управлять долей ложных срабатываний.

Полнота поддержки языка обеспечивается использованием полнофункционального компилятора SystemVerilog (см. главу 5), выполняющего разбор и построение развернутой модели для конструкций стандартов SystemVerilog IEEE 1800-2017 и IEEE 1800-2023 в полном объеме. Это обеспечивает обработку произвольных корректных описаний на SystemVerilog, а не только синтезируемого подмножества, и расширяет применимость метода по сравнению с анализаторами, ограниченными синтезируемым подмножеством.

2.10 Выводы по главе

Введены понятия внутреннего представления, правила, нарушения и детектора, обеспечивающие расширяемость набора проверок без модификации ядра анализа. Разработан и классифицирован по 19 категориям набор из 118 правил статического анализа (114 правил для языков Verilog и SystemVerilog, 2 правила корректности модели подсистемы питания и 2 правила корректности пересечения доменов синхронизации и сброса), направленных на выявление структурных и семантических дефектов и нарушений свойств безопасности. Правилам сопоставлены техники анализа, достаточные для их проверки (приложение А). Разработан метод статического анализа, основанный на внутреннем представлении и обеспечивающий проверку правил без выполнения логического синтеза и имитационного моделирования. Метод объединяет техники, различающиеся вычислительной стоимостью и множеством выявляемых свойств. Это обход АСД для структурных правил, классификация типов логики и анализ путей в процедурных блоках с интервальными ограничениями для правил о типе логики и порядке операторов, распознавание семантических шаблонов на примере анализа синтезируемости пользовательских примитивов, анализ на основе логической

сети для правил о замкнутых путях передачи значений и анализ потоков данных, расширенный отслеживанием диапазонов значений сигналов, с фактами, вычисляемыми однократно и разделяемыми между детекторами, для правил, требующих знания о значениях сигналов и достижимости операторов. Отдельно ведется учет установленных разрядов и источников сигналов для правил об инициализации, источниках и третьем состоянии сигналов. Для сокращения объема и унификации кода детекторов структурных правил предложен предметно-ориентированный язык сопоставления узлов АСД с механизмом захвата узлов и кэшированием обхода, развивающий расширяемость набора проверок. Результаты главы соответствуют первому положению, выносимому на защиту.

Глава 3. Метод извлечения и статического анализа конечных автоматов из RTL-описаний

Конечные автоматы реализуют управляющую логику RTL-описаний, а их дефекты (мертвые переходы, недостижимые состояния, тупиковые состояния и взаимоблокировки) не выявляются синтаксическими проверками (раздел 1.3.5) и трудно обнаруживаются методами имитационной верификации. В настоящей главе вводятся формальные определения, обосновывается выбор подхода к извлечению конечных автоматов на основе сети логических элементов (netlist) и представлен разработанный в работе метод, включающий извлечение автоматов из RTL-описаний с сохранением соответствия исходному коду, их графовый анализ и поиск взаимодействующих автоматов.

3.1 Основные определения

Множество состояний конечного автомата и переходы между ними образуют ориентированный *граф автомата*, вершинами которого являются состояния, а ребрами — переходы. На этом графе определяются используемые далее понятия достижимости и связности (построение графа по извлеченному автомату описано в разделе 3.4).

- *Конечный автомат* (Finite State Machine, FSM) — модель последовательностной логики, описывающая поведение системы через конечное множество дискретных *состояний*, *переходов* между ними и соответствующих *выходов*. В любой момент времени автомат находится ровно в одном из возможных состояний. Следующее состояние определяется *функцией переходов*, зависящей от текущего состояния и входов. В RTL-коде состояние хранится в выделенном регистре, состояния кодируются именованными константами, а входы автомата соответствуют условиям переходов, встречающимся в операторах `if`, `case` и тернарных выражениях.
- *Начальное состояние* — состояние, занимаемое автоматом в начальный момент работы. В практических RTL-описаниях этот момент соответ-

ствует активации сигнала сброса, переводящего все регистры в значения по умолчанию.

- *Мертвый переход* — переход, который никогда не происходит (например, вследствие невыполнимости условий).
- *Недостижимое состояние* — состояние, которое никогда не достигается в процессе выполнения.
- *Сильно связанная компонента* (Strongly Connected Component, SCC) — максимальный подграф, в котором каждая вершина достижима из любой другой.
- *Терминальная сильно связанная компонента* (Terminal SCC, TSCC) — SCC, не содержащая начального состояния автомата и не имеющая исходящих ребер к вершинам вне компоненты.
- *Тупиковое состояние* (deadlock) — состояние, из которого отсутствуют переходы в другие состояния (формально: терминальная сильно связанная компонента, содержащая единственное состояние).
- *Взаимоблокировка* (livelock) — группа состояний, которые могут переходить друг в друга, но не в состояния вне этой группы (формально: терминальная сильно связанная компонента, содержащая несколько состояний). Термин «взаимоблокировка» употребляется в работе именно в этом смысле: автомат не останавливается в одном состоянии, но и не может покинуть группу состояний (в англоязычной литературе такое поведение называют livelock).

3.2 Обзор подходов к извлечению и обоснование выбора

Существующие подходы к извлечению конечных автоматов из RTL-описаний условно делятся на основанные на абстрактном синтаксическом дереве и основанные на сети логических элементов [39].

Подход на основе АСД обнаруживает автомат, выявляя процедурный блок с оператором case, извлекая состояния из его меток и восстанавливая переходы из внутренней логики. Преимуществом является прямое соответствие результатов исходному коду. Однако подход сталкивается с практическими трудностями: не все автоматы следуют каноническому шаблону на основе case, логика пере-

ходов может быть распределена по нескольким процедурным блокам, а каждый альтернативный стиль кодирования требует отдельной эвристики обнаружения, при этом параметры учитываются лишь на уровне модуля, без значений конкретного экземпляра.

Подход на основе сети логических элементов выполняет обнаружение и извлечение на уровне сети логических элементов, полученной после построения развернутой модели и трансляции. Сеть логических элементов унифицирует структурно схожие конструкции (if и case) и разрешает экземпляры параметризованных модулей, что упрощает идентификацию автоматов и обеспечивает обработку распределенной логики. Недостатками являются возможная потеря информации об исходном коде и влияние оптимизаций синтеза. Параметры модуля могут входить в условия переходов наряду с сигналами, поэтому раскрытия параметров вне процедурной логики недостаточно: извлечение по сети логических элементов выполняется для каждого экземпляра с его значениями параметров (раздел 3.7).

Сопоставление существующих инструментов извлечения и анализа конечных автоматов по ключевым возможностям приведено в таблице 3. Ни один из них не выявляет мертвые переходы и взаимодействующие автоматы.

Таблица 3 — Сопоставление инструментов анализа конечных автоматов

	Wang	Yosys	AVFSM	FSMx	Shi	Moon	svan-fsm
Уровень входа	АСД	сеть	вент.	вент.	вент.	вент.	сеть
Покрытие обнаружения	низк.	сред.	выс.	выс.	выс.	выс.	выс.
Соответствие коду	прям.	нет	нет	нет	нет	н/д	прям.
Тупиковые состояния и взаимоблокировки	нет	нет	нет	нет	нет	да	да
Недостижимые состояния	да	нет	нет	да	нет	нет	да
Мертвые переходы	нет	нет	нет	нет	нет	нет	да
Анализ безопасности	нет	нет	да	да	нет	нет	нет
Взаимодействующие автоматы	нет	нет	нет	нет	нет	нет	да

Источники: Wang [78], Yosys [70], AVFSM [79], FSMx [42; 43], Shi [80], Moon [81]. Обозначения уровня входа: «АСД» — абстрактное синтаксическое дерево, «сеть» — сеть логических элементов RTL, «вент.» — список соединений вентильного уровня. Покрытие обнаружения — доля распознаваемых инструментом стилей кодирования автоматов.

В настоящей работе выбран подход на основе сети логических элементов как обеспечивающий независимость от стиля кодирования и естественную обработку распределенной и параметризованной логики. Для устранения недостатка

(потери соответствия исходному коду) разработан механизм сопоставления сети логических элементов с узлами АСД (см. раздел 3.5).

3.3 Метод извлечения конечных автоматов из сети логических элементов

Предлагаемый метод состоит из пяти этапов (рисунок 3.1):

1. построение АСД и развернутой модели по исходному коду;
2. трансляция развернутой модели в промежуточное представление уровня регистровых передач RTLIL (RTL Intermediate Language) инфраструктуры логического синтеза Yosys [70];
3. обнаружение и извлечение автоматов из RTLIL в отдельные ячейки с информацией о состояниях, переходах, входах и выходах;
4. построение и анализ ориентированного графа каждого автомата;
5. сопоставление результатов с исходным кодом.

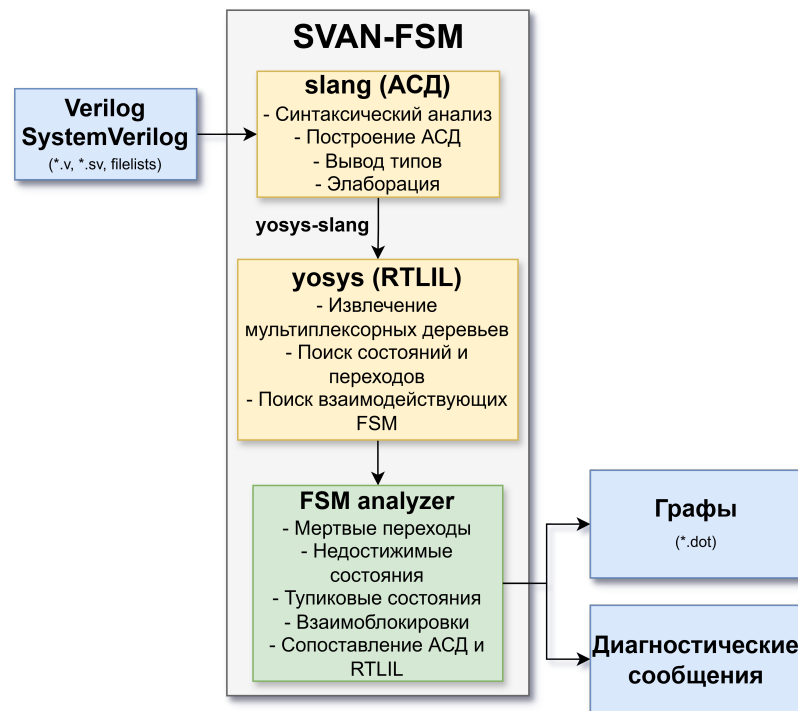


Рисунок 3.1 — Конвейер извлечения и статического анализа конечных автоматов

Обнаружение автоматов основано на поиске триггеров (D-триггеров) и анализе дерева мультиплексоров, управляющего входом данных триггера: если на входах мультиплексоров присутствуют только константы и выход того же

триггера, пара рассматривается как кандидат на конечный автомат. Извлечение преобразует триггер и его дерево мультиплексоров в выделенную ячейку, инкапсулирующую логику автомата: входы и возможные состояния идентифицируются обходом дерева мультиплексоров, а функции переходов и выходов определяются вычислением схемы для каждого возможного значения состояния.

В рамках работы штатная схема обнаружения и извлечения автоматов инфраструктуры Yosys (проходы `fsm_detect` и `fsm_extract`) была расширена и доработана для повышения точности анализа и покрытия.

Частичное разделение проводов (wire slicing). Триггеры и провода не всегда находятся во взаимно однозначном соответствии: при использовании массива, каждый элемент которого содержит логику автомата, создается отдельный триггер для каждого элемента, тогда как все элементы могут использовать общий провод. Соответствие восстанавливается выделением среза общего провода в новый независимый провод (алгоритм 3.1).

Алгоритм 3.1 — Частичное разделение проводов (wire slicing)

Вход: триггер-кандидат F ; провод w , подаваемый на D-вход F

Выход: пара «триггер — провод» со взаимно однозначным соответствием

- 1 **если** на F приходится лишь срез провода w , а не весь провод **то**
 - 2 $S \leftarrow$ срез провода w , соответствующий триггеру F ;
 - 3 $w' \leftarrow$ новый независимый провод той же разрядности, что и S ;
 - 4 перенаправить все входящие и исходящие связи среза S на w' ;
 - 5 **иначе** $w' \leftarrow w$;
 - 6 применить штатный проход извлечения инфраструктуры Yosys к паре (F, w') ;
-

Поддержка автоматов Мили. Поскольку выходная функция нерелевантна для рассматриваемых классов ошибок (недостижимые состояния, мертвые переходы, тупиковые состояния и взаимоблокировки), выходы, зависящие от сигналов помимо регистра состояния, отбрасываются, что позволяет единообразно обрабатывать автоматы Мили и Мура.

Обработка синхронного сброса. Для синхронных регистров логика сброса встроена в дерево мультиплексоров, что приводит к появлению переходов в начальное состояние из каждого состояния. Эта особенность критична для анализа тупиковых состояний и взаимоблокировок. Начальное состояние и корень логики переходов без логики сброса определяются по числу достижимых состояний

в поддеревьях корневого мультиплексора (алгоритм 3.2), что обеспечивает единообразную обработку синхронных регистров наравне с асинхронными.

Алгоритм 3.2 — Определение начального состояния при синхронном сбросе

Вход: корневой мультиплексор m дерева логики переходов триггера

Выход: начальное состояние s_0 ; корень логики переходов без логики сброса

- 1 **для каждого** поддерева T_i входов данных мультиплексора m : $n_i \leftarrow$
число достижимых состояний в поддереве T_i ;
 - 2 **если** существует ровно одно поддерево T_k с $n_k = 1$ **то**
 - 3 $s_0 \leftarrow$ единственное состояние в T_k ; // начальное состояние
 - 4 пометить ветвь управления, выбирающую вход k , как сигнал сброса;
 - 5 корнем логики переходов сделать мультиплексор m без поддерева T_k ;
 - 6 **иначе**
 - 7 обрабатывать регистр как асинхронный: $s_0 \leftarrow$ значение сброса
 регистра, корень логики переходов не изменяется
 - 8 **вернуть** (s_0 , новый корень логики переходов);
-

Вычислительная сложность процедуры извлечения определяется построением таблицы переходов: для каждого из $|S|$ возможных состояний схема вычисляется перебором комбинаций входов автомата, что в худшем случае дает сложность $O(|S| \cdot 2^{|I|} \cdot |M|)$, где $|I|$ — число входов автомата, $|M|$ — размер дерева мультиплексоров. На практике число входов автомата невелико (порядка десятков), а ранний останов при вычислении существенно сокращает число итераций, не улучшая, однако, оценку худшего случая. При переходах, зависящих от многоразрядных входных шин, именно перебор комбинаций входов становится доминирующей составляющей времени извлечения, что подтверждается экспериментально (глава 6). Поскольку дерево мультиплексоров представляет собой булеву схему над входами автомата, та же таблица переходов может быть получена одним структурным обходом дерева с восстановлением для каждого следующего состояния условия его выбора в виде дизъюнктивной нормальной формы или сокращенной упорядоченной двоичной решающей диаграммы (ROBDD) [82], что исключает множитель $2^{|I|}$ из оценки. Восстановление условий переходов в дизъюнктивной нормальной форме уже выполняется при сопоставлении результатов с исходным кодом (раздел 3.5).

3.4 Графовый анализ извлеченных автоматов

Пример графа конечного автомата с отмеченными дефектами (недостижимое состояние, мертвый переход, тупиковое состояние, взаимоблокировка) и соответствующего ему фрагмента описания приведен на рисунке 3.2.

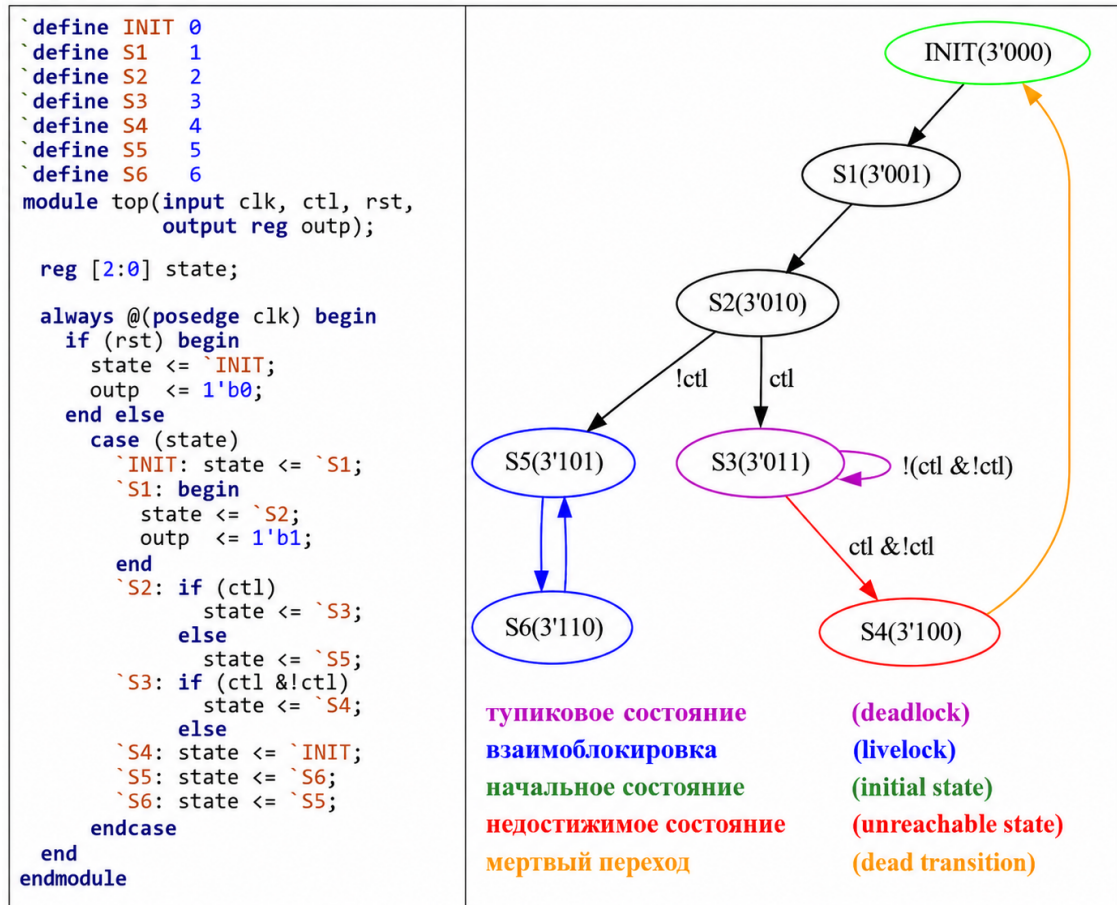


Рисунок 3.2 — Граф конечного автомата с отмеченными дефектами и соответствующее описание на SystemVerilog

После извлечения по каждому автомату строится граф автомата (раздел 3.1). Каждой вершине сопоставляется статус состояния (обычное, начальное, недостижимое, тупиковое, взаимоблокировка), а каждому ребру — статус перехода (обычный, мертвый, выявленный как мертвый по следствию) и условие перехода. Анализ выполняется в три последовательных этапа, каждый из которых уточняет граф для последующего: пометка мертвых переходов (раздел 3.4.1), анализ достижимости (раздел 3.4.2) и анализ живости (раздел 3.4.3). Реализация конвейера анализа описана в главе 5.

3.4.1 Анализ мертвых переходов

Обнаружение мертвых переходов основано на выявлении противоречий с участием входов автомата, принимающих постоянные значения. Такие входы могут возникать вследствие явных присваиваний, значений параметров или оптимизаций синтеза. Условие перехода реконструируется как конъюнкция атомарных предикатов над входами автомата, и переход классифицируется как мертвый, если хотя бы один предикат противоречит известному постоянному значению входа (алгоритм 3.3).

Алгоритм 3.3 — Обнаружение мертвых переходов по входам с постоянными значениями

Вход: граф автомата $G = (V, E)$; частичное отображение $K : I \rightarrow \{0, 1\}$
входов с постоянными значениями

Выход: ребра графа G с пометками мертвых переходов

- 1 **для каждого** ребра $e \in E$:
 - 2 представить условие перехода $\text{cond}(e)$ конъюнкцией атомарных предикатов над входами I ;
 - 3 **если** *некоторый атомарный предикат условия требует значения входа, противоречащего K* **то**
 - 4 пометить e как мертвый переход и исключить из дальнейшего графового анализа;
-

Удаление мертвых переходов повышает точность последующего анализа достижимости и анализа SCC, устраняя ложные пути, привнесенные артефактами синтеза. Более точное обнаружение невыполнимых переходов потребовало бы рассуждений о соотношениях между входами, не являющимися константами, что эквивалентно общей задаче удаления мертвого кода и вычислительно дорого, поэтому в методе мертвым считается только переход, условие которого противоречит постоянному значению одного из входов автомата.

3.4.2 Анализ недостижимых состояний

Недостижимые состояния выявляются обходом графа в ширину от начального состояния (алгоритм 3.4), а исходящие из них переходы помечаются как мертвые по следствию. Поскольку мертвые переходы при обходе не учитываются, состояние, достижимое лишь через них, корректно классифицируется как недостижимое.

Алгоритм 3.4 — Анализ достижимости состояний

Вход: граф $G = (V, E)$ с помеченными мертвыми переходами; начальное состояние s_0

Выход: множество недостижимых состояний

```

1  $R \leftarrow \emptyset$ ; очередь  $Q \leftarrow [s_0]$ ;
2 пока очередь  $Q$  не пуста :
3    $v \leftarrow$  извлечь из  $Q$ ; добавить  $v$  в  $R$ ;
4   для каждого ребра  $(v, u) \in E$ , не помеченного как мертвое :
5     если  $u \notin R$  то добавить  $u$  в  $Q$ ;
6 для каждого состояния  $v \in V \setminus R$  :
7   пометить  $v$  как недостижимое;
8   пометить все исходящие из  $v$  переходы как мертвые по следствию;
```

Как и анализ мертвых переходов (раздел 3.4.1), анализ достижимости является структурным: из графа удаляются только заведомо мертвые переходы (по входам с постоянными значениями), поэтому недостижимым не объявляется ни одно фактически достижимое состояние (отсутствие ложных срабатываний), однако состояние, недостижимость которого следует лишь из соотношений между входами, не являющимися константами, может остаться необнаруженным.

3.4.3 Анализ тупиковых состояний и взаимоблокировок

Обнаружение тупиковых состояний и взаимоблокировок основано на вычислении сильно связанных компонент алгоритмом Тарьяна [76] с временной

сложностью $O(|V| + |E|)$, где $|V|$ и $|E|$ — число вершин и ребер графа. Терминальные компоненты, тупиковые состояния и взаимоблокировки выделяются по определениям раздела 3.1 (алгоритм 3.5).

Алгоритм 3.5 — Анализ тупиковых состояний и взаимоблокировок на основе сильно связанных компонент

Вход: граф автомата $G = (V, E)$ с помеченными мертвыми переходами и недостижимыми состояниями; начальное состояние s_0

Выход: тупиковые состояния и группы состояний взаимоблокировки

- 1 $G' \leftarrow$ подграф G , образованный переходами, не помеченными как мертвые;
 - 2 вычислить сильно связанные компоненты G' алгоритмом Тарьяна;
 - 3 **для каждого** компоненты C :
 - 4 **если** $s_0 \in C$ **то перейти к следующей итерации;**
 - 5 **если существует ребро из вершины C к вершине вне C то перейти к следующей итерации;**
 - 6 исключить из C состояния, помеченные недостижимыми;
 - 7 **если** $C = \emptyset$ **то перейти к следующей итерации;**
 - 8 **если** $|C| = 1$ **то** пометить состояние из C как тупиковое **иначе** пометить все состояния C как взаимоблокировку;
-

Анализ выполняется над графом без мертвых переходов, а состояния, помеченные недостижимыми, не классифицируются, поэтому все классифицируемые компоненты достижимы из начального состояния.

3.5 Сопоставление результатов с исходным кодом

Поскольку информация об исходном коде не сохраняется на уровне сети логических элементов, для повышения интерпретируемости результатов разработан механизм сопоставления сети логических элементов с узлами АСД. Автоматы представляются пользователю в виде ориентированных графов в формате DOT системы Graphviz [83], вершины которых помечены именами состояний, а ребра — условиями переходов.

Сопоставление выполняется в два прохода.

1. *Сопоставление условий переходов.* Входы автомата сопоставляются с соответствующими выражениями АСД. Для этого формат атрибута, хранящего информацию об исходном расположении, был доработан: вместо имени файла и текстовой позиции хранится идентификатор источника и смещение в байтах, что позволяет однозначно восстановить символ АСД, соответствующий каждому проводу. Поддерживается также хранение расположения для отдельных битовых срезов и распространение информации о расположении между соединенными проводами. Это позволяет обходом АСД сопоставлять условия переходов с их расположением в исходном коде.
2. *Восстановление имен состояний.* Логика автомата нередко разделена между переменными текущего и следующего состояний, поэтому именованные константы состояний собираются по всем связанным присваиваниями символам АСД итеративным распространением до неподвижной точки, а символ регистра состояния находится по иерархическому имени (алгоритм 3.6).

Алгоритм 3.6 — Восстановление имен состояний автомата

Вход: множество символов АСД; граф ссылок между символами

Выход: имена состояний для каждого извлеченного автомата

```

1 для каждого символа АСД  $s$  :
2    $C(s) \leftarrow$  именованные константы, присваиваемые  $s$  или сравниваемые
   с  $s$ ;
3    $R(s) \leftarrow$  символы, присваиваемые  $s$  (в том числе через порты
   модулей);
4 повторять
5   для каждого символа  $s$  :  $C(s) \leftarrow C(s) \cup \bigcup_{t \in R(s)} C(t)$ ;
6 пока не будет достигнута неподвижная точка;
7 для каждого извлеченного автомата  $A$  :
8    $s \leftarrow$  символ АСД, найденный по иерархическому имени регистра
   состояния  $A$ ;
9   именами состояний  $A$  считать элементы множества  $C(s)$ ;
```

3.6 Обнаружение взаимодействующих автоматов

В практических RTL-описаниях управляющая логика редко реализуется единственным изолированным автоматом. Зачастую несколько автоматов взаимодействуют при реализации протоколов и конвейеров, и ошибки нередко возникают именно в их взаимодействии. Два автомата считаются *взаимодействующими*, если сигнал, формируемый одним автоматом, достигает логики переходов другого автомата по пути, содержащему произвольную комбинационную логику и не более одного последовательностного элемента. Порог в один последовательностный элемент соответствует типовой регистровой (конвейерной) развязке взаимодействующих контроллеров: сигнал, прошедший одну регистровую ступень, сохраняет непосредственное влияние на переходы принимающего автомата в следующем такте, тогда как через два и более регистров прямое управление переходами уже не прослеживается.

Взаимодействия обнаруживаются прямым обходом сети логических элементов от каждого выхода автомата с ограничением в один последовательностный элемент на пути (алгоритм 3.7). Обход линеен относительно размера сети логических элементов и вносит незначительные накладные расходы.

Алгоритм 3.7 — Обнаружение взаимодействующих конечных автоматов

Вход: множество извлеченных автоматов F ; сеть логических элементов N

Выход: граф зависимостей D между автоматами

- 1 $D \leftarrow \emptyset$;
 - 2 **для каждого** автомата $A \in F$ **и** каждого его выхода o :
 - 3 выполнить прямой обход сети N от выхода o : через комбинационные элементы — без ограничений, через триггеры — не более одного на каждом пути;
 - 4 **если** в ходе обхода достигнут вход автомата $B \in F$ **то** добавить в D направленную зависимость $A \rightarrow B$;
 - 5 **вернуть** D ; // циклы в D — обратная связь между контроллерами
-

Идентификация взаимодействующих автоматов помогает разработчикам понимать архитектурные зависимости управления, локализовать первопричины дефектов по нескольким управляющим блокам (например, когда условие тупикового состояния в одном автомате распространяется на зависимый автомат),

а также выявлять циклические зависимости, указывающие на непреднамеренную обратную связь между контроллерами.

3.7 Обработка параметризованных автоматов

Современные описания на SystemVerilog часто используют параметризованные модули, несколько экземпляров одного автомата в которых могут демонстрировать различное поведение в зависимости от значений параметров. Поскольку предлагаемый метод оперирует сетями логических элементов, построенными по развернутой модели с подставленными значениями параметров, каждый экземпляр автомата анализируется со своими конкретными значениями параметров. Это позволяет обнаруживать поведение, зависящее от конфигурации, включая состояния и переходы, достижимые в одном экземпляре и недостижимые в другом. Подобные эффекты наблюдались при экспериментальной оценке проекта BlackParrot. В модуле `br_ucse` (унифицированный контроллер кэша) один и тот же автомат используется в двух экземплярах с различными значениями параметров: как `icache_ucse` для кэша команд и как `dcache_ucse` для кэша данных. После подстановки значений параметров и извлечения из сети логических элементов два экземпляра дают неодинаковые графы: часть состояний, достижимых в экземпляре `dcache_ucse`, оказывается недостижимой в экземпляре `icache_ucse` (рисунок 3.3). Недостижимыми в `icache_ucse` оказываются состояния обратной записи (`e_uc_writeback_evict`, `e_uc_writeback_write_req` и последующие состояния `e_writeback_*`): переходы в них обусловлены параметром `writeback_p`, который для кэша команд равен нулю, поскольку в конфигурации проекта кэш команд объявлен без признака обратной записи. Это соответствует архитектуре: кэш команд только читается ядром, не содержит измененных строк и не выполняет обратную запись в память, а кэш данных ее выполняет. Разработчики зафиксировали тот же замысел утверждением в конце модуля, запрещающим состояния обратной записи при нулевом значении параметра. Таким образом, анализ по экземплярам не только ищет ошибки, но и подтверждает замысел разработчика.

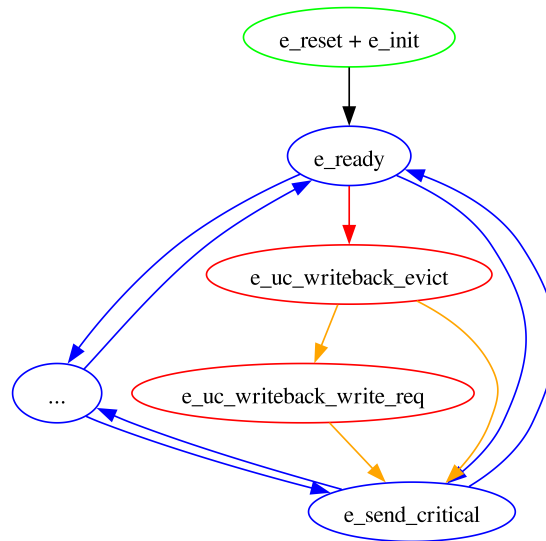


Рисунок 3.3 — Два экземпляра одного извлеченного автомата модуля `br_use` проекта BlackParrot с различными значениями параметров: выделенные состояния недостижимы в экземпляре `icache_use`, но достижимы в `dcache_use`

3.8 Выводы по главе

В главе введены формальные определения дефектов конечных автоматов, обоснован выбор подхода к извлечению на основе сети логических элементов, обеспечивающего независимость от стиля кодирования, и представлен разработанный в работе метод извлечения и статического анализа конечных автоматов из RTL-описаний. Метод сочетает извлечение автоматов из представления RTLIL (с доработками для поддержки частичного разделения проводов, автоматов Мили и синхронного сброса) с графовым анализом, выявляющим мертвые переходы, недостижимые состояния, тупиковые состояния и взаимоблокировки, а также с механизмом сопоставления результатов с исходным кодом и обнаружением взаимодействующих автоматов. Результаты главы соответствуют второму положению, выносимому на защиту.

Глава 4. Метод интеграции сопутствующих проектных форматов

В настоящей главе представлен разработанный в работе метод интеграции сопутствующих проектных форматов Liberty, UPF и SDF (раздел 1.4) в статический анализ. Основой метода служит единая информационная модель, которая входит во внутреннее представление (раздел 2.1.2) и связывает развернутую модель с объектами сопутствующих описаний. Наибольшую сложность для интеграции представляет UPF, поскольку корректная интерпретация его команд требует построения отдельной модели подсистемы питания и ее связывания с HDL-описанием, поэтому основное внимание уделяется построению модели подсистемы питания по HDL-описанию, описанию подсистемы питания в формате UPF и библиотекам стандартных ячеек Liberty и ее статическому анализу. В настоящей главе HDL-описанием называется описание на языках Verilog, SystemVerilog или VHDL независимо от уровня абстракции, поскольку сопутствующие форматы применяются и к RTL-описаниям, и к спискам соединений с экземплярами библиотечных ячеек. Детали программной реализации соответствующих компонентов вынесены в главу 5.

4.1 Формат описания подсистемы питания UPF

Для снижения энергопотребления интегральную схему разделяют на отдельные части, называемые *доменами питания* (power domain), для каждого из которых задаются различные параметры распределения питания. Совокупность описаний питания для всех доменов представляет собой описание подсистемы питания (power intent). Языки Verilog/SystemVerilog и VHDL не предоставляют средств описания подсистемы питания, для чего и был разработан формат UPF (Unified Power Format, IEEE Std 1801 [47]).

Синтаксически команды UPF представляют собой расширение языка сценариев Tcl (Tool Command Language). Важным свойством UPF является его независимость от логических операций в HDL-описании: единственным связующим звеном является иерархия объектов (модулей, экземпляров, логических портов и сигналов).

4.1.1 Основные компоненты подсистемы питания

В подсистеме питания экземпляры модулей из логического описания группируются в домены питания командой `create_power_domain`. Распределение питания между отдельными доменами и внешними источниками обеспечивается через порты (`supply port`) и сети питания (`supply net`). Режимы работы компонентов определяются состояниями сетей питания, подключенных к ним. Совокупность сетей питания, играющих отдельную роль, образует *набор сетей питания* (`supply set`), далее для краткости *набор питания*. Особый интерес представляют две обязательные сети набора, отвечающие за соединение с источником напряжения и с землей. Состояния и переходы между ними задаются командами `add_power_state` и `add_state_transition`.

Для портов, сетей и наборов питания стандарт определяет отношение эквивалентности двух видов. *Электрически эквивалентны* соединенные между собой порты и сети питания, а также объекты, объявленные электрически эквивалентными командой `set_equivalent`. *Функционально эквивалентны* электрически эквивалентные объекты, а также объекты, объявленные функционально эквивалентными той же командой. Два набора питания эквивалентны, если эквивалентны сети, выполняющие в них одинаковые функции.

Корректное взаимодействие компонентов с разными уровнями напряжения и режимами питания обеспечивается *ячейками изоляции* (`isolation cell`) и *преобразователями уровня напряжения* (`level shifter`). Ячейки изоляции вставляются между двумя объектами системы, один из которых может перейти в режим «выключен», тогда как другой продолжает работу, предотвращая ошибки из-за неопределенного состояния выходных сигналов отключенного объекта. Преобразователи уровня напряжения обеспечивают корректную связь между двумя включенными объектами с разными уровнями напряжения. Регистры сохранения состояния (`retention state register`) поддерживают минимальное напряжение для сохранения значений в ячейках памяти внутри отключенного домена.

Пример описания подсистемы питания приведен в листинге 4.1: описание из двух модулей на SystemVerilog и минимальное описание его подсистемы питания из двух доменов по версии IEEE 1801-2018. Описание питания задает версию стандарта, домены и их основные наборы сетей питания, состояния питания,

ключ питания отключаемого домена и стратегии на его границе, а сигналы управления ключом и изоляцией берет из портов модуля top.

Листинг 4.1: Описание на SystemVerilog (вверху) и минимальное описание его подсистемы питания в формате UPF (внизу)

```

1 module top (
2     input logic      clk,
3     input logic      pg_en,
4     input logic      iso_en,
5     input logic [7:0] d_in,
6     output logic [7:0] d_out
7 );
8     logic [7:0] core_out;
9     core u_core (.clk(clk), .d_in(d_in), .d_out(core_out));
10    assign d_out = core_out;
11 endmodule
12
13 module core (
14     input logic      clk,
15     input logic [7:0] d_in,
16     output logic [7:0] d_out
17 );
18     always_ff @(posedge clk)
19         d_out <= d_in;
20 endmodule

```

```

1 upf_version 3.1
2 set_design_top top
3 create_supply_net VDD
4 create_supply_net VDD_LO
5 create_supply_net VSS
6 create_supply_net VDD_SW
7 create_supply_set ss_top -function {power VDD} -function {ground VSS}
8 create_supply_set ss_core -function {power VDD_SW} -function {ground VSS}
9 create_power_domain PD_TOP -elements {.*} -supply {primary ss_top}
10 create_power_domain PD_CORE -elements {u_core} -supply {primary ss_core}
11 add_power_state -supply PD_TOP.primary \
12     -state {ON -supply_expr {power == {FULL_ON 1.0}} && ground == {FULL_ON 0.0}}
13 add_power_state -supply PD_CORE.primary \
14     -state {ON -logic_expr {pg_en == 1} \
15         -supply_expr {power == {FULL_ON 0.8}} && ground == {FULL_ON 0.0}} \
16     -state {OFF -logic_expr {pg_en == 0} -supply_expr {power == OFF}}
17 create_power_switch sw_core -domain PD_CORE \
18     -input_supply_port {vin VDD_LO} -output_supply_port {vout VDD_SW} \
19     -control_port {ctrl pg_en} -on_state {on vin {ctrl}} -off_state {off {!ctrl}}
20 set_isolation iso_core -domain PD_CORE -applies_to outputs -clamp_value 0 \
21     -isolation_signal iso_en -isolation_sense high \
22     -isolation_supply ss_top -location parent
23 set_level_shifter ls_core -domain PD_CORE -applies_to both -rule both

```

Пример показывает источник сложности анализа. Домен PD_CORE имеет состояние «выключен», а домен PD_TOP всегда включен, значит на каждом выходе PD_CORE нужна ячейка изоляции, а на каждом порту между доменами с напряжениями 0,8 и 1,0 В нужен преобразователь уровня напряжения. Применимость стратегий `iso_core` и `ls_core` к конкретному порту определяется не перечислением портов, а фильтрами, учитывающими направление порта, границу домена, соотношение напряжений и эквивалентность сетей питания источника и приемника (раздел 4.4). Смысл описания зависит от версии стандарта, которая задается явно командой `upf_version` (раздел 4.1.2).

4.1.2 Эволюция стандарта

Стандарт UPF активно развивается: впервые UPF 1.0 был представлен организацией Accellera в 2007 г., с 2009 года развитие ведется в рамках IEEE, а актуальной версией является UPF 4.0 (IEEE 1801-2024). От версии к версии множество команд изменялось: команды запросов версии 2.0 в версии 2.1 вынесены в приложение, а с версии 3.0 заменены информационной моделью, восемь команд версии 2.1 объявлены устаревшими и в версии 3.0 удалены, версия 4.0 расширяет стандарт на аналого-цифровые блоки и регистры сохранения состояния и добавляет шесть команд. Стандарт не определяет семантику описания, составленного из команд разных версий (команда `upf_version`), поэтому привязка реализации к одной версии соответствует стандарту, а версия описания задается явно. За основу в работе взят стандарт IEEE 1801-2018 (UPF 3.1) как действующий на момент разработки: реализованы все его команды. Команды, отнесенные в этой версии к устаревшим, принимаются с диагностикой, а команды, отсутствующие в ней, диагностируются как неизвестные. Исключение составляют команды запроса версии 2.0, которые поддерживаются для совместимости с описаниями, составленными по ранним версиям стандарта.

4.1.3 Аппаратная реализация и типы ошибок управления питанием

На физическом уровне поддержка множественных доменов питания реализуется специальными ячейками технологической библиотеки (описываемыми в формате Liberty): ключами питания, ячейками изоляции, преобразователями уровня напряжения (в том числе управляемыми, объединяющими функции изоляции и преобразования уровня напряжения), повторителями и триггерами сохранения состояния.

Возможность выключения питания части схемы (с последующим включением и восстановлением состояния), а также передача сигналов между доменами с разными уровнями напряжения порождают пять основных типов ошибок, которые желательно выявлять на ранних этапах проектирования.

1. *Отсутствие или некорректная реализация преобразования уровня напряжения.* Преобразование задается командой `set_level_shifter`.
2. *Отсутствие или некорректная реализация изоляции домена.* Изоляция задается командами `set_isolation` и `set_isolation_control`.
3. *Отсутствие или некорректная реализация сохранения состояния.* Сохранение задается командами `set_retention` и `set_retention_control`.
4. *Переход системы в некорректное состояние питания.* Допустимые и недопустимые состояния и переходы описываются состояниями питания (команда `add_power_state`), а в ранних версиях стандарта — таблицами состояний питания (Power State Table, PST, команды `create_pst` и `add_pst_state`).
5. *Некорректная инициализация значений.* Установка данных при обработке события включения питания может приводить к гонкам и появлению неопределенных значений.

Статический анализ применяется для выявления ошибок первых двух типов на основе состояний питания (критерий необходимости стратегии сформулирован в разделе 4.4). Ошибки остальных трех типов связаны с поведением схемы при переключении состояний питания и выявляются главным образом имитационным моделированием с учетом питания и формальной верификацией. Дополнительно проверяется, что управляющие сигналы не исходят из выключенных доменов, не проходят через них и не принимают неопределенных значений,

а специальные ячейки не вставляются в цепи тактовых сигналов, сигналов сброса и цепи подтяжки. Пример схемы с преобразованием уровня напряжения между двумя доменами приведен на рисунке 4.1.

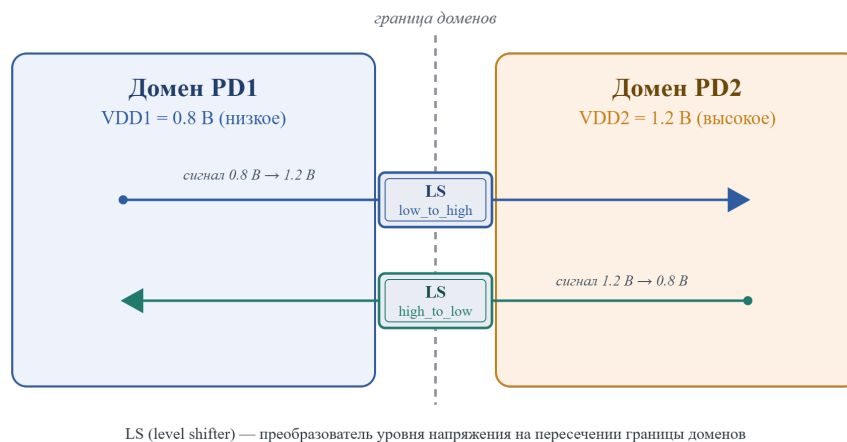


Рисунок 4.1 — Передача сигналов между доменами с разными уровнями напряжения через преобразователи уровня напряжения (low_to_high и high_to_low)

4.2 Построение единой информационной модели

Единая информационная модель связывает узлы развернутой модели HDL-описания (раздел 2.1.1) с объектами сопутствующих описаний. Метод интеграции един для всех трех форматов и включает три шага. На первом шаге по описанию строится объектная модель формата: модель библиотечных ячеек Liberty, модель подсистемы питания UPF, семантическая модель SDF. На втором шаге объекты этой модели привязываются по именам к узлам развернутой модели: ячейки Liberty к транслированным модулям, домены питания и стратегии UPF к экземплярам и портам иерархии, записи SDF к экземплярам, путям задержек и временным проверкам. Несоответствия, выявленные при привязке UPF- и SDF-описаний, сообщаются диагностикой. На третьем шаге над связанной моделью выполняются проверки, для которых недостаточно одного HDL-описания (разделы 4.4 и 4.5).

Из примеров команд UPF следует, что для корректной интерпретации UPF-описания и построения модели подсистемы питания необходима информация об объектах из HDL-описания, поэтому для формата UPF привязка выполняется в ходе самой интерпретации. В качестве входных данных, помимо файлов

с описанием подсистемы питания, используется развернутая модель, построенная по соответствующему HDL-описанию. Кроме того, для полной и корректной интерпретации требуется обработка библиотек стандартных ячеек в формате Liberty: HDL-описание задает иерархию экземпляров, а библиотечные ячейки (указываемые в командах `use_interface_cell` и `map_power_switch`) задают, какая именно ячейка должна реализовывать компонент на схеме.

Таким образом, единая информационная модель строится из четырех источников: развернутой модели HDL-описания, объектной модели Liberty-ячеек, интерпретированного UPF-описания и семантической модели SDF-описания. Схема построения единой информационной модели приведена на рисунке 4.2, привязка SDF-описания описана в разделе 4.2.2.

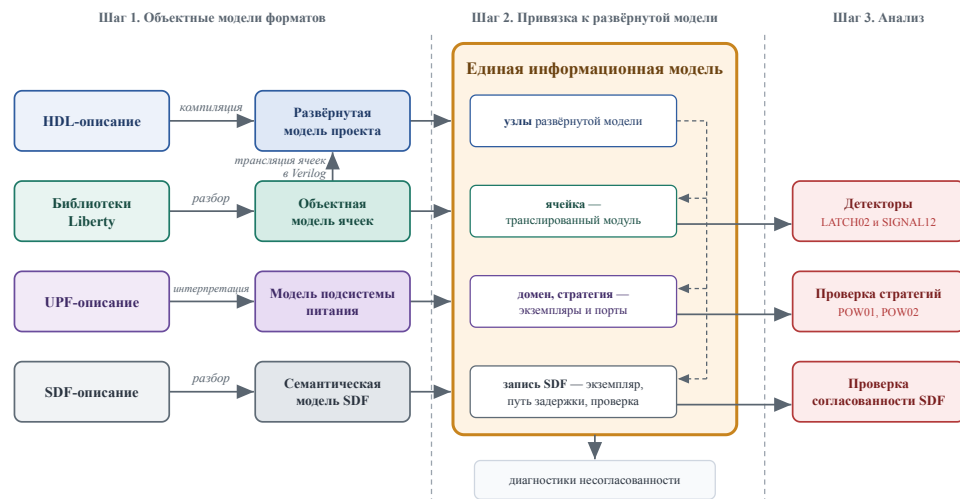


Рисунок 4.2 — Построение единой информационной модели из HDL-описания, библиотек Liberty, UPF- и SDF-описаний

4.2.1 Обработка библиотек стандартных ячеек Liberty

По результатам разбора Liberty-описаний строится объектная модель библиотечных ячеек, хранящая атрибуты выводов, направления, логические функции и описания последовательностных элементов: триггеров и защелок с сигналами `clocked_on`, `clear`, `preset`.

Поскольку анализатор оперирует RTL-описанием на языках Verilog и SystemVerilog, для межмодульного анализа разработан транслятор библиотечных ячеек Liberty в Verilog: без него экземпляры библиотечных ячеек

оставались бы в модели «черными ящиками», недоступными для анализа. Транслятор по описанию последовательностной ячейки восстанавливает эквивалентное поведенческое описание (алгоритм 4.1), которое подключается к компилятору наравне с пользовательским RTL-описанием, что позволяет анализировать схему, требующую информации о характеристиках библиотечных ячеек, единообразно с остальной частью описания.

Алгоритм 4.1 — Трансляция последовательностной ячейки Liberty в поведенческий Verilog

Вход: описание ячейки Liberty: группы `ff/ff_bank/latch/latch_bank`; выходы `pin` с атрибутами `direction` и `function`

Выход: поведенческий модуль Verilog

- 1 Для группы `ff(var1, var2)`: назначить `var1` прямым узлом состояния (например, `IQ`), `var2` — инверсным (`IQN`); атрибут `bits` задает разрядность (для `ff_bank`);
 - 2 Разобрать булевы атрибуты `next_state`, `clocked_on`, `clocked_on_also`, `clear`, `preset` в выражения Verilog, перенося активный уровень асинхронных сигналов в сами выражения;
 - 3 Сформировать список чувствительности `always @(posedge clk [or posedge clear] [or posedge preset])` по внутренним проводам выражений;
 - 4 Задать в теле блока приоритетные ветви: при активном `clear` — `IQ <= 0`, при активном `preset` — `IQ <= 1`, иначе — захват следующего состояния `IQ <= next_state` по фронту такта;
 - 5 **если одновременно активны `clear` и `preset` то** присвоить значение по атрибутам `clear_preset_var1/var2`: `L` → 0, `H` → 1, `N` → без изменения, `T` → переключение (через сторожевой регистр, срабатывающий однократно при входе в условие);
 - 6 **если задан `clocked_on_also` (master-slave) то** выпустить два `always`-блока: ведущая часть пишет промежуточный регистр `IQ_M` из `next_state`, ведомая переносит `IQ <= IQ_M` по своему такту;
 - 7 Добавить инверсный выход `IQN <= ~IQ`;
 - 8 **если ячейка — защелка (`latch`) то** построить уровнечувствительный мультиплексор `IQ <= enable ? data_in : IQ`;
 - 9 Транслировать выходы `pin` с атрибутом `function` в непрерывные присваивания (буферы, элементы `and/or`, ячейки изоляции — как `and` с входом разрешения); внешние выходы связать с внутренними узлами состояния;
 - 10 **если задана группа `generated_clock` то** построить тактовую цепь: переключение по фронту `master_pin` либо делитель на основе счетчика;
 - 11 Отобразить единицу времени `time_unit` в директиву ``timescale`;
 - 12 **если последовательностная группа объявлена, но ее выходы не разрешаются то** зафиксировать диагностику (ячейка не пропускается без диагностики);
-

Ключевой деталью алгоритма 4.1 является кодирование активного уровня асинхронных сигналов: инверсия в выражениях Liberty (! или ') переносится в само выражение установки/сброса, поэтому в списке чувствительности всегда используется `posedge` внутреннего провода, хранящего это выражение (корректная идиома асинхронного сброса: для активного низкого сброса выражение $\neg RST$ дает срабатывание по заднему фронту `RST`).

Транслированные модули оборачиваются директивами ``celldefine ...`resetall`, поэтому экземпляры таких модулей в развернутой модели несут признак библиотечной ячейки (признак `celldefine`). Этот признак служит связующим звеном: по нему как модель подсистемы питания, так и детекторы статического анализа (раздел 4.5) отличают стандартные ячейки технологической библиотеки от пользовательской логики. Классификация библиотечных ячеек и их связывание с HDL-символами в единую информационную модель (рисунок 4.2) выполняются по алгоритму 4.2.

Алгоритм 4.2 — Построение единой информационной модели и связывание Liberty-ячеек с HDL

Вход: объектная модель ячеек Liberty; развернутая модель HDL-описания; интерпретированное UPF-описание; семантическая модель SDF-описания (если задано)

Выход: единая информационная модель

- 1 **для каждого** ячейки *c* объектной модели Liberty :
 - 2 классифицировать тип *c* по атрибутам: `is_isolation_cell` — ячейка изоляции, `is_level_shifter` — преобразователь уровня напряжения, `switch_cell_type` (по наличию атрибута) — ключ питания, `retention_cell` — ячейка сохранения состояния;
 - 3 **если** *c* относится к одному из типов подсистемы питания **то** создать связующий объект со ссылкой на Liberty-описание и флагом происхождения «Liberty»;
 - 4 найти HDL-символ по имени ячейки в развернутой модели (имя транслированного модуля совпадает с именем Liberty-ячейки);
 - 5 **если** HDL-символ найден **то** установить ссылку на него и флаг «Verilog»; иначе оставить ячейку связанной только с Liberty;
 - 6 Связать команды UPF (`use_interface_cell`, `map_power_switch`) с соответствующими связующими объектами по имени ячейки, устанавливая флаг «UPF»;
 - 7 **если** задано SDF-описание **то** привязать его записи к узлам развернутой модели (алгоритм 4.3);
 - 8 **вернуть** единую модель, связывающую развернутую модель с объектами Liberty, UPF и SDF
-

4.2.2 Интеграция формата задержек SDF

Формат SDF, определенный стандартом IEEE Std 1497-2001 [50], передает временные характеристики реализованной схемы из средств расчета задержек в средства имитационного моделирования: файл перечисляет экземпляры ячеек схемы (конструкция CELL с типом и иерархическим именем экземпляра), задержки путей «вход — выход» (IOPATH, в том числе условные COND), задержки межсоединений (INTERCONNECT, PORT, NETDELAY), пределы временных проверок (TIMINGCHECK) и новые значения параметров `specparam` (LABEL). Правила сопоставления этих конструкций с конструкциями SystemVerilog (пути блоков `specify`, системные временные проверки, параметры `specparam`, соединения портов) определены разделом 32 стандарта IEEE 1800. В маршруте проектирования сопоставление выполняет система имитационного моделирования при моделировании на вентиляльном уровне, а конструкция, не нашедшая адресата, по стандарту вызывает лишь предупреждение. Поэтому рассогласование списка соединений вентиляльного уровня (gate-level netlist, описания схемы в терминах экземпляров библиотечных ячеек и соединений между ними) и SDF-описания (после инженерных изменений, при смене версии библиотеки ячеек, при различиях в записи иерархических имен) остается незамеченным: часть задержек не аннотируется, а результат многочасового имитационного моделирования недостоверен.

В работе поддержка SDF реализована как статическая проверка согласованности SDF-описания с развернутой моделью HDL-описания: сопоставление, которое выполняет аннотатор системы имитационного моделирования, проводится по правилам раздела 32 стандарта заранее, без имитационного моделирования, а его результат выдается диагностиками с позицией в SDF-файле. Проверка включает три уровня. Разбор файла по полной грамматике формата строит семантическую модель SDF, не зависящую от HDL-описания: заголовок (версия, разделитель иерархии, масштаб времени), ячейки, записи задержек, временных проверок и параметров с тройками значений «минимальное : типичное : максимальное». Над моделью выполняются проверки, не требующие HDL-описания: упорядоченность троек, допустимое число значений задержки, соответствие конструкций версии формата. Привязка модели к развернутой модели HDL-описания

выполняется для заданной области (модуля верхнего уровня, явно указанной области либо вызова системной задачи `$sdf_annotate`) по следующим правилам.

- Иерархическое имя экземпляра разрешается пошагово с учетом экранирования, массивов экземпляров и блоков генерации, в том числе безымянных. Отсутствие экземпляра — ошибка, несовпадение типа ячейки с определением — предупреждение.
- Запись `IOPATH` сопоставляется с путями блока `specify` экземпляра по портам источника и приемника и фронту. Условная задержка сопоставляется с условным путем по канонической форме условия (эквивалентность операторов, свертка сравнений с константами, нормализация порядка операндов), безусловная — со всеми путями между портами.
- Запись `TIMINGCHECK` сопоставляется с системными временными проверками по табличному отображению видов проверок с учетом различного порядка аргументов, причем фронт и условие сигнала сопоставляются, только если заданы в `SDF`.
- Запись `LABEL` сопоставляется с параметром `specparam` модуля или блока `specify`.
- Записи межсоединений сопоставляются с портами и цепями с проверкой направления и принадлежности источника и приемника одной цепи, которая устанавливается обходом псевдонимов цепи вверх по иерархии.
- Конструкции, не относящиеся к имитационному моделированию (`TIMINGENV`), пропускаются без диагностик, как предписано стандартом.

Результат привязки, отображение конструкций `SDF` на символы развернутой модели (экземпляры, пути задержек, временные проверки, параметры, порты), сохраняется и доступен другим проверкам наравне с общими данными (раздел 2.9). Проверка приведена в алгоритме 4.3. Вычисление эффективных значений задержек (расширение до двенадцати переходов, пределы отбраковки импульсов, накопление режимов `ABSOLUTE` и `INCREMENT`, выбор компоненты тройки) выполняется системой имитационного моделирования при обратной аннотации и в задачу проверки согласованности не входит. Детали реализации приведены в главе 5.

Алгоритм 4.3 — Статическая проверка согласованности SDF-описания с развернутой моделью HDL-описания

Вход: семантическая модель SDF F ; корневой экземпляр r (область привязки)

Выход: отображение конструкций F на символы развернутой модели; диагностики

```

1  для каждого ячейки  $c \in F$  :
2       $e \leftarrow$  экземпляр, полученный пошаговым разрешением пути  $c$  от  $r$ 
        (экранирование, массивы экземпляров, блоки генерации);
3      если  $e$  не найден то выдать ошибку с последним найденным уровнем;
        перейти к следующей итерации;
4      если тип ячейки  $c$  не совпадает с определением  $e$  то выдать
        предупреждение;
5       $T \leftarrow$  пути задержек, временные проверки и параметры блока specify
        тела  $e$  (собираются один раз на тело);
6      для каждого записи  $d$  ячейки  $c$  (задержка, временная проверка,
        параметр, межсоединение) :
7           $S \leftarrow$  элементы  $T$ , порты и цепи, сопоставленные с  $d$  по портам,
            фронту, виду проверки и канонической форме условия;
8          если  $S = \emptyset$  то выдать предупреждение об отсутствии адресата  $d$ ;
9          иначе сохранить  $d \rightarrow S$ ;
```

4.3 Интерпретация UPF-описания

Поскольку UPF является надстройкой над Tcl, для корректной обработки файлов требуется интерпретатор Tcl. Имена объектов HDL-описания в командах UPF разрешаются по развернутой модели, в том числе командой `find_objects`. Каждой команде стандарта сопоставляется собственный обработчик: встретив в процессе интерпретации зарегистрированную команду, интерпретатор вызывает обработчик, который выполняет первичную проверку корректности аргументов, реализует логику команды (создание объектов модели, изменение характеристик или установку связей с развернутой моделью) и возвращает код

состояния. Результатом этапа интерпретации является объектная модель подсистемы питания. Интерпретация выполняется после построения развернутой модели анализируемого HDL-описания. При успешном завершении этапа построенная модель передается в модуль статического анализа.

4.4 Статический анализ модели подсистемы питания

Анализ UPF-описания направлен на проверку корректности построенной модели. Проверяемые свойства разделяются на два типа:

1. корректность конкретных языковых конструкций (использование неподдерживаемой команды или опции, обращение к объекту до его создания или вне области видимости, некорректное использование опций и аргументов);
2. корректность полученной модели (отсутствующие или некорректно заданные объекты, в частности стратегии изоляции и преобразования уровня напряжения).

Проверка свойств первого типа выполняется в процессе интерпретации. Для проверки свойств второго типа реализован статический анализатор модели, рассматривающий два наиболее значимых компонента: ячейки изоляции и преобразователи уровня напряжения.

Домены питания представляют собой группы экземпляров модулей (или ячеек) из HDL-описания. *Граничным портом* называется порт экземпляра, через который сигнал передается между экземплярами, принадлежащими разным доменам. *Соседними* считаются домены, между элементами которых есть хотя бы один граничный порт. При описании стратегий изоляции и преобразования уровня напряжения через опции соответствующих команд задаются фильтры, определяющие, для каких портов задана данная стратегия. Проверка наличия и корректности стратегий сводится к обходу всех граничных портов и определению для каждого из них соответствующей стратегии путем сопоставления параметров порта с условиями заданных фильтров.

Необходимость стратегии определяется для каждого граничного порта по наборам питания *источника* (стороны, управляющей сигналом порта) и *приемника*. Двухнаправленные порты не проверяются, так как у них нельзя выделить

источник и приемник. Стратегии рассматриваются сначала для родительского домена, затем для дочернего. Стратегия изоляции необходима, если у набора питания источника есть состояние «выключено», а наборы питания источника и приемника не эквивалентны ни функционально, ни электрически (раздел 4.1.1). В отсутствие стратегии изоляции из выключенного домена во включенный передавалось бы неопределенное значение, что указывает на нарушение требования корректности (рисунок 4.3). Стратегия преобразования уровня напряжения необходима, если наборы питания источника и приемника не эквивалентны и работают при разных номинальных напряжениях. Если заданы группы состояний питания, определяющие допустимые сочетания состояний наборов, напряжения сравниваются в каждом допустимом сочетании, и преобразование требуется, если хотя бы в одном из них номинальные напряжения наборов положительны и различны. Иначе сравниваются множества номинальных напряжений наборов, и преобразование требуется, если эти множества различны или одно из них содержит несколько значений. Набор без сведений о напряжении преобразования уровня напряжения не требует. В отсутствие такой стратегии значение сигнала, передаваемого между доменами, искажается.

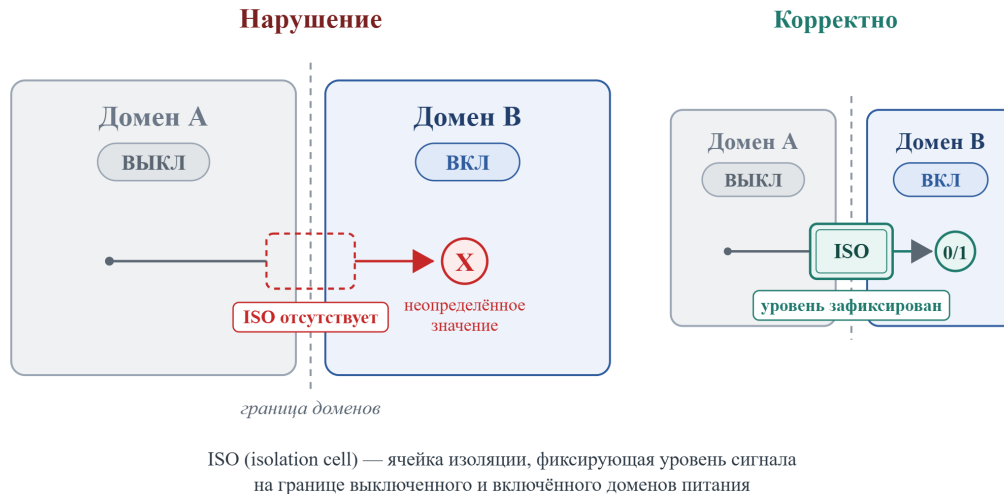


Рисунок 4.3 — Граница выключенного и включенного доменов питания: слева — нарушение (при отсутствии ячейки изоляции во включенный домен передается неопределенное значение), справа — корректная реализация с ячейкой изоляции

Проверка выполняется для каждой пары соседних доменов по алгоритму 4.4. Для изоляции она проверяет правило POW02, для преобразования уровня напряжения — правило POW01.

Алгоритм 4.4 — Проверка стратегий изоляции и преобразования уровня напряжения на границах доменов питания

Вход: модель подсистемы питания: домены питания, наборы питания с состояниями, стратегии с наборами фильтров, а также вид проверяемой стратегии k (изоляция или преобразование уровня напряжения)

Выход: диагностики о нарушениях стратегий вида k

- 1 $P \leftarrow$ пары доменов, между элементами которых есть граничные порты;
 - 2 **для каждого** пары доменов $(D_1, D_2) \in P$:
 - 3 **для каждого** граничного порта p с набором питания источника a
и приемника b :
 - 4 **если** p двунаправленный **или не ТребуетсяСтратегия**(k, a, b) **то**
перейти к следующей итерации;
 - 5 $S \leftarrow$ стратегии вида k , заданные для доменов D_1 и D_2 (сначала
родительского домена, затем дочернего);
 - 6 **если** $S = \emptyset$ **то**
 - 7 выдать диагностику об отсутствии стратегии для порта p ;
 - 8 **перейти к следующей итерации;**
 - 9 **для каждого** стратегии $s \in S$: отметить фильтры s , которым p
не соответствует;
 - 10 **если нет** стратегии, всем фильтрам которой p соответствует
то выдать диагностику по наиболее значимому невыполненному
фильтру с указанием порта p ;
 - 11 **функция** ТребуетсяСтратегия(k, a, b):
 - 12 **если** a и b эквивалентны функционально или электрически **то вернуть**
ложь;
 - 13 **если** k — изоляция **то вернуть** у a есть состояние «выключено»;
 - 14 **если** заданы допустимые сочетания состояний a и b **то вернуть**
в одном из них номинальные напряжения a и b положительны
и различны;
 - 15 **вернуть** множества номинальных напряжений a и b непусты
и различны или одно из них содержит несколько значений;
-

Вычислительная стоимость проверки полиномиальна: она ограничена произведением числа пар соседних доменов, числа портов на их границах и суммарного числа фильтров заданных для них стратегий. Определение необходимости стратегии для порта требует, кроме того, перебора состояний питания наборов источника и приемника.

Фильтры стратегии упорядочены по значимости и ниже перечислены в порядке ее убывания. При несоответствии порта сразу нескольким фильтрам диагностика выдается по наиболее значимому невыполненному фильтру, что сокращает число сообщений и упрощает локализацию причины.

Для стратегии *изоляции* рассматриваются следующие фильтры: соответствие сети питания приемника (-sink) и источника (-source) сигнала, направление порта (-applies_to), граница домена (-applies_to_boundary), эквивалентность сетей питания (-diff_supply_only), значения при отключении питания и источника (-applies_to_clamp и подобные). Если все фильтры выполнены, но стратегия для порта не задана, выдается диагностика об отсутствии стратегии. Для стратегии *преобразования уровня напряжения* дополнительно проверяются направление изменения напряжения (-rule: low_to_high, high_to_low или both), минимальная разница напряжений (-threshold) и местоположение ячейки (-location). Сообщение об ошибке содержит имя порта, тип нарушенного фильтра и номер строки команды в UPF-описании, что облегчает локализацию дефекта.

4.5 Анализ RTL-описаний с привлечением библиотек Liberty

Помимо построения модели подсистемы питания, информация из библиотек Liberty непосредственно используется рядом детекторов статического анализа RTL, отличающих библиотечные ячейки от пользовательской логики по признаку `celldefine` (раздел 4.2.1). В рамках работы реализованы два таких детектора.

Обнаружение RS-защелок. Реализован детектор правила LATCH02, выявляющий RS-защелки, описанные перекрестно связанными логическими элементами. Благодаря транслятору Liberty (раздел 4.2.1) детектор работает единообразно как над встроенными вентильными примитивами языка (and, or, nand, nor, not), так и над библиотечными ячейками: оба источника при-

водятся к единому *дескриптору логического элемента* — тройке $\langle \text{логическая функция, выходной сигнал, входные сигналы} \rangle$. Для примитива функция берется из его типа, для библиотечной ячейки восстанавливается из правой части непрерывного присваивания ее единственного выхода по образцам: $\neg(a \wedge b) \rightarrow \text{nand}$, $\neg(a \vee b) \rightarrow \text{nor}$, $a \wedge b \rightarrow \text{and}$, $a \vee b \rightarrow \text{or}$, $\neg a \rightarrow \text{not}$. Процедура обнаружения приведена в алгоритме 4.5.

Алгоритм 4.5 — Обнаружение RS-защелок на примитивах и библиотечных ячейках

Вход: экземпляры встроенных вентильных примитивов и библиотечных ячеек (признак `celldefine`) в теле модуля

Выход: предупреждения о RS-защелках

- 1 **для каждого** элемента g :
 - 2 восстановить дескриптор $\langle f, out, ins \rangle$: для примитива функция f — из его типа (многовходовые элементы `and`, `or`, `nand`, `nor`, `xor` с числом входов, отличным от двух, пропускаются), для библиотечной ячейки — из непрерывного присваивания ее выхода; нераспознанные функции (`xor`, `buf`) помечаются как `other`;
 - 3 Построить отображение каждого сигнала в список пар (направление порта, элемент);
 - 4 **для каждого** элемента a типа `and` или `or` :
 - 5 **если** выход a подан только на входы элементов `not` и не является выходным портом модуля **то** считать композицию a с каждым таким `not` элементом `nand` или `nor` соответственно;
 - 6 **для каждого** пары элементов (g_1, g_2) одного типа, `nand` или `nor` (с учетом композиций), где выход каждого подан на вход другого :
 - 7 выдать предупреждение о RS-защелке с указанием обоих элементов;
-

RS-защелка, описанная на вентилях, нарушает правило LATCH02: средства временного анализа воспринимают ее как обратную связь в комбинационной схеме, что затрудняет статический временной анализ и синтез.

Обнаружение буферизованных тактовых сигналов. Реализован детектор правила SIGNAL12, выявляющий тактовые сигналы, проходящие через библиотечные буферные ячейки или буферные примитивы перед использованием

в списке чувствительности последовательностной логики (алгоритм 4.6). Предупреждение о тактовом сигнале, поданном на тактовый вход триггера через буфер, выдается, так как распространение тактовых сигналов через произвольную логику затрудняет анализ синхронизации и физическое проектирование тактового дерева.

Алгоритм 4.6 — Обнаружение буферизованных тактовых сигналов

Вход: экземпляры библиотечных ячеек (признак `celldefine`)

с возможностью исключения по шаблону имени и экземпляры примитивов `buf`, `not`, `bufif0`, `bufif1`, `notif0`, `notif1`

Выход: предупреждения о буферизованных тактовых сигналах

1 **для каждого** библиотечной ячейки *c* :

2 **если** *c* имеет ровно два вывода и единственное присваивание *out* = *in* **то** пометить *c* как буфер;

3 пометить как буферы экземпляры перечисленных примитивов;

4 Проследить распространение тактовых сигналов через цепочки буферов;

5 **для каждого** последовательностного элемента, чувствительного к фронту :

6 **если** на его тактовый вход подан буферизованный тактовый сигнал **то** выдать предупреждение;

Источники единой информационной модели, используемые реализованными проверками, сведены в таблице 4. Связи ячеек Liberty с командами UPF устанавливаются при построении модели.

Таблица 4 — Источники единой информационной модели, используемые проверками

Анализ	Развернутая модель	Liberty	UPF	SDF
Проверка команд UPF при интерпретации	+		+	
Проверка стратегий (POW01, POW02)	+		+	
Обнаружение RS-защелок (LATCH02)	+	+		
Буферизованные тактовые сигналы (SIGNAL12)	+	+		
Согласованность SDF-описания с HDL-описанием	+			+

4.6 Выводы по главе

В главе представлен разработанный в работе метод интеграции сопутствующих проектных форматов (Liberty, UPF, SDF) в статический анализ RTL-описаний. Основу метода составляет единая информационная модель, которая входит во внутреннее представление и связывает узлы развернутой модели с объектной моделью библиотечных ячеек Liberty, объектной моделью подсистемы питания, получаемой полной интерпретацией UPF-описания по стандарту IEEE 1801-2018, и записями SDF-описания. Трансляция библиотечных ячеек в Verilog-представление обеспечивает единообразный межмодульный анализ схем, использующих стандартные ячейки. Для модели подсистемы питания разработан статический анализ, проверяющий необходимость, наличие и корректность стратегий изоляции и преобразования уровня напряжения по сформулированному критерию. Информация Liberty дополнительно используется реализованными детекторами RS-защелок на примитивах и библиотечных ячейках и буферизованных тактовых сигналов. Поддержка формата SDF реализована на уровне разбора и статической проверки согласованности SDF-описания с развернутой моделью HDL-описания по правилам раздела 32 стандарта IEEE 1800: конструкции SDF привязываются к путям задержек, временным проверкам, параметрам и портам развернутой модели, а рассогласования выявляются до имитационного моделирования. Единая информационная модель создает основу для дальнейшего развития анализа. Сведения библиотек Liberty (логические функции, временные дуги, атрибуты выводов, ограничения нагрузки, признаки специальных ячеек) позволяют проверять тактовые цепи и цепи сброса по функциональным признакам ячеек, контролировать ограничения нагрузочной способности до физического проектирования, уточнять анализ комбинационных петель по временным дугам и распространить извлечение конечных автоматов (глава 3) на описания после технологического отображения. Привязка SDF-описания к развернутой модели позволяет после вычисления эффективных задержек по правилам раздела 32 стандарта IEEE 1800 построить граф задержек для поиска критических путей и оценки запасов временных проверок, определить долю путей и проверок, получивших значения из SDF-описания, и уточнить анализ сбросов и пересечений доменов синхронизации пределами RECOVERY и REMOVAL. К тому же кругу направлений относятся статическая проверка сохранения состояния при отключении доменов и поддержка версий

стандарта UPF, отличных от IEEE 1801-2018. Интеграция сопутствующих форматов расширяет множество выявляемых дефектов: ошибки задания стратегий изоляции и преобразования уровня напряжения не могут быть обнаружены анализом только RTL-описания. Результаты главы соответствуют третьему положению, выносимому на защиту.

Глава 5. Программная реализация системы SVAN

На основе разработанных в предыдущих главах методов реализована расширяемая система SVAN, предназначенная для статического анализа описаний цифровой аппаратуры на языках Verilog и SystemVerilog. В настоящей главе описываются архитектура системы и ее ключевые компоненты, а также вклад работы в развитие используемых открытых инструментов. Экспериментальная оценка системы приведена в главе 6.

5.1 Архитектура системы SVAN

На верхнем уровне система статического анализа состоит из следующих компонентов (рисунок 5.1):

- фронтенд (входная часть системы, выполняющая разбор исходных описаний) — синтаксические анализаторы Verilog/SystemVerilog и подмножества VHDL, интерпретаторы SDC и UPF, транслятор Liberty и анализатор формата SDF;
- компилятор SystemVerilog — модуль, строящий по деревьям разбора АСД и развернутую модель, и взаимодействующие с ним модули трансформации, диагностики, сериализации, анализа потока данных, менеджер исходных файлов и драйвер опций;
- анализаторы: модуль анализа svan-tidy (статические детекторы правил), подсистема анализа конечных автоматов svan-fsm на основе инфраструктуры Yosys, анализатор модели подсистемы питания (UPF), а также анализатор пересечений доменов синхронизации и сброса (CDC/RDC).

Интерпретатор формата проектных ограничений SDC (Synopsys Design Constraints: определения тактовых сигналов и временные ограничения схемы) и использующий его анализатор CDC/RDC присутствуют в составе системы, однако правила корректности синхронизации вынесены за рамки настоящей работы (раздел 2.2), и их реализация далее не рассматривается.

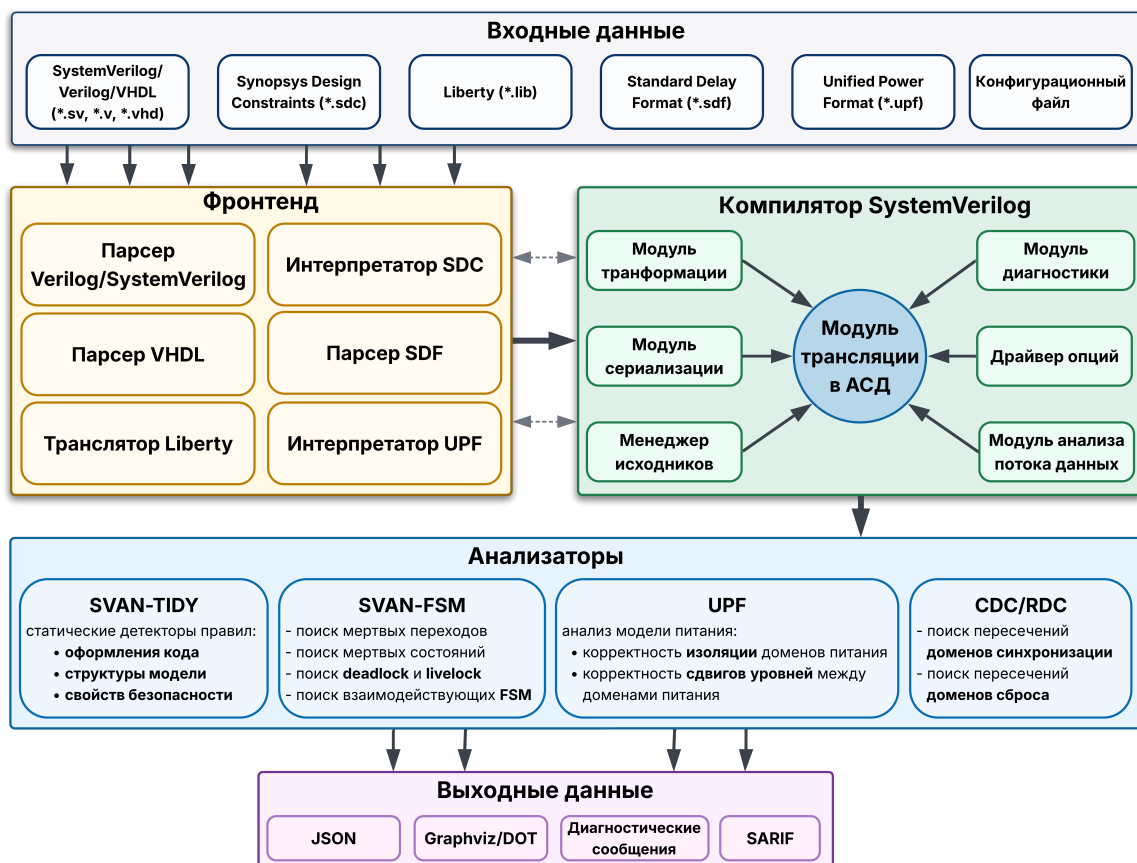


Рисунок 5.1 — Структурная схема системы статического анализа SVAN

Модуль работы с исходным текстом анализирует пути к каталогам с файлами и библиотеками и составляет одну или несколько единиц трансляции, которые компилятор (раздел 5.1.1) переводит в АСД, подаваемое на вход модулю анализа (раздел 5.1.7).

5.1.1 Компилятор SystemVerilog

Компилятор основан на программной библиотеке с открытым исходным кодом slang [62], предоставляющей компоненты для лексического анализа, проверки типов и построения развернутой модели описаний на SystemVerilog. Компилятор строит уровни внутреннего представления раздела 2.1.1: дерево разбора, АСД, предварительную и развернутую модель.

В рамках работы синтаксический анализатор slang был доработан в целях обеспечения полной поддержки стандартов SystemVerilog IEEE 1800-2017 [12]

и IEEE 1800-2023 [13]: доработки статических проверок компилятора описаны в разделе 5.2.1, добавленный анализатор формата SDF — в разделе 5.1.4.

Лексический и синтаксический анализ выполняют препроцессор, раскрывающий директивы компилятора, лексический анализатор и синтаксический анализатор, строящий дерево разбора методом рекурсивного спуска. Листья дерева хранят позицию в исходном тексте и незначимые фрагменты (пробелы, комментарии), что позволяет восстанавливать исходный текст обходом дерева. Синтаксический анализ устойчив к ошибкам в исходном тексте: при ошибке анализатор пропускает ошибочную конструкцию и продолжает разбор. Построение АСД, предварительной и развернутой модели выполняет модуль трансляции в АСД, а обход АСД — класс `ASTVisitor` с переопределением методов `handle`. Диагностика снабжается кодом и позицией в исходном тексте.

Системные задачи, регистрируемые через VPI. Тестовые окружения нередко используют системные задачи и функции, определяемые в приложениях на C/C++ через интерфейсы VPI (Verilog Procedural Interface) и DPI (Direct Programming Interface). Подпрограммы DPI объявляются в самом описании и поддерживаются компилятором штатно. Задачи, регистрируемые через VPI, в описании не объявляются, и без дополнительных сведений вызов такой задачи приводит к предупреждению о неизвестной системной задаче, а выражение вызова получает тип ошибки, что подавляет дальнейшие диагностики по нему. Для таких описаний реализован механизм извлечения имен задач из исходных текстов обвязки на C/C++: файлы, заданные опцией `--vpi`, разбираются инкрементальным синтаксическим анализатором `tree-sitter` с грамматикой языка C. Полный разбор не требуется: достаточно устойчивого к ошибкам дерева, в котором отыскиваются присваивания строковым литералом полю `tfname` структуры регистрации `s_vpi_systf_data`, поскольку именно в этом поле стандарт предписывает задавать имя задачи. Собранные имена регистрируются в таблице встроенных подпрограмм компилятора как системные задачи без ограничений на число и типы аргументов. Извлекаются только имена: сигнатуры задач по обвязке не восстанавливаются.

5.1.2 Интерпретатор UPF

Интерпретатор UPF основан на открытом интерпретаторе языка Tcl, реализованном в виде библиотеки на языке C с возможностью расширения набора команд. Интерпретатор преобразует произвольный Tcl-сценарий в последовательность команд формата, строит их каноническую форму и выполняет проверку параметров в соответствии со стандартом IEEE 1801-2018. Преобразование выполняется интерпретацией сценария с механизмом обратного вызова (callback) при встрече полной команды. Команды реализованы в виде функций на C++, принимающих список аргументов и метаданные. Для декларативной проверки аргументов используется библиотека разбора командной строки CLI11. В разделах 6 и 7 стандарта IEEE 1801-2018 описаны 63 команды задания подсистемы питания и ячеек управления питанием, и для каждой из них реализован собственный обработчик, регистрируемый в интерпретаторе Tcl средствами Tcl C API. Кроме того, реализованы обработчики функций доступа к информационной модели UPF (раздел 11 стандарта) и 39 команд запроса версии 2.0 стандарта (например, `query_power_domain`), которые возвращают сведения о созданных объектах модели и поддерживаются для совместимости с описаниями, составленными по ранним версиям стандарта (раздел 4.1.2). Всего реализовано 120 обработчиков. Пример реализации команды `create_logic_port` приведен в листинге 5.1.

Листинг 5.1: Реализация обработчика UPF-команды `create_logic_port`

```

1  int UPF_create_logic_port(ClientData data, Tcl_Interp* interp,
2                               int argc, const char* argv[]) {
3      auto* reader = static_cast<UpfReader*>(data);
4      auto [cli, exists] = reader->getCLI(argv[0]);
5      // первичная регистрация опций команды
6      if (!exists) {
7          cli->add_option("-direction")->default_str("in")
8              ->check(CLI::IsMember({"in", "out", "inout"}));
9          cli->add_option("port_name")->required();
10     }
11     try {
12         // разбор и проверка аргументов
13         cli->parse(argc, argv);
14         std::string portName, dirName;
15         cli->get_option("-direction")->results(dirName);
16         cli->get_option("port_name")->results(portName);
17         // далее следует создание объекта порта в модели подсистемы питания

```

```

18     } catch (const CLI::ParseError& e) {
19         Tcl_SetResult(interp, e.what(), TCL_VOLATILE);
20         return TCL_ERROR;
21     }
22     return TCL_OK;
23 }

```

В результате работы интерпретатор UPF строит промежуточное представление в виде набора объектов на C++, определенных в соответствии с разделом 10.6 стандарта IEEE 1801-2018.

Пакет UPF. Стандарт IEEE 1801-2018 (раздел 11.2.4) предусматривает пакет UPF на SystemVerilog, через который HDL-описание обращается к объектам модели питания. Его типы и функции используются в тестовых окружениях и моделях ячеек, поэтому без них такие описания не компилируются. Текст пакета хранится в системе как исходный текст на SystemVerilog и при наличии UPF-описания добавляется в компиляцию как обычная единица, разбираемая штатным фронтом. Этим пакет отличается от остальных встроенных сущностей компилятора (пакет std, системные функции), которые строятся программно, и может сопровождаться синхронно со стандартом без изменения кода. Пакет объявляет базовые типы, перечисление состояний питания, структуру значения питания с напряжением, дескрипторные типы объектов модели, функции навигации по модели, чтения и записи значений питания и предопределенные функции разрешения питания. При переносе текста пакета в систему в нем выявлены пять дефектов самого стандарта: три пропущенные или лишние запятые в перечислениях, использование перечисления `upfSimstateE` до его объявления и необъявленный тип `upfSupplyTypeVectorT`, определенный только в пакете для VHDL. Все пять подтверждены по тексту стандарта и исправлены в тексте пакета в системе. Поддержка пакета проверяется модульным тестом, компилирующим описание с `import UPF::*` совместно с UPF-файлом.

Анализ подсистемы питания запускается передачей пути к UPF-файлу вместе с HDL-описанием. Если описание подсистемы питания разделено на несколько файлов, передается файл, определяющий домен питания, соответствующий главному модулю HDL-описания. Переход к остальным файлам задается командой `load_upf`. При обнаружении синтаксической или семантической ошибки выводится сообщение с именем файла, номером строки и описанием ошибки. В проверке стратегий изоляции и преобразования уровня напряжения (алгоритм 4.4) для каждого граничного порта вычисляется обобщенное состояние

проверки, представленное битовой маской, каждый бит которой соответствует невыполнению одного из фильтров стратегии, и по вычисленной маске выдается диагностика по наиболее значимому невыполненному фильтру.

Связь модели питания с детекторами. Модель подсистемы питания передается детекторам модуля анализа как отдельный общий объект (раздел 5.1.7). Проверки категории power исключаются из запуска при отсутствии UPF-файла и не выполняются при ошибках его чтения. Детекторы обходят модель единым обходчиком, диспетчеризируя вызовы по виду объекта модели (домен, набор питания, стратегия, граничный порт) и по узлам иерархии HDL-описания. Правилам POW01 и POW02 соответствуют два детектора, сообщающие об ошибке для граничного порта без требуемого преобразователя уровня напряжения или ячейки изоляции. В диагностике указывается первый непрошедший фильтр стратегии (приемник, источник, направление, граница, значение фиксации, а для преобразователя уровня напряжения — также правило, порог и расположение) и позиция создавшей стратегию команды в UPF-файле. Дополнительно реализованы две проверки согласованности самой модели, выдающие предупреждения: о стратегии изоляции, не применившейся ни к одному порту, и о домене питания без функции питания или земли в первичном наборе. Эти проверки правилам каталога не соответствуют и служат для выявления ошибок в UPF-описании, не проявляющихся как нарушения стратегий.

5.1.3 Разбор библиотек Liberty

Разбор и объектное представление Liberty-описаний обеспечивает открытая библиотека ReadCells, разрабатываемая в ИСП РАН и интегрированная в систему статического анализа. Модуль обработки Liberty состоит из двух частей: синтаксический анализатор строит АСД входного описания, а построитель модели формирует по АСД промежуточное объектное представление библиотечных ячеек. Лексический анализ выполняется анализатором, генерируемым инструментом Flex, синтаксический — анализатором, генерируемым инструментом GNU Bison, с построением АСД в процессе разбора. АСД Liberty состоит из узлов двух типов: Attribute (отдельные атрибуты, задающие значения параметров объектов) и Group (наборы связанных атрибутов и вложенных групп атрибутов).

Семантический анализ проверяет корректность использования конструкций, вложенности групп и значений атрибутов в соответствии со спецификацией Liberty (версия 2017.06).

Объектная модель ячеек (раздел 4.2.1) входит в единую информационную модель и используется на разных стадиях анализа: от построения модели подсистемы питания до детекторов, анализирующих экземпляры библиотечных ячеек (раздел 4.5). В рамках интеграции ReadCells в систему разработан транслятор библиотечных ячеек в Verilog-представление, делающий экземпляры ячеек доступными межмодульному анализу.

5.1.4 Анализатор формата SDF

Анализатор формата SDF (раздел 4.2.2) реализован отдельным модулем компилятора, не зависящим от фронтенда SystemVerilog: собственные лексический и синтаксический анализаторы формата IEEE 1497-2001 [50] (рекурсивный спуск с восстановлением после ошибок и ограничением глубины вложенности), анализатор условных выражений, семантическая модель, модуль привязки и модуль запуска. Узлы синтаксического дерева SDF описаны в общей спецификации грамматики и порождаются тем же генератором, что и узлы SystemVerilog, а условия COND строятся теми же видами узлов выражений, что и условия путей specify, поэтому каноническая форма условий вычисляется единым кодом для обеих сторон. Семантическая модель неизменяема, строится один раз на файл и не зависит от HDL-описания, а модуль привязки работает над развернутой моделью (алгоритм 4.3). Файлы задаются опцией `--sdf` с необязательной областью привязки либо вызовами `$sdf_annotate` в описании с константным именем файла, аргументы которых также проверяются. При кэшировании тел одинаково параметризованных экземпляров привязка выполняется для каждого экземпляра, а повторяющиеся диагностики отбрасываются. Результат привязки выгружается в формат JSON опцией `--sdf-json`. Анализатор выполняет 28 проверок, результат которых выдается в виде 2 ошибок и 26 предупреждений, сгруппированных в 17 категорий с общим ключом `-wsdf`. Объем реализации около 5,2 тыс. строк, 94 модульных теста воспроизводят все примеры раздела 32 стандарта IEEE 1800-2023. Пример фрагмента SDF-описания приведен в листинге 5.2: конструкции

IOPATH задают задержки путей «вход — выход» ячейки в виде троек значений (минимальное : типичное : максимальное) для переднего и заднего фронтов, а конструкции TIMINGCHECK задают временные проверки (времена предустановки и удержания).

Листинг 5.2: Пример фрагмента SDF-описания

```

1 (DELAYFILE
2   (SDFVERSION "3.0")
3   (DESIGN "top")
4   (TIMESCALE 1ns)
5   (CELL (CELLTYPE "DFF")
6     (INSTANCE u_core.u_reg)
7     (DELAY (ABSOLUTE
8       (IOPATH CLK Q (0.12:0.15:0.21) (0.14:0.17:0.24))))
9     (TIMINGCHECK
10      (SETUP D (posedge CLK) (0.08))
11      (HOLD D (posedge CLK) (0.05))))))

```

5.1.5 Сериализация АСД в формат JSON

Для обработки результатов сторонними инструментами и отладки детекторов используется модуль сериализации АСД компилятора slang в формат JSON (ASTSerializer), рекурсивным обходом в глубину преобразующий каждый узел дерева вместе с его атрибутами в JSON-строку. Модуль доработан для вывода позиций узлов и имени файла. Вывод управляется опциями компилятора: заданием файла вывода, ограничением области видимости или символа иерархическим путем, включением сведений об исходном расположении узлов и развернутого описания типов.

5.1.6 Фронтенд языка VHDL

Помимо Verilog и SystemVerilog в системе реализован фронтенд подмножества языка VHDL (IEEE 1076-2019) [10] для описаний, часть модулей которых задана на VHDL. Принцип построения — общее внутреннее представление для двух

языков: конструкции VHDL транслируются в те же символы АСД, что и конструкции SystemVerilog, поэтому построение развернутой модели, анализ потоков данных и детекторы применяются к ним без существенных изменений.

Лексический анализатор единый: набор ключевых слов и правила выделения лексем выбираются по версии языка. Синтаксический анализатор VHDL наследует общий базовый класс синтаксического анализатора и использует ту же инфраструктуру восстановления после ошибок. Конкретное синтаксическое дерево для VHDL свое: его узлы и диагностики порождаются генераторами по отдельным спецификациям грамматики и сообщений, как и для SystemVerilog (раздел 5.1.1).

На уровне АСД сущности и архитектуры VHDL отображаются в определения модулей, компоненты и их экземпляры — в экземпляры, процессы — в процедурные блоки со списками чувствительности, пакеты и тела пакетов (включая отложенные константы) — в пакеты, объявления конфигураций — в привязки экземпляров к архитектурам. Формальные обобщенные параметры (generics) представляются теми же символами параметров, что и параметры модулей SystemVerilog, с вычислением актуальных значений при построении развернутой модели по карте соответствия, а операторы generate и блочные операторы — теми же символами блоков генерации и вложенных областей видимости. Подпрограммы VHDL с телами, перегрузкой по сигнатуре и разрешением перегрузки по контексту вызова, операторы циклов и возврата, индексирование, срезы и вызовы функций транслируются в общие узлы операторов и выражений. Для конструкций без аналога в SystemVerilog (перегружаемые подпрограммы, подтипы, физические и ссылочные типы, строковые и битовые строковые литералы) иерархия символов и типов расширена. Единица компиляции может содержать деревья на двух языках, поэтому экземпляр VHDL-сущности в модуле SystemVerilog разрешается как обычный экземпляр (имена портов сопоставляются без учета регистра), и детекторы правил, применимых к обоим языкам, работают на описаниях VHDL без модификации, а детекторы конструкций, специфичных для SystemVerilog, к нему не применяются. Стандартная библиотека VHDL (пакеты std и ieee) поставляется исходными текстами и разбирается тем же фронтендом при первом обращении.

Фронтенд проверен на синтетических тестах, стандартной библиотеке и открытом многоязыковом проекте (раздел 6.1). В системе типов VHDL поддержаны

перечислимые, диапазонные и агрегатные типы по IEEE 1076-2019, а также под-типы, физические и ссылочные типы.

5.1.7 Модуль анализа и интерфейс детекторов

Модуль анализа основан на инструменте статического анализа slang-tidy из библиотеки slang.

Интерфейс детекторов. Все детекторы реализуют общий интерфейс, абстрактный класс `TidyCheck`, ключевые методы которого приведены в листинге 5.3. Метод `check` принимает корневой символ развернутой модели (`RootSymbol`) и менеджер анализа (`AnalysisManager`) с результатами анализа потоков данных компилятора, метод `registerMatchers` регистрирует сопоставители узлов АСД (раздел 2.8). Имя диагностики в стиле каталога SVAN (например, `svan-port-rev-conn-buses`) формируется автоматически из категории и имени детектора методом `cmdName`. Дорогостоящие общие данные детектор не вычисляет сам, а объявляет потребность в них виртуальными методами: `needsPBInfo` (классификация процедурных блоков), `needsNetList` (цепочки соединений сигналов через порты экземпляров и карта использований сигналов, раздел 2.7.1), `needsTriFourState` (четырёхзначная модель значений), `needsContAssign` (константы непрерывных присваиваний), `needsRecursiveSubroutines` (подпрограммы с достижимой рекурсией) и `dataFlowRequest` (требуемые факты анализа потоков данных и пределы анализа). Драйвер модуля анализа вызывает эти методы после применения опций конфигурации, объединяет заявки всех активированных детекторов и строит каждую структуру не более одного раза за запуск до начала работы детекторов, предоставляя ее через защищенные методы доступа (`getPBInfo`, `getDataFlowCache`). Модель подсистемы питания передается тем же способом (раздел 5.1.2). Время построения общих данных учитывается отдельно от времени работы детекторов.

Листинг 5.3: Ключевые методы интерфейса детектора `TidyCheck`

```

1 | class TidyCheck {
2 | public:
3 |     // Точка входа анализа; возвращает true, если нарушений не найдено
4 |     virtual bool check(const ast::RootSymbol& root,
5 |                     const analysis::AnalysisManager& analysisManager) = 0;
```

```

6
7 // Имя и описание детектора, код и формат выдаваемой диагностики
8 virtual std::string name() const = 0;
9 virtual std::string description() const = 0;
10 virtual std::string shortDescription() const = 0;
11 virtual DiagCode diagCode() const = 0;
12 virtual DiagnosticSeverity diagDefaultSeverity() const = 0;
13 virtual std::string diagString() const = 0;
14
15 // Регистрация сопоставителей узлов АСД
16 // переопределяется( детектором при необходимости)
17 virtual void registerMatchers(ast_matchers::MatchFinder&) const {};
18 // Применение опций детектора из конфигурационного файла
19 virtual bool applyConfigOptions(
20     std::unordered_map<std::string, std::vector<std::string>>&);
21
22 // Объявление потребности в общих данных, строящихся драйвером
23 // однократно: классификация процедурных блоков, цепочки соединений
24 // через порты и карта использований сигналов,
25 // четырехзначная модель, константы непрерывных присваиваний,
26 // рекурсивные подпрограммы и заявка на факты анализа потоков данных
27 virtual bool needsPBInfo() const { return false; }
28 virtual bool needsNetList() const { return false; }
29 virtual bool needsTriFourState() const { return false; }
30 virtual bool needsContAssign() const { return false; }
31 virtual bool needsRecursiveSubroutines() const { return false; }
32 virtual std::optional<DataFlowRequest> dataFlowRequest() const;
33
34 protected:
35 // Доступ к общим данным строятся( при первом обращении)
36 const ProceduralBlockKindsSet* getPBInfo() const;
37 SvanDataFlowCache* getDataFlowCache() const;
38 // Накопленные диагностические предупреждения
39 Diagnostics diagnostics;
40 };

```

Детекторы регистрируются в глобальном реестре (Registry) при помощи макроса REGISTER и активируются по конфигурационному файлу, обрабатываемому компонентами TidyConfig и TidyConfigParser (категории детекторов TidyKind, состав настроек описан в разделе 2.9). Конкретный детектор, как правило, реализует обходчик АСД, наследуемый от шаблонного класса SvanASTVisitor, и переопределяет метод handle для интересующих типов узлов. Интерфейс обходчиков доработан для поддержки автоматического определения узлов-потомков и их обхода. Для упрощения разработки детекторов,

анализирующих свойства тактовых сигналов и сигналов сброса, разработаны эвристические алгоритмы поиска таких сигналов в модулях.

В качестве примера в листинге 5.4 приведена реализация обработчика экземпляров модуля для правила PORT10, техника проверки которого описана в разделе 2.3.

Листинг 5.4: Обработчик экземпляров модуля детектора правила PORT10

```

1 void handle(const InstanceSymbol& instanceSymbol) {
2     for (auto pConn : instanceSymbol.getPortConnections()) {
3         // рассматриваются только соединения, тип которых
4         // является упакованным массивом
5         const auto* expr = pConn->getExpression();
6         if (!expr) continue;
7         const auto& expType = expr->type->getCanonicalType();
8         if (!expType.isPackedArray()) continue;
9
10        // извлечение типа порта
11        const Type* t; const auto* port = &pConn->port;
12        if (port->kind == SymbolKind::Port)
13            t = &port->as<PortSymbol>().getType();
14        else if (port->kind == SymbolKind::MultiPort)
15            t = &port->as<MultiPortSymbol>().getType();
16        else continue;
17        const auto& portType = t->getCanonicalType();
18        if (!portType.isPackedArray()) continue;
19
20        // если порядок битов порта и соединения противоположен
21        // и оба сигнала не являются одноразрядными, выдается диагностика
22        const auto& pAT = portType.as<PackedArrayType>();
23        const auto& pATConn = expType.as<PackedArrayType>();
24        bool portRev = pAT.range.left > pAT.range.right;
25        bool connRev = pATConn.range.left > pATConn.range.right;
26        if (portRev != connRev && pAT.range.left != pAT.range.right
27            && pATConn.range.left != pATConn.range.right) {
28            diags.add(diag::RevConnBuses, expr->sourceRange) << pConn->port.name;
29        }
30    }
31    // рекурсивный обход тела модуля
32    instanceSymbol.body.visit(*this);
33 }

```

Реализация механизма сопоставителей. Механизм сопоставителей узлов АСД (раздел 2.8) встроен в интерфейс детекторов методом registerMatchers: детектор регистрирует выражения-сопоставители вызовом AddMatcher объекта

MatchFinder, а захваченные узлы получает в методе check после этапа сопоставления. Узлы АСД хранятся в единых контейнерах со стиранием типов, а единственный полный обход дерева (MatchFinder::CollectAllNodes) заполняет кэш и, при необходимости, граф узлов (NodeGraph) для сопоставителей отношений «предок — потомок» (раздел 2.8.3). Детектор без таких отношений отказывается от графа вызовом MatchFinder::SkipNodeGraph(), а для проверок, которым необходим каждый экземпляр модуля, предусмотрен режим MatchFinder::RequirePerInstanceBodies(). Результаты доступны в двух видах (раздел 2.8.2): Result (объединенные привязки по идентификаторам bind) и Results (привязки каждого совпадения). Шаблон MatchedNodes<T> дополнительно обеспечивает типизированную выборку узлов. Всего на сопоставителях реализовано 18 детекторов модуля анализа.

Примеры детекторов на сопоставителях. Различие двух способов реализации иллюстрирует детектор правила DESIGN04, выявляющий блоки always без управления событиями (отсутствие события в таком блоке ведет к расхождению поведения при имитационном моделировании и синтезе). В варианте на обходчиках (листинг 5.5) требуются два класса-обходчика: основной отбирает процедурные блоки вида always, а вспомогательный выполняет вторичный обход тела блока в поисках оператора с временным управлением или утверждения. Продолжение и прерывание обхода управляются вручную.

Листинг 5.5: Детектор правила DESIGN04 на обходчиках (фрагмент; конструкторы опущены)

```

1 // вторичный обходчик: поиск оператора с временным управлением
2 // или утверждения в теле блока
3 class TimedStatementVisitor
4     : public SvanASTVisitor<TimedStatementVisitor, VisitFlags::AllGood> {
5 public:
6     template<class T>
7         requires(std::is_same_v<T, TimedStatement> ||
8                 std::is_same_v<T, ConcurrentAssertionStatement> ||
9                 std::is_same_v<T, ImmediateAssertionStatement>)
10        void handle(const T&) {
11            found = true;
12            terminate();           // досрочное прерывание обхода
13        }
14        bool found = false;
15 };
16
17 // основной обходчик: отбор блоков вида always

```

```

18 class MainVisitor : public SvanASTVisitor<MainVisitor, VisitFlags::AllGood> {
19 public:
20     void handle(const ProceduralBlockSymbol& pbs) {
21         if (pbs.procedureKind != ProceduralBlockKind::Always)
22             return;
23         if (pbs.getBody().bad())
24             return;
25         TimedStatementVisitor tsVis(diags); // вторичный обход тела
26         pbs.visit(tsVis);
27         if (!tsVis.found)
28             diags.add(diag::NoSensitivityList, pbs.location);
29     }
30 };
31
32 bool check(const ast::RootSymbol& root, const AnalysisManager&) override {
33     MainVisitor visitor(diagnostics);
34     root.visit(visitor);
35     return diagnostics.empty();
36 }

```

В варианте на сопоставителях (листинг 5.6) то же условие записывается одним выражением-сопоставителем, а захваченные блоки детектор получает типизированной выборкой `MatchFinder::MatchedNodes` и сразу формирует по ним диагностики. Обход дерева выполняется механизмом однократно и совместно с остальными проверками.

Листинг 5.6: Детектор правила DESIGN04 на сопоставителях (фрагмент)

```

1 // предикат: процедурный блок вида always с корректным телом
2 auto isAlwaysBlockWithGoodBody = [](const ProceduralBlockSymbol* block,
3                                     BoundNodesTreeBuilder&) {
4     return block->procedureKind == ProceduralBlockKind::Always &&
5            !block->getBody().bad();
6 };
7
8 void registerMatchers(MatchFinder& finder) const override {
9     finder.AddMatcher(
10         ProceduralBlockSymbol(
11             isAlwaysBlockWithGoodBody,
12             unless(hasDescendant(anyOf(TimedStatement(),
13                                         ConcurrentAssertionStatement(),
14                                         ImmediateAssertionStatement()))))
15         .bind("block"),
16         this->name());
17 }
18
19 bool check(const ast::RootSymbol&, const AnalysisManager&) override {

```

```

20     for (const auto* block :
21         MatchFinder::MatchedNodes<ProceduralBlockSymbol>(&this->name(),
22                                                         "block"))
23         diagnostics.add(diag::NoSensitivityList, block->location);
24     return diagnostics.empty();
25 }

```

Приведем также сопоставители двух других детекторов. Детектор правила SYNTH10 (использование операторов сравнения `==` и `!=`) отбирает бинарные выражения по виду операции готовым параметризуемым предикатом `hasAnyOperator`. Отношения «предок — потомок» ему не нужны, поэтому он объявляет отказ от построения графа узлов (листинг 5.7).

Листинг 5.7: Регистрация сопоставителей детектора правила SYNTH10

```

1 void registerMatchers(MatchFinder& finder) const override {
2     finder.SkipNodeGraph(); // проверка не использует граф узлов
3     finder.AddMatcher(BinaryExpression(
4         hasAnyOperator(BinaryOperator::CaseEquality,
5                         BinaryOperator::CaseInequality))
6         .bind("caseEqual"),
7         this->name());
8 }

```

Детектор правила PORT10 (тот же, что в листинге 5.4) на сопоставителях сводится к отбору экземпляров модулей предикатом, проверяющим наличие подключения с обратным порядком разрядов (листинг 5.8). Ручной разбор видов портов и явное продолжение обхода, характерные для варианта на обходчиках, здесь отсутствуют.

Листинг 5.8: Детектор правила PORT10 на сопоставителях (фрагмент; функция `isReverseConnected` сравнивает направления диапазонов разрядов, ср. листинг 5.4)

```

1 // предикат: у экземпляра есть подключение с обратным порядком разрядов
2 auto hasReverseConnectedBus = [](const InstanceSymbol* instance,
3                                  BoundNodesTreeBuilder&) {
4     for (const auto* connection : instance->getPortConnections()) {
5         const auto* connExpr = connection->getExpression();
6         if (connExpr && isReverseConnected(connection->port, *connExpr))
7             return true;
8     }
9     return false;
10 };
11
12 void registerMatchers(MatchFinder& finder) const override {

```

```

13     finder.SkipNodeGraph();           // проверка не использует граф узлов
14     finder.AddMatcher(InstanceSymbol(hasReverseConnectedBus).bind("instance"),
15                       this->name());
16 }

```

Менеджер интервальных ограничений. Для анализа путей в процедурных блоках (раздел 2.4) реализован менеджер ограничений `ConstraintManager`: ограничение на значения символа представлено классом `RangeArray` (упорядоченный список непересекающихся отрезков `SVIntValueRange`) с операциями `addRange`, `subtractRange`, `intersect` и `unite`.

Ключевые обработчики детектора правила `ASSIGN04` приведены в листинге 5.9. Обработчик присваивания (`AssignmentExpression`) запоминает первое неблокирующее присваивание последовательности в поле `FirstNonBlocking` и выдает диагностику на блокирующее присваивание, если задано это поле или поле `PossibleNonBlocking` либо непусто отображение `ConstrainedPaths`. Обработчик условного оператора (`ConditionalStatement`) вычисляет ограничения ветвей `if/else` через `ConstraintManager`, передает состояние в обходчики ветвей методом `inheritPaths` (без записи для выражения своего условия), возвращает их сведения методом `mergePaths` и объединяет ограничения ветвей, содержащих неблокирующие присваивания, в отображении `ConstrainedPaths`. Теми же методами состояние передается в ветви оператора `case` и тело цикла.

Листинг 5.9: Обработчики детектора правила `ASSIGN04` (упрощенно)

```

1  bool handle(const AssignmentExpression& Assign) {
2      // запоминается первое неблокирующее присваивание на пути
3      if (!FirstNonBlocking && Assign.isNonBlocking())
4          FirstNonBlocking = &Assign;
5      // блокирующее присваивание, которому на какомлибо пути
6      // могло предшествовать неблокирующее: нарушение
7      if (Assign.isBlocking() && (FirstNonBlocking || PossibleNonBlocking ||
8                                  !ConstrainedPaths.empty()))
9          diags.add(diag::AlwaysCombNonblocking, Assign.sourceRange());
10     return false;
11 }
12
13 bool handle(const ConditionalStatement& Cond) {
14     // вычисление ограничений, порождаемых условием оператора if
15     ConstraintManager Manager(Context);
16     auto [Ref, IfConstraint] = Manager.addConstraint(Cond.conditions.front().expr);
17     SimpleConstraint ElseConstraint = IfConstraint.getComplementConstraint();
18     // независимый обход ветвей if и else с унаследованным состоянием
19     // (IfFirst, ElseFirst: N[e] или FirstNonBlocking)

```

```

20 AssignmentVisitor visTrue(diags, Context, IfFirst);
21 visTrue.inheritPaths(*this, Ref);
22 Cond.ifTrue.visit(visTrue);
23 AssignmentVisitor visFalse(diags, Context, ElseFirst);
24 visFalse.inheritPaths(*this, Ref);
25 if (Cond.ifFalse) Cond.ifFalse->visit(visFalse);
26 // сведения ветвей возвращаются в объемлющую последовательность
27 const bool tracked = Ref && IfConstraint.isSimple;
28 mergePaths(visTrue, tracked ? nullptr : visTrue.getFirstNonBlocking());
29 mergePaths(visFalse, tracked ? nullptr : visFalse.getFirstNonBlocking());
30 // объединение ограничений путей с неблокирующими присваиваниями
31 if (tracked && visTrue.getFirstNonBlocking())
32     ConstrainedPaths[*Ref].unite(IfConstraint);
33 return false;
34 }

```

Реализация анализа потоков данных. Анализ потоков данных (раздел 2.7) реализован иерархией шаблонных классов: базовые уровни `AbstractFlowAnalysis` и `DataFlowAnalysis` принадлежат компилятору `slang`, а разработанные в работе уровни `SvanDataFlowAnalysis` и `SvanTrackingDataFlowAnalysis` добавляют к состоянию значения и диапазоны значений сигналов и прослеживание символов через вызовы подпрограмм. Множества разрядов представлены интервальными отображениями (`IntervalMap`), альтернативные пути — вектором отображений «сигнал — диапазоны значений». Общий кэш результатов (`SvanDataFlowCache`) создается драйвером по заявкам `dataFlowRequest` активированных детекторов и заполняется до их запуска. Кэш потокобезопасен: детекторы выполняются параллельно, поиск записи выполняется под блокировкой, а сам анализ — однократно под собственным флагом записи. Отдельные кэши хранят результаты анализа подпрограмм по набору прослеживаемых аргументов и по диапазонам фактических аргументов. Агрегирование источников (драйверов) сигналов по всей иерархии развернутой модели выполняется классом компилятора `DriverTracker`, доработанным в рамках работы и используемым детекторами правил о неподключенных портах и портах без источника, множественных источниках и тактовых сигналах в последовательностной логике. Результаты базового анализа доступны детекторам также через менеджер анализа (`AnalysisManager`), передаваемый в метод `check`.

5.1.8 Реализация подсистемы анализа конечных автоматов

Подсистема анализа конечных автоматов реализована в виде конвейера из трех взаимодействующих компонентов:

- фронтенд *yosys-slang* — транслятор АСД библиотеки *slang* во внутреннее представление RTLIL инфраструктуры Yosys, доработанный для сохранения сведений об исходном расположении конструкций (атрибут `src`, раздел 3.5);
- модифицированные проходы Yosys `fsm_detect` и `fsm_extract` — обнаружение и извлечение конечных автоматов из представления RTLIL с доработками раздела 3.3;
- модуль *svan-fsm* — графовый анализатор на основе Boost Graph Library, выполняющий графовый анализ раздела 3.4 (сильно связанные компоненты вычисляются штатной реализацией алгоритма Тарьяна) и обнаружение взаимодействующих автоматов (раздел 3.6) с экспортом графов в формат DOT.

5.2 Вклад в развитие открытых инструментов

Реализация системы потребовала существенной доработки используемых открытых инструментов: компилятора SystemVerilog *slang* [62], фронтенда *yosys-slang* [84] и инфраструктуры логического синтеза Yosys [70]. Существенная часть внесенных изменений принята в основные ветки соответствующих проектов и составляют часть личного вклада автора. Доработки обеспечили полноту поддержки стандартов SystemVerilog IEEE 1800-2017 и IEEE 1800-2023, корректность построения внутреннего представления и применимость инструментов к задачам статического анализа.

5.2.1 Доработки компилятора SystemVerilog

Доработки компилятора slang сгруппированы по следующим направлениям.

Утверждения и последовательности (SVA). Реализована проверка невырожденности (nondegeneracy) последовательностей в соответствии с разделом стандарта о свойствах и последовательностях: последовательность считается вырожденной, если она допускает только пустые сопоставления либо вовсе не допускает сопоставлений, что как правило свидетельствует об ошибке. Разработан механизм разрешения тактовых событий утверждений (clock resolution, разделы 16.13 и 16.16 стандарта): тактовое событие представляется структурой `ClockingEvent`, хранящей его в двух формах (фронт и сигналы либо строковое представление сложного выражения), а вложенные обходчики (`ClockResolutionVisitor`, `ClockingBlockVisitor`, `ConcurrentAssertionVisitor`, `SequenceVisitor`) выявляют девять классов ошибок: от свойства без привязанного тактового события и явного управления событиями в блоке `clocking` до несоответствия тактовых событий операндов бинарных операций над последовательностями и конкатенаций с задержкой. Кроме того, исправлены ошибки рекурсивной проверки отрицания свойств, доработана проверка вызовов функций в отложенных утверждениях (deferred assertions), проверка методов последовательностей и функций выборочных значений (sampled value functions) в условии `disable iff` с учетом явного тактирования, а также проверка совпадения локальных переменных с формальными аргументами.

Пользовательские примитивы и таблицы истинности. Добавлена проверка покрытия комбинаций входов и фронтов в таблице состояний пользовательского примитива (UDP), исправлены ошибки обработки перекрытия строк таблицы UDP, а также ошибки разбора программируемых логических матриц (PLA) и предельных значений конструкции `PATHPULSE` в блоках `specify`.

Псевдонимы цепей и контроль типов. Реализована проверка дублирования разрядов в псевдонимах цепей (alias): анализатор отслеживает поразрядное соответствие связанных псевдонимом цепей и выявляет некорректное многократное связывание одних и тех же разрядов. Добавлена проверка несовместимости типов цепей при неявном именованном соединении портов.

Виртуальные интерфейсы и инфраструктура анализа. Исправлены сравнение виртуальных интерфейсов, бесконечная рекурсия обходчика тактовых

сигналов (ClockVisitor) и ряд ошибок инфраструктуры анализа (определение анализируемой области видимости, обход кэшированных тел). Доработки синтаксического анализатора включили запрет вложенной спецификации атрибутов, разбор ограничений без указания времени, поддержку потоковых операторов в ветвях условной операции и опцию версии ключевых слов.

Инструмент slang-tidy. В инструмент slang-tidy внесены исправления, устраняющие дублирование диагностик, аварийные завершения отдельных проверок и ошибки сопоставления имен проверок и их конфигурации.

5.2.2 Доработки фронтенда yosys-slang и инфраструктуры Yosys

Для построения корректного внутреннего представления RTLIL, используемого при извлечении конечных автоматов (глава 3), доработан фронтенд yosys-slang: реализована поддержка циклов foreach, обработка пользовательских примитивов как «черных ящиков», трансляция системной функции `$clog2`, поддержка одноразрядных выборок для асинхронных триггеров, неименованных соединений портов «черных ящиков», корректная индексация неупакованных массивов с возрастающим порядком индексов, а также обнаружение выходных портов интерфейсов без источника при сохранении иерархии.

В инфраструктуру логического синтеза Yosys внесены доработки, повышающие корректность построения сети логических элементов: поддержка квалификатора времени жизни `automatic` для процедурных переменных и разделение асинхронных триггеров (`aldff`) для устранения логических петель по данным. Перечисленные доработки обеспечивают корректную работу конвейера извлечения конечных автоматов на открытых RTL-проектах.

5.3 Технические средства, вызов и данные

Система реализована на языке C++ 20 с системой сборки CMake. Для тестирования используются библиотека Catch2 и набор синтетических тестов sv-

tests. Для работы требуется операционная система на основе Linux (Ubuntu 20.04 или выше, Astra Linux 1.8 или выше) и пакеты `libc++`, `libstdc++`, `zlib`.

На вход системы поступают исходные тексты на языках Verilog, SystemVerilog и VHDL, библиотеки ячеек Liberty, UPF- и SDF-описания, файлы проектных ограничений SDC (для CDC/RDC-анализатора вне рамок работы) и конфигурационный файл с опциями компилятора и модуля анализа (рисунок 5.1). При отсутствии того или иного типа спецификаций соответствующий вид анализа не выполняется. Выходными данными являются диагностические предупреждения (в текстовом формате или формате SARIF), сериализованное в JSON абстрактное синтаксическое дерево, а также графы конечных автоматов в формате DOT. Для описаний на Verilog и SystemVerilog анализ выполняется для синтаксических конструкций в рамках стандартов IEEE 1800-2017 и IEEE 1800-2023, для описаний на VHDL — для поддерживаемого подмножества стандарта IEEE 1076-2019 (раздел 5.1.6). Поддерживаются все разделы стандартов Verilog IEEE 1364-2005 (разделы 3–19 и 28), SystemVerilog IEEE 1800-2017 (разделы 3–34) и UVM IEEE 1800.2-2020 (разделы 5–19), описывающие конструкции языка и верификационного окружения. Более ранние версии стандартов поддерживаются в силу преемственности. Системные задачи и функции, определяемые через программные интерфейсы VPI и DPI (разделы 35–41 IEEE 1800-2017), разрешаются при анализе (раздел 5.1.1).

5.4 Выводы по главе

В главе описана программная реализация системы статического анализа SVAN, объединяющей компилятор SystemVerilog на основе библиотеки `slang` с синтаксическим анализатором подмножества языка VHDL и средствами разбора сопутствующих форматов (UPF, Liberty, SDF), расширяемый модуль анализа с общим интерфейсом детекторов и механизмом сопоставителей узлов АСД (раздел 2.8), а также подсистему анализа конечных автоматов на основе инфраструктуры Yosys. Модуль анализа содержит 89 детекторов, которые вместе с диагностиками компилятора, средствами анализа конечных автоматов и анализатором модели подсистемы питания проверяют 116 из 118 правил набора (раздел 6.2). Существенная часть доработок используемых открытых инструмен-

тов (компилятора slang, фронтенда yosys-slang и инфраструктуры Yosys) принята в основные ветки соответствующих проектов. Результаты главы соответствуют четвертому положению, выносимому на защиту.

Глава 6. Экспериментальная оценка системы SVAN

В настоящей главе приводится методика экспериментальной оценки реализованной системы и результаты тестирования: точность детекторов правил и средств анализа конечных автоматов на открытых проектах цифровых СБИС, оценка анализа с привлечением сопутствующих форматов, оценка механизма сопоставителей узлов АСД, сравнение с коммерческим инструментом SpyGlass и с открытыми средствами статического анализа, а также сведения о внедрении результатов.

6.1 Методика экспериментальной оценки

Для оценки качества реализованных детекторов использовались синтетические тесты и проекты цифровых СБИС с открытым исходным кодом. Тестовый набор составили десять открытых проектов: аппаратная реализация симметричного блочного шифра AES, процессорные ядра RISC-V (PicoRV32, BlackParrot, CVA6, SCR1, Ibex, VeeR EH1, VeeR EL2, OpenC910) и аппаратный корень доверия OpenTitan. Дополнительно в качестве нагрузочного теста масштабируемости использовался проект NVDLA (аппаратный ускоритель глубокого обучения объемом около 1,5 млн строк кода на Verilog/SystemVerilog), анализ которого выполнен успешно.

Качество анализа оценивалось ручной разметкой срабатываний и вычислением точности как отношения числа истинных срабатываний (True Positive, TP) к сумме истинных и ложных (False Positive, FP) размеченных срабатываний:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Если проверка (детектор правила, диагностика компилятора, вид диагностики автоматов или правило сравниваемого средства) дала на проекте не более 100 срабатываний, размечались все ее срабатывания, иначе размечались не менее 100 срабатываний из разных файлов проекта. Сообщения об одном месте исходного текста (например, о разрядах одного сигнала или об экземплярах одного автомата) размечались один раз. Диагностики взаимодействующих автоматов

размечены по выборке из 100 различных пар (раздел 6.6.4). Точность инструмента на проекте оценивалась переносом доли ложных срабатываний каждой проверки, определенной по разметке этого проекта, на все срабатывания этой проверки на проекте. Для оценок по выборке приводится 95 %-й доверительный интервал Уилсона для биномиальной доли [85].

В терминах раздела 2.1.2 истинным срабатываниям детектора соответствуют элементы множества $\hat{V}_r(P) \cap V_r(P)$, а ложным соответствуют элементы множества $\hat{V}_r(P) \setminus V_r(P)$. Полнота (recall, доля покрытия множества $V_r(P)$) на открытых проектах непосредственно не измерима, поскольку множество $V_r(P)$ неизвестно. Она проверялась на синтетических тестах, где для каждого правила подготовлены описания, нарушающие его и соответствующие ему. В сравнительных исследованиях (разделы 6.6 и 6.7) дополнительно оценивается *взаимная полнота*, то есть доля истинных находок одного средства, обнаруженных другим. *Находкой* называется место исходного текста, на которое указывает срабатывание. Находка считается обнаруженной другим средством, если оно сообщило о нарушении сопоставленного правила в том же месте. Число истинных находок оценивается так же, как точность, переносом доли истинных среди размеченных находок проекта.

Правила сравниваемых средств близки к правилам набора (приложение А), но не совпадают с ними дословно, поэтому сообщения всех средств размечались по определениям правил набора. Сообщение сравниваемого средства считалось истинным, если указанный в нем фрагмент исходного текста нарушает сопоставленное правило набора. Например, правило W415a инструмента SpyGlass сообщает и о значении по умолчанию, которое затем переопределяется в условном операторе. Сопоставленное ему правило ASSIGN05 о повторном присваивании такой прием нарушением не считает, поэтому такие сообщения W415a размечались как ложные.

Составлено 3134 синтетических и регрессионных теста для детекторов правил (795 позитивных и 2339 негативных), 49 тестов для анализа конечных автоматов и 830 тестов для подсистемы интерпретации UPF (раздел 6.4). Компилятор SystemVerilog дополнительно тестировался на наборе sv-tests и на пяти миллионах позитивных и негативных тестов, автоматически сгенерированных по грамматике языка [86]. Фронтенд языка VHDL (раздел 5.1.6) проверялся на 233 синтетических тестах, на пакетах стандартных библиотек STD и IEEE и на открытом проекте GuitarPedalBoard [87], модули которого описаны на VHDL и Verilog. Для этого

проекта построено единое АСД, детекторы правил на его VHDL-части срабатываний не дали.

Эксперименты выполнены на двух стендах. На рабочей станции (процессор Intel Core i7-14700, 20 ядер, 31 ГБ оперативной памяти, ОС Ubuntu 24.04.3) выполнены сравнение с открытыми средствами (раздел 6.7), оценка механизма сопоставителей (раздел 6.5) и измерения времени анализа автоматов (раздел 6.3). На сервере (восемь процессорных ядер 2,6 ГГц, 15 ГБ оперативной памяти, ОС CentOS Linux 7) выполнены оценка детекторов правил (раздел 6.2) и сравнение с инструментом SpyGlass (раздел 6.6), поскольку имеющаяся установка SpyGlass работает только под управлением ОС CentOS, а время и память обоих инструментов требовалось измерять в одинаковых условиях. Время и пиковая резидентная память измерялись утилитой `/usr/bin/time`. Детекторы выполнялись последовательно в одном потоке, режим параллельного выполнения (раздел 5.1.7) не использовался.

6.2 Результаты анализа детекторов правил

В рамках работы реализована проверка 116 из 118 правил составленного списка (приложение А, без двух правил категории `sync`, раздел 2.2): 89 правил проверяются 89 детекторами модуля анализа, 22 правила проверяются диагностиками компилятора, 3 правила категории `fsm` проверяются средствами анализа конечных автоматов, а 2 правила модели подсистемы питания проверяются детекторами модели питания (раздел 6.4). Часть проверок опирается на диагностики компилятора, доработанные в рамках работы (раздел 5.2.1).

На проектах с открытым исходным кодом детекторы оценивались на полном наборе исходных текстов каждого проекта, включая тестовое окружение, с полным набором проверок. Он состоит из детекторов модуля анализа, которые проверяют 89 правил (например, правило LATCH01 о непреднамеренных защелках), и диагностик компилятора, которые проверяют еще 22 правила (например, правило WIDTH01 о неявном изменении разрядности при присваивании проверяется диагностиками `-wwidth-expand` и `-wwidth-trunc`). Те же запуски использованы в сравнении с инструментом SpyGlass (раздел 6.6). Срабатывания дали 42 детектора и 13 диагностик компилятора, всего 53 486 срабатываний.

Из них 40 322 приходится на шумное правило PORT04 («порты должны быть регистровыми»), то есть правило с большим числом срабатываний, которое отражает практику проектирования, а не дефект. Для этого правила размечены 100 срабатываний из разных проектов и файлов, все они истинны, и далее его срабатывания не рассматриваются: чтобы шумное правило не завышало оценку, таблица 5 (итоги разметки по методике раздела 6.1) и все последующие подсчеты приведены по остальным 13 164 срабатываниям 41 детектора и 13 диагностик. Из них размечены 6846 (52 %), в том числе все срабатывания на проектах AES, PicoRV32, SCR1, VeeR EH1, VeeR EL2 и Ibex.

Из результатов (таблица 5) следует, что разработанные детекторы демонстрируют высокую точность (от 90,5 до 100 % по проектам, 96,3 % в целом) и приемлемую ресурсоемкость: анализ крупнейших проектов занимает от 8 до 24 мин при пиковой памяти до 12,5 ГБ. Зависимость времени анализа от количества строк не является линейной, так как время анализа детектора больше зависит от сложности кода схемы (глубины иерархии модулей, плотности межсоединений), чем от его объема. Пиковая память на проекте SCR1 (4,7 ГБ при 16,9 тыс. строк) определяется одним детектором правила ASSIGN02: при отдельном запуске на рабочей станции он требует 4,7 ГБ, а каждая из остальных проверок требует не более 0,3 ГБ.

Все 305 размеченных ложных срабатываний относятся к пяти проверкам. Детектор правила ASSIGN05 (121 ложное срабатывание из 310 размеченных) сообщает о повторных присваиваниях, которые правило нарушением не считает. Это переопределение в условной ветви значения по умолчанию, присвоенного в начале блока (84 срабатывания), присваивания во взаимоисключающих ветвях (15), присваивания разным элементам массива или полям структуры на разных итерациях цикла (20) и присваивания локальной переменной, объявленной в теле цикла (2). Диагностика компилятора -warith-op-mismatch (правило WIDTH04, 139 из 462) сообщает об операндах одинаковой разрядности, различающихся только знаковостью (126 случаев, например `int` и `int unsigned`, `logic` и `bit signed` той же разрядности), и об операнде перечислимого типа, разрядность которого равна разрядности другого операнда (13). Детектор правила SYNTH13 относит синтезируемую системную функцию `$onehot0` к несинтезируемым конструкциям (29 срабатываний из 1127, размечены все). Детектор правила SIGNAL17 (9 из 723) не считает чтением обращение к массиву по динамическому индексу и чтение элемента массива в цикле (четыре срабатывания), а также чте-

ние выходного аргумента функции в вызывающей задаче и чтение через модпорт интерфейса (пять). Детектор правила WIDTH07 (7 из 592) сообщает об усечении при присваивании литерала 'bх без размера (три срабатывания), значения типа chandle (два) и результата унарного минуса, разрядность которого равна разрядности приемника (два).

Таблица 5 — Срабатывания детекторов и диагностик компилятора системы SVAN на проектах с открытым исходным кодом, без правила PORT04

Проект	Срабатываний	Размечено	FP	Оценка точности	Время	Память	Строк, тыс.
AES	4	4	0	100 %	0,6 с	35 МБ	4,6
PicoRV32	222	222	21	90,5 %	12 с	52 МБ	7,8
SCR1	377	377	9	97,6 %	17 с	4,7 ГБ	16,9
VeeR EH1	833	833	38	95,4 %	27 с	273 МБ	21,2
VeeR EL2	952	952	19	98,0 %	2,2 мин	1,7 ГБ	51,5
Ibex	234	234	1	99,6 %	22 с	438 МБ	85,0
BlackParrot	3980	977	13	99,7 %	25 с	782 МБ	148,2
CVA6	2543	1859	103	91,6 %	12,7 мин	7,7 ГБ	270,5
OpenC910	2132	726	2	99,9 %	8,3 мин	4,0 ГБ	406,0
OpenTitan	1887	662	99	91,1 %	24,2 мин	12,5 ГБ	838,9
Всего	13 164	6846	305	96,3 %			1850,6

Проекты упорядочены по числу строк. Оценка точности по проекту вычисляется по срабатываниям как $1 - \sum_d n_d f_d / \sum_d n_d$, где n_d обозначает число срабатываний проверки d в проекте, f_d обозначает долю ложных среди размеченных срабатываний этой проверки в проекте. Столбец «Строк» содержит объем исходных текстов проекта.

Число срабатываний в таблице 5 не следует отождествлять с числом дефектов. Согласно определениям раздела 2.1.2, срабатывание фиксирует нарушение проверяемого свойства, которое может быть отклонением от практики проектирования или стиля кодирования, конструкцией тестового окружения либо намеренной конструкцией. Приведенная точность характеризует корректность установления факта нарушения. Состав истинных срабатываний (следствия конфигурации, намеренные конструкции, отклонения от стиля, функциональные дефекты) разобран в сравнительных исследованиях (разделы 6.6 и 6.7).

Распределение срабатываний по детекторам и диагностикам на примере проекта CVA6, на котором размечено наибольшее число срабатываний, приведено в таблице 6. Ложные срабатывания на этом проекте дали детекторы правил ASSIGN05, SYNTH13, SIGNAL17 и WIDTH07 и диагностика -warith-or-mismatch, остальные проверки ложных срабатываний не дали.

Таблица 6 — Срабатывания детекторов и диагностик компилятора на проекте CVA6 (без правила PORT04)

Правило SVAN	Проверка	Срабатываний	Размечено	FP	Точность
SYNTH13	детектор	443	443	9	98,0 %
WIDTH04	-warith-op-mismatch	308	100	42	58,0 %
WIDTH07	детектор	295	102	1	99,0 %
WIDTH01	-wwidth-expand	203	100	0	100 %
SIGNAL17	детектор	180	100	2	98,0 %
ASSIGN05	детектор	139	100	49	51,0 %
CASE04	детектор	138	100	0	100 %
WIDTH01	-wwidth-trunc	123	100	0	100 %
SIGNAL03	детектор	90	90	0	100 %
ASSIGN06	-wsign-conversion	79	79	0	100 %
ASSIGN06	-wsign-compare	76	76	0	100 %
WIDTH05	детектор	60	60	0	100 %
PORT01	-wunused-port	51	51	0	100 %
ASSIGN01	детектор	32	32	0	100 %
SIGNAL07	детектор	30	30	0	100 %
CASE02	детектор	28	28	0	100 %
ASSIGN03	детектор	26	26	0	100 %
TIMING01	детектор	25	25	0	100 %
CASE03	детектор	23	23	0	100 %
CASE01	детектор	22	22	0	100 %
DESIGN02	детектор	20	20	0	100 %
SYNTH10	детектор	20	20	0	100 %
LATCH01	детектор	19	19	0	100 %
SIGNAL18, TYPE04	-wint-bool-conv	18	18	0	100 %
VARIABLE04	детектор	18	18	0	100 %
PORT12, VARIABLE06	-wunassigned-variable	10	10	0	100 %
Остальные проверки (18)*		67	67	0	100 %
Всего		2543	1859	103	91,6 %

* Детекторы правил ARITH01, ASSIGN02, CASE07, CASE08, LOOP02, PORT03, PORT06, PORT13, RANGE01, RESET01, RESET08, SYNTH07, SYNTH08 и WIDTH06 и диагностики -wport-width-trunc и -wport-width-expand (WIDTH02), -wundriven-port (SIGNAL16) и -wconstant-conversion (WIDTH08), у каждой менее 10 срабатываний.

Разработанные детекторы позволили обнаружить ряд ошибок в проектах с открытым исходным кодом. Разработчикам сообщено о 24 нарушениях, вручную проверенных по исходному коду: в основном о функциональных дефектах, а также об отклонениях описаний от стандартов (приложение Б, таблица 24). Раз-

работчики подтвердили **13** нарушений, из них **10** уже исправлены в проектах, три ожидают исправления. По восьми обращениям ожидается ответ разработчиков. Среди исправленных дефектов:

- выход за границы битового вектора (правило RANGE01) в пакете конфигурации списков управления доступом (RACL) проекта OpenTitan;
- два выхода за границы массива (правило RANGE01) в ядре CVA6: в модуле регистров управления обращение к элементу `index+1` массива конфигурации защиты памяти для последнего допустимого индекса и в модуле трассировки тестового окружения цикл по 100 элементам массивов из 11 элементов;
- недостижимый код в декодере команд ядра Ibex (правило DESIGN02);
- нарушение правила LATCH01 («защелки не должны выводиться непреднамеренно») в модуле трассировки команд ядра CVA6: выходному сигналу не присваивается значение по умолчанию перед условным оператором;
- нарушение правила CASE03 («разрядность метки оператора case должна соответствовать разрядности селектора») в проекте BlackParrot.

6.3 Результаты анализа конечных автоматов

Средства извлечения и анализа конечных автоматов протестированы на 49 синтетических и регрессионных тестах (отдельные тесты содержат по несколько автоматов) и на 10 проектах с открытым исходным кодом. Анализатор продемонстрировал корректную работу по всем четырем анализируемым типам дефектов. Результаты приведены в таблице 7: число извлеченных экземпляров автоматов и число диагностик каждого вида.

Ручной анализ результатов показал, что обнаруживаемые сценарии делятся на три категории. Это *конфигурационные артефакты* (большинство мертвых переходов и недостижимых состояний возникает из параметризованных модулей, экземпляры которых созданы с конкретными значениями параметров), *преднамеренное терминальное поведение* (часть взаимоблокировок соответствует многофазным автоматам, не возвращающимся в начальное состояние) и *потенциальные ошибки* (меньшая часть находок, подтверждение которых требует

обсуждения с разработчиками описания схемы). Размеченные мертвые переходы в основном обусловлены значениями параметров и константами конфигурации (в том числе режимы кэша BlackParrot, выключенные параметрами, и векторные перестановки OpenC910, запрещенные константой в декодере), остальные обусловлены намеренными терминальными состояниями. Причиной мертвых переходов в OpenC910 служит провод с постоянным значением. Отличить намеренную настройку от ошибки в этом случае можно лишь по замыслу разработчиков, поэтому о выявленном мертвом коде сообщено разработчикам. Инструмент предоставляет аннотированные графы в формате DOT, облегчающие разбор находок.

Таблица 7 — Результаты извлечения и анализа конечных автоматов на проектах с открытым исходным кодом (число диагностик), сквозное время анализа и пиковая память

Проект	Автоматов	Мертвые переходы	Недостижимые состояния	Тупиковые состояния	Взаимо-блокировки	Время	Память
AES	4	0	0	0	0	0,2 с	65 МБ
PicoRV32	2	0	1	0	0	0,2 с	50 МБ
SCR1	26	0	4	0	0	1,8 с	140 МБ
VeeR EH1	15	0	0	0	0	6,9 с	469 МБ
Ibex	6	0	0	0	8	0,6 с	108 МБ
VeeR EL2	12	0	0	0	0	10 с	646 МБ
BlackParrot	18	3	8	3	41	9,4 с	954 МБ
CVA6	25	0	14	0	0	3,2 мин	2,2 ГБ
OpenTitan	625	55	100	113	68	30 мин	6,6 ГБ
OpenC910	1201	991	134	48	20	67 мин	8,1 ГБ
Всего	1934	1049	261	164	137		

Время и память измерены на рабочей станции (раздел 6.1) на синтезируемой части проектов и включают разбор, трансляцию в RTLIL, извлечение и графовый анализ автоматов.

Точность средств анализа автоматов оценивалась по разметке всех извлеченных автоматов и всех диагностик (раздел 6.1). Автоматом считался регистр, следующее значение которого выбирается из констант, причем хотя бы одно условие выбора зависит, непосредственно или через комбинационную логику, от текущего значения регистра. Регистр, который лишь загружает константу по внешнему условию и удерживает ее, автоматом не считается. Для отсева таких регистров средство извлечения применяет фильтр зависимости, отключаемый отдельной опцией. Под критерий подпадают также указатели арбитров и очередей

и регистры выбора и масок, условия загрузки которых зависят от их значения (например, `port_sel_r` в SCR1 и `fpu_op_q` в CVA6). По этому критерию истинны 241 из 255 размеченных автоматов (**94,5 %**, 95 %-й доверительный интервал 91,0–96,7 %). Ложные извлечения (семь в OpenC910, шесть в OpenTitan и одно в CVA6) относятся к регистрам, которые загружают значение по внешним условиям и удерживают его и которые фильтр зависимости не отбросил. Истинны все размеченные диагностики каждого из четырех видов: 135 мертвых переходов, 134 недостижимых состояния, 79 тупиковых состояний и 101 взаимоблокировка. Часть взаимоблокировок приходится на регистры такого же рода, что и регистры выбора: маску смены потока управления в OpenC910 и регистр числа раундов блока AES в OpenTitan. Оценка точности анализа автоматов по всем диагностикам и извлеченным автоматам составляет **95,5 %** (по проектам от 93,4 до 100 %).

Существенная часть экземпляров автоматов порождается при построении развернутой модели из параметризованных модулей: из 1201 экземпляра проекта OpenC910 уникальными являются **88** (для дубликатов дублируются и диагностики). Штатный проход обнаружения автоматов Yosys (раздел 6.7.6) без предложенных в главе 3 доработок обнаруживает на том же проекте 432 экземпляра (**34** уникальных), то есть доработки увеличивают число выявляемых уникальных автоматов более чем вдвое.

Сквозное время анализа автоматов и пиковая память приведены в той же таблице 7. На сервере с менее производительным процессором анализ проекта OpenC910 занимает 177 мин (таблица 9). Время определяется прежде всего числом и сложностью извлекаемых автоматов, а не объемом описания: проекты с сотнями экземпляров автоматов (OpenTitan, OpenC910) анализируются десятки минут, остальные проекты анализируются от долей секунды до нескольких минут. Такая зависимость согласуется с оценкой вычислительной сложности извлечения, приведенной в разделе 3.3, где время определяется перебором комбинаций входов автомата. Из 67 мин анализа OpenC910 около 55 % (37 мин) приходится на извлечение автоматов, около 34 % (23 мин) приходится на выявление регистров состояния, каждый из остальных проходов (разбор и построение сети логических элементов, сопоставление условий переходов, графовый анализ, оптимизация) занимает не более 4 %.

6.4 Оценка анализа с привлечением сопутствующих форматов

Подсистема интерпретации UPF и статического анализа модели подсистемы питания (глава 4) оценивалась на синтетических тестах и открытых проектах. Для проверки интерпретатора подготовлено 830 синтетических тестов языковых конструкций UPF. Корректность построения и анализа модели проверялась на 20 полноценных проектах, в том числе 2 позитивных, 17 негативных и одном контрольном проекте из приложения Е стандарта IEEE 1801-2018, для которого был разработан соответствующий RTL-код.

Ввиду малого количества открытых проектов с описанием подсистемы питания в формате UPF тестирование на открытых проектах выполнялось на OpenC910, x-heep, 8b10b, Low-Power-Async-CDC-FIFO-IP и тестовом экземпляре из проекта OpenROAD. В OpenC910 обнаружены незначительные отклонения от стандарта (комментарии в стиле C/C++ вместо Tcl, передача команде `set_design_top` имени экземпляра модуля вместо имени самого модуля), которые могут быть следствием особенностей инструмента, использовавшегося разработчиками. В проектах x-heep и 8b10b выявлены несоответствия стандарту: отсутствие обязательной опции `-update` в команде `add_power_state` и недокументированная опция `-control_ports` в команде `create_power_switch` соответственно. О нарушениях сообщено разработчикам проектов. В экземпляре из проекта OpenROAD обнаружена ошибка использования необъявленного объекта, которая была подтверждена и исправлена разработчиками. В проекте Low-Power-Async-CDC-FIFO-IP выявлено использование в команде `create_power_switch` необъявленной логической цепи управляющего порта, о чем сообщено разработчикам.

Сравнительное тестирование с инструментами VCS и SpyGlass показало ограниченность поддержки стандарта UPF в обоих инструментах: VCS обрабатывает UPF-описания с небольшими ограничениями, тогда как в SpyGlass наблюдаются значительные отличия в синтаксисе опций и степени поддержки команд (в частности, не поддерживаются команды `set_variation`, `set_correlated` и опция `-model` команды `add_power_state`), что требует адаптации описания подсистемы питания и соответствующего RTL-кода под конкретный инструмент. Разработанная система поддерживает все команды стандарта IEEE 1801-2018.

Анализатор формата SDF (раздел 4.2.2) проверен на 94 модульных тестах, включающих все примеры раздела 32 стандарта IEEE 1800-2023: разрешение иерархии с массивами экземпляров и блоками генерации, сопоставление путей, межсоединений и временных проверок, привязку вызовов \$sdf_annotate при кэшировании тел экземпляров модулей. SDF-описания формируются после логического синтеза и размещения схемы, а проекты тестового набора представляют собой RTL-описания и SDF-описаний не содержат, поэтому для проверки анализатора использованы модульные тесты.

Детекторы, использующие информацию библиотек Liberty (обнаружение RS-защелок и буферизованных тактовых сигналов, раздел 4.5), оценивались в составе общего набора детекторов (раздел 6.2). На рассмотренных проектах с открытым исходным кодом срабатываний этих детекторов не зафиксировано. Отсутствие срабатываний отражает свойства тестового набора, а не неработоспособность детекторов: проекты тестового набора представляют собой RTL-описания до логического синтеза, в которых экземпляры ячеек технологической библиотеки отсутствуют, а выявляемые указанными детекторами конструкции (RS-защелка, собранная из отдельных логических элементов, и ручная буферизация тактового сигнала библиотечными буферами) характерны для описаний, адаптированных под конкретную технологическую библиотеку, и в рассмотренных проектах не встречаются. Корректность обоих детекторов подтверждена на синтетических тестах, содержащих соответствующие нарушения и парные им описания без нарушений (раздел 6.1).

6.5 Оценка механизма сопоставителей узлов АСД

Механизм сопоставителей (раздел 2.8) оценивался контрольным экспериментом на трех детекторах, каждый из которых реализован в двух вариантах, на обходчиках и на сопоставителях (примеры реализации приведены в разделе 5.1.7): DESIGN04 «блок always без управления событиями» (вторичный поиск потомков сопоставителем hasDescendant, использует граф узлов), SYNTH10 «использование операторов сравнения === и !==» (отбор бинарных выражений по виду операции, граф узлов не используется) и PORT10 «подключение шин в обратном порядке разрядов» (отбор экземпляров модулей предикатом, граф уз-

лов не используется). Детекторы выбраны так, чтобы охватить оба режима работы механизма: с построением графа узлов и без него (раздел 2.8.3).

Методика эксперимента следующая. Каждый детектор запускался по отдельности в обеих реализациях на десяти открытых проектах, используемых в экспериментах главы (раздел 6.2). Затем все три детектора запускались совместно. Измерялось полное астрономическое время работы анализатора (elapsed wall clock, `/usr/bin/time -v`), включающее разбор исходных текстов, построение АСД и развернутой модели, построение кэшей механизма сопоставителей и регистрацию сопоставителей. Каждая конфигурация запускалась десять раз. В таблице 8 приведены геометрические средние по десяти запускам.

Таблица 8 — Полное время работы анализатора с детекторами на обходчиках и на сопоставителях, с (в ячейке: обходчики / сопоставители, ускорение выделено полужирным)

Проект	SYNTH10	DESIGN04	PORT10	Все три совместно
VeeR EL2	0,244 / 0,123 (1,98)	0,233 / 0,132 (1,77)	0,242 / 0,123 (1,97)	0,295 / 0,142 (2,08)
BlackParrot	0,288 / 0,237 (1,22)	0,311 / 0,254 (1,23)	0,293 / 0,244 (1,20)	0,324 / 0,274 (1,18)
OpenTitan	0,975 / 0,684 (1,43)	0,954 / 0,769 (1,24)	0,969 / 0,687 (1,41)	1,194 / 0,898 (1,33)
CVA6	0,193 / 0,142 (1,36)	0,185 / 0,158 (1,17)	0,189 / 0,143 (1,32)	0,216 / 0,151 (1,43)
OpenC910	1,225 / 0,508 (2,41)	1,255 / 0,613 (2,05)	1,234 / 0,512 (2,41)	1,517 / 0,661 (2,29)

В таблицу включены пять проектов, на которых полное время работы превышает 0,1 с (на остальных пяти время обеих реализаций меньше и для сравнения нерепрезентативно). Результаты объясняются устройством механизма (раздел 2.8.3). Ускорение уже при одиночном запуске детектора (от **1,17** до **2,41** раза) обусловлено двумя свойствами реализации. Во-первых, обходчик посещает тело каждого экземпляра модуля, тогда как механизм сопоставителей обходит каждое уникальное тело однократно, через канонический экземпляр. Выигрыш наибольший на проектах с большим числом экземпляров модулей (на OpenC910 до 2,41 раза). Во-вторых, сопоставление выполняется последовательным просмотром непрерывного контейнера узлов целевого типа, а не полным обходом дерева. Влияние графа узлов также прослеживается в данных: детекторы SYNTH10 и PORT10 декларируют отказ от его построения, и их ускорение выше, чем у детектора DESIGN04, которому граф необходим для сопоставителя `hasDescendant` (на OpenC910 2,41 против 2,05, на VeeR EL2 1,97–1,98 против

1,77). На проекте BlackParrot различия между детекторами находятся в пределах разброса измерений.

При совместном запуске каждый детектор на обходчиках выполняет собственный полный обход дерева, тогда как на сопоставителях один общий обход и общие кэши обслуживают все три проверки, поэтому переход от одного детектора к трем почти не увеличивает время: на OpenC910 наблюдается рост с 0,51–0,61 до 0,66 с против роста с 1,22–1,25 до 1,52 с на обходчиках. Ускорение совместного запуска составляет от **1,18** до **2,29** раза. Отметим, что измеренное время включает одинаковую для обеих реализаций стоимость разбора, построения АСД и развернутой модели, поэтому приведенные значения являются нижними оценками ускорения собственно анализа.

Таким образом, механизм сопоставителей дает выигрыш в производительности уже при одной включенной проверке и растущую экономию при выполнении набора проверок при меньшем объеме кода детектора: фрагменты реализации правила DESIGN04 в листингах 5.5 и 5.6 занимают 36 и 25 строк соответственно. Всего на сопоставителях реализовано 18 детекторов модуля анализа.

6.6 Сравнение с коммерческим аналогом

Выполнено сравнение с коммерческим инструментом статического анализа SpyGlass [56] (версия V-2023.12) на десяти проектах тестового набора (раздел 6.2). Оба инструмента запускались на сервере (раздел 6.1) на одних и тех же исходных текстах: детекторы правил на полном наборе с тестовым окружением, средства анализа автоматов на синтезируемой части проектов. Для разработанного инструмента использованы запуски раздела 6.2 с полным набором проверок. Сравнение ограничено пересечением наборов правил. По документации SpyGlass аналог найден для 106 правил набора, им соответствуют 124 правила SpyGlass. Числа различаются, поскольку соответствие не взаимно однозначно: например, правилу CASE01 соответствуют три правила SpyGlass (W69, STARC05-2.8.1.4 и Av_case_default_missing), а правило W287a соответствует трем правилам набора (PORT03, PORT12 и PORT13). Сопоставленные правила SpyGlass проверяются четырьмя целями анализа (структурная и расширенная структурная проверки, проверка сбросов и проверка тестопригодности). Сообщения SpyGlass по осталь-

ным правилам из подсчетов исключены. Имена тактовых сигналов и сигналов сброса задавались обоим инструментам одинаково. На каждый запуск отводилось три часа, успешность запуска устанавливалась по журналу (отсутствие ошибок разбора и построения развернутой модели, наличие итогового отчета). Точность и взаимная полнота оценивались по методике раздела 6.1.

6.6.1 Применимость и ресурсоемкость

Разработанный инструмент завершил анализ всех десяти проектов как детекторами правил, так и средствами анализа автоматов (таблица 9). SpyGlass завершил анализ шести проектов: AES, PicoRV32, Ibex, VeeR EH1, OpenC910 и OpenTitan. На проектах SCR1 и VeeR EL2 разбор SpyGlass остановлен на конструкциях тестового окружения (системная задача \$assertoff с иерархическим аргументом и метод randomize), на проекте BlackParrot разбор остановлен на блоке generate без условия и цикла, который компилятор разработанного инструмента принимает. Сообщений по сопоставленным правилам на этих трех проектах нет. На проекте CVA6 анализ SpyGlass превысил лимит времени в три часа и не был завершен. В проекте Ibex модуль ядра не прошел построение модели для синтеза из-за иерархических ссылок в коде трассировки и анализировался как «черный ящик», поэтому автоматов в нем SpyGlass не обнаружил.

Детекторы правил разработанного инструмента работают быстрее проверки сопоставленных правил в SpyGlass на всех шести проектах, анализ которых SpyGlass завершил: от 0,6 до 27 с против 65 с и более на проектах объемом до 85 тыс. строк, 8,3 против 32 мин на OpenC910 и 24,2 против 111 мин на OpenTitan. Основную часть времени SpyGlass на двух крупнейших проектах занимают проверки сбросов и тестопригодности. Одна структурная проверка RTL-описания, к которой относятся 105 из 124 сопоставленных правил, занимает 3,3 мин на OpenC910 и 4,4 мин на OpenTitan, то есть меньше времени детекторов правил разработанного инструмента. По памяти разработанный инструмент экономнее на пяти проектах из шести и требует больше памяти на OpenTitan (12,5 против 8,5 ГБ).

Таблица 9 — Время и пиковая память разработанного инструмента и SpyGlass (время / память)

Проект	Детекторы правил		Анализ автоматов	
	SVAN	SpyGlass	SVAN	SpyGlass
AES	0,6 с / 35 МБ	76 с / 611 МБ	1,2 с / 55 МБ	15 с / 461 МБ
PicoRV32	12 с / 52 МБ	65 с / 596 МБ	0,9 с / 39 МБ	14 с / 431 МБ
SCR1	17 с / 4,7 ГБ	ошибка компиляции	5,6 с / 134 МБ	ошибка компиляции
VeeR EH1	27 с / 273 МБ	3,1 мин / 803 МБ	19 с / 459 МБ	31 с / 676 МБ
Ibex	22 с / 438 МБ	77 с / 634 МБ	2,1 с / 99 МБ	17 с / 462 МБ
VeeR EL2	2,2 мин / 1,7 ГБ	ошибка компиляции	29 с / 641 МБ	ошибка компиляции
BlackParrot	25 с / 782 МБ	ошибка компиляции	28 с / 953 МБ	ошибка компиляции
CVA6	12,7 мин / 7,7 ГБ	превышен лимит времени	9,0 мин / 2,2 ГБ	превышен лимит времени
OpenTitan	24,2 мин / 12,5 ГБ	111 мин / 8,5 ГБ	83 мин / 6,5 ГБ	4,2 мин / 3,3 ГБ
OpenC910	8,3 мин / 4,0 ГБ	32 мин / 5,5 ГБ	177 мин / 8,1 ГБ	4,0 мин / 3,5 ГБ

Детекторы правил SVAN запускались так же, как в таблице 5. Для детекторов правил SpyGlass приведены сумма времени и максимум памяти по четырем целям анализа. Анализ автоматов SpyGlass входит в расширенную структурную проверку, для него приведены время и память этой проверки.

Анализ автоматов разработанного инструмента быстрее на четырех малых проектах и существенно медленнее на OpenTitan и OpenC910 (83 и 177 мин против 4,2 и 4,0 мин). При этом SpyGlass за указанное время только обнаруживает автоматы и их взаимодействия и находит на порядок меньше автоматов (раздел 6.6.4). Формальная проверка переходов автоматов, которая соответствует диагностикам правила FSM03, в таблицу не включена: у SpyGlass она дала результаты на проектах AES, PicoRV32 и VeeR EH1 и не уложилась в три часа на OpenC910 и OpenTitan.

6.6.2 Точность и полнота

У SpyGlass сработали 23 правила из 124, всего 19 452 сообщения по сопоставленным правилам, из которых 13 415 приходятся на шумное правило W287b (явно неподключенный выходной порт экземпляра). Размечено 1658 сообщений, из них истинны 1364 и ложны 294. Точность и взаимная полнота сравниваются на шести проектах, анализ которых SpyGlass завершил, по правилам набора, имеющим аналог в SpyGlass. Оценка точности без шумных правил PORT04 и W287b

составляет **95,7 %** у разработанного инструмента и **79,8 %** у SpyGlass, с шумными правилами оценки равны **99,4** и **93,9 %**. Правила об автоматах в эти оценки не входят и разобраны отдельно (раздел 6.6.4).

Сообщения сведены к находкам: сообщения одного правила об одной позиции исходного текста (файл и строка) считаются одной находкой. Например, 768 сообщений правила W240 о разных разрядах и полях портов дают 373 находки. Разработанный инструмент находит **49 %** истинных находок SpyGlass без учета правила W287b (8 % с его учетом). SpyGlass находит **29 %** истинных находок разработанного инструмента без учета правила PORT04 (4 % с его учетом). По 16 правилам, по которым истинные находки есть у обоих инструментов, SpyGlass находит 47 % находок разработанного инструмента. Точность и взаимная полнота по проектам приведены в таблице 10.

Таблица 10 — Точность и взаимная полнота SVAN и SpyGlass по проектам

Проект	Срабатываний		Точность, %		Взаимная полнота, %	
	SVAN	SpyGlass	SVAN	SpyGlass	найдено SVAN	найдено SpyGlass
AES	4	4	100	50,0	0	0
PicoRV32	222	511	90,5	54,0	11	33
VeeR EH1	833	732	95,4	80,3	53	26
Ibex	234	191	99,6	75,9	54	36
OpenC910	2129	522	99,9	99,3	65	16
OpenTitan	1875	3860	91,1	80,7	49	45
Всего	5297	5820	95,7	79,8	49	29

Без шумных правил и правил об автоматах. Столбец «найдено SVAN» содержит долю истинных находок SpyGlass, найденных разработанным инструментом, столбец «найдено SpyGlass» содержит обратную долю.

Точность и полнота по отдельным правилам приведены в таблице 11. В нее включены все сопоставленные правила набора, по которым на шести проектах сработал хотя бы один из инструментов. Если одно правило SpyGlass соответствует нескольким правилам набора, эти правила набора приведены в одной строке. Правила набора, сопоставленные правилам SpyGlass которых не сработали, приведены одной строкой, кроме шумного правила PORT04. Точность правила оценена по его срабатываниям на шести проектах. Правила об автоматах приведены в конце таблицы и в итоговые строки не входят. Для них находкой считается автомат (FSM01), пара автоматов (FSM02) и состояние автомата (FSM03).

Таблица 11 — Точность и взаимная полнота SVAN и SpyGlass по правилам

Правило SVAN	Правило SpyGlass	Точность, %		Истинные находки SpyGlass		Истинные находки SVAN	
		SVAN	SpyGlass	всего	найдено SVAN	всего	найдено SpyGlass
SIGNAL17	W528	99,3	99,5	1131	833 (74 %)	1217	832 (68 %)
ASSIGN05	W415a	63,8	43,7	787	43 (5 %)	134	125 (93 %)
ASSIGN02	STARC05-2.2.3.3	100	100	174	0	27	0
PORT01	W240	100	100	373	337 (90 %)	466	325 (70 %)
PORT03, PORT12, PORT13	W287a	100	100	38	38 (100 %)	89	38 (43 %)
ASSIGN01	NoAssignX-ML	100	100	21	17 (81 %)	665	18 (3 %)
DESIGN02	Av_deadcode01	100	100	58	0	6	0
DESIGN03	FlopEConst	—	100	10	0	0	—
TYPE02	W216	—	100	8	0	0	—
LATCH01	InferLatch	100	100	4	1	5	1
RESET08	STARC05-1.3.1.3	100	100	4	0	6	0
SIGNAL18, TYPE04	W224, STARC05-2.1.5.3	100	100	4	4	14	4 (29 %)
WIDTH08	ParamWidthMismatch-ML	100	100	3	0	1	0
SYNTH13	SYNTH_5143	94,8	100	2	2	228	2 (1 %)
SIGNAL14	STARC05-1.4.3.4	—	100	2	0	0	—
ARITH01	W486	100	100	1	1	14	1 (7 %)
PORT02	W287b	—	100	13 415	0	0	—
SIGNAL07, VARIABLE06	W123*	100	0	0	—	28	0
SIGNAL16	UndrivenInTerm-ML*	100	0	0	—	131	0
Остальные правила (22) [†]	срабатываний нет	93,6	—	0	—	1666	0
PORT04	срабатываний нет	100	—	0	—	32 563	0
Всего		99,4	93,9	16 035	1276 (8 %)	37 260	1346 (4 %)
Всего без шумных правил		95,7	79,8	2620	1276 (49 %)	4697	1346 (29 %)
FSM01	Av_fsminf01	91,4	100	168	137 (82 %)	1694	125 (7 %)
FSM02	Av_fsminf02	99,7	61,8	5	5	22 019	5 (< 0,1 %)
FSM03 [‡]	Av_fsm_analysis	100	100	1	1	1	1

* Все сообщения правила ложны. [†] Правила ASSIGN03, ASSIGN06, CASE01, CASE03, CASE04, CASE05, LOOP02, RESET05, RESET06, SIGNAL03, SIGNAL04, SIGNAL09, SYNTH07, SYNTH08, SYNTH10, TIMING01, VARIABLE04, WIDTH01, WIDTH04, WIDTH05, WIDTH06 и WIDTH07. [‡] На проектах AES, PicoRV32 и VeeR EH1, где формальная проверка SpyGlass завершилась.

6.6.3 Разбор находок

Ложные срабатывания. Ложные срабатывания разработанного инструмента относятся к классам, описанным в разделе 6.2. Ложные сообщения SpyGlass относятся к пяти правилам. Из 383 размеченных сообщений правила W415a (аналог ASSIGN05) ложны 264: это переопределение значения по умолчанию по условию, присваивания во взаимоисключающих ветвях и накопление значения ($x \mid = \dots$), которые правило ASSIGN05 нарушением не считает. Из 392 размеченных сообщений правила W528 (аналог SIGNAL17) ложны семь: сигналы читаются иерархической ссылкой, в утверждении или через неявное подключение портов либо являются массивами памяти, которые SpyGlass не обработал из-за размера. Все восемь сообщений правила W123 (аналог SIGNAL07) сообщают о сигналах, не обработанных из-за размера, а оба сообщения правила UndrivenInTerm-ML (аналог SIGNAL16) относятся к массиву ПЗУ, который входным сигналом не является. Ложные сообщения правила Av_fsminf02 разобраны в разделе 6.6.4.

Находки только SpyGlass. Основная часть пропусков разработанного инструмента приходится на четыре правила. По правилу ASSIGN05 (744 находки) детектор не диагностирует присваивание части сигнала (разряда или поля) после присваивания всего сигнала в том же блоке, по правилу ASSIGN02 (174 находки) не диагностируется повторное присваивание разрядов регистра во вложенном условном операторе. По правилу SIGNAL17 (298 находок) не диагностируется сигнал, получающий значение через выходной порт экземпляра модуля. По правилу PORT02 (13 415 находок) правило W287b отмечает явно пустое подключение выходного порта ($.q()$), тогда как диагностика компилятора сообщает только об опущенных портах. Почти все такие находки относятся к автоматически сгенерированным регистровым блокам OpenTitan, поэтому итоги приведены и без этого правила. Остальные 128 находок относятся к девяти правилам (DESIGN02, PORT01, DESIGN03, TYPE02, ASSIGN01, RESET08, WIDTH08, LATCH01 и SIGNAL14). Формальный анализ SpyGlass находит ветви, недостижимые при заданных значениях параметров (DESIGN02, 58 находок), и выводит константы и сигналы сброса из структуры описания (DESIGN03, RESET08, SIGNAL14), тогда как детекторы этих правил опираются на явную привязку к константе и на имена сигналов из конфигурации. Не диагностируются

также отдельные нечитаемые разряды структурных портов (PORT01), выбор диапазона разрядов переменной типа `int` (TYPE02) и литерал без разрядности в значении параметра (WIDTH08). Защелки, порожденные атрибутом `full_case` (LATCN01), разработанный инструмент диагностирует по правилу CASE05.

Находки только разработанного инструмента. По 25 правилам с сопоставленным аналогом (операторы `case`, разрядности, блокирующие присваивания в последовательностной логике, операторы `===` и `!==`, знаковость и другие) истинные находки на шести проектах есть только у разработанного инструмента, всего 1825 находок без учета правила PORT04. Только у SpyGlass истинные находки есть по четырем правилам (DESIGN03, TYPE02, SIGNAL14 и PORT02), у обоих инструментов истинные находки есть по 16 правилам. Среди находок разработанного инструмента есть функциональные дефекты. В проекте OpenC910 разряды 32–39 сигнала прерываний управляются двумя непрерывными присваиваниями с перекрывающимися диапазонами (правило SIGNAL09). В проекте CVA6 метки со значением `?` использованы в обычном операторе `case`, где сравнение с ними всегда ложно (правила CASE07 и CASE08), а индекс массива в модуле регистров управления выходит за границу (правило RANGE01).

6.6.4 Анализ конечных автоматов

Извлечение автоматов. Разработанный инструмент извлек 1934 экземпляра автоматов на десяти проектах, из них 1853 на шести проектах, анализ которых SpyGlass завершил. SpyGlass извлек на этих проектах 168 автоматов, все 160 размеченных из них истинны. Оценка точности извлечения у разработанного инструмента на этих проектах, вычисленная по разметке раздела 6.3, составляет **91,4 %**. Автоматы сопоставлялись по иерархическому имени регистра состояния. Разработанный инструмент нашел все 137 автоматов SpyGlass с числом состояний больше двух. У остальных 31 автомата SpyGlass два состояния и одноразрядный регистр состояния. Такие автоматы разработанный инструмент по умолчанию не извлекает, их извлечение включается отдельной опцией. Таким образом, полнота извлечения относительно находок SpyGlass составляет **82 %**, а для автоматов более чем с двумя состояниями **100 %**. SpyGlass, в свою очередь, обнаружил 137 из 1853 экземпляров, извлеченных разработанным инструментом (**7 %**). При-

чина состоит в способе обнаружения: SpyGlass распознает автомат по оператору case в процедурном блоке, а разработанный инструмент распознает его по структуре сети логических элементов независимо от стиля кодирования.

Показательный пример дает семейство ядер VeeR. В ядре VeeR EH1 SpyGlass обнаружил один автомат из 15, извлеченных разработанным инструментом. В ядрах VeeR логика следующего состояния размещена в блоке `always_comb`, а регистр состояния представлен экземпляром отдельного параметризованного модуля `rvdff`, подключенным через порты (рисунок 6.1). После построения развернутой модели и трансляции в RTLIL экземпляр модуля триггера раскрывается, и дерево мультиплексоров оказывается непосредственно подключенным к триггеру, поэтому предложенный метод выявляет такие автоматы.

Дефекты графа автомата. Формальная проверка переходов в SpyGlass дала результаты на проектах AES, PicoRV32 и VeeR EH1. На AES и VeeR EH1 нарушений не найдено ни одним из инструментов. На PicoRV32 SpyGlass сообщил о шести невыполнимых переходах в состояние `cpu_state_ld_rs2` и из него. Это состояние разработанный инструмент диагностирует как недостижимое при значении параметра `ENABLE_REGS_DUALPORT` по умолчанию. На проектах OpenC910 и OpenTitan, на которые приходятся 1529 из 1611 диагностик правила FSM03, формальный анализ SpyGlass не уложился в три часа.

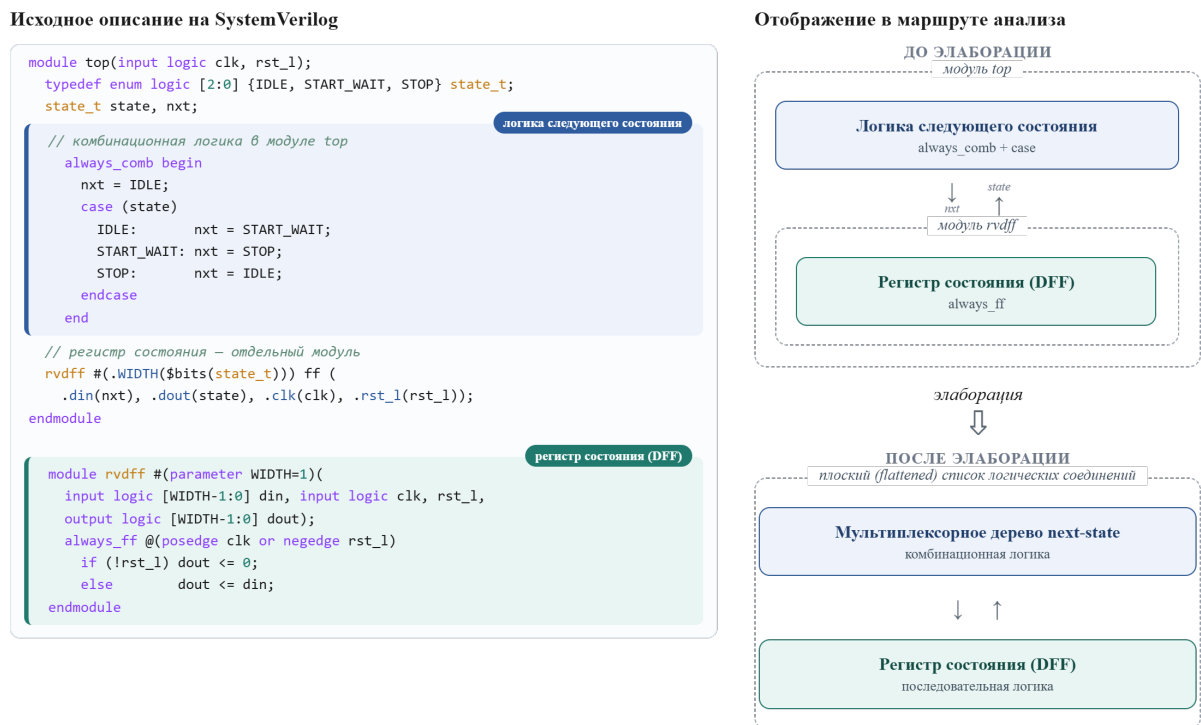


Рисунок 6.1 — Логика следующего состояния и регистр состояния в разных экземплярах модулей, соединенных через порты (ядро VeeR EL2)

Взаимодействующие автоматы. SpyGlass (правило Av_fsminf02) выдал 34 сообщения о 17 парах автоматов на шести проектах, отдельное сообщение выдается на каждый сигнал связи. По критерию правила FSM02 (на пути между автоматами не более одного последовательностного элемента) истинны пять пар, и все они найдены разработанным инструментом. В остальных 12 парах (13 сообщений, девять пар в OpenC910 и три в OpenTitan) связь проходит через несколько регистров, через шину TL-UL или через канал оповещений с синхронизатором. Разработанный инструмент сообщил еще о 22 071 паре взаимодействующих экземпляров автоматов, различных пар автоматов среди них 998. В выборке из 100 различных пар взаимодействие управляющих автоматов подтверждено для 97. В трех остальных парах проекта OpenTitan вторым автоматом является служебный регистр средств защиты от сбоев (регистр оповещения об ошибке блока КМАС и теневой регистр управления блока AES), значение которого изменяется по сигналам контроля ошибок.

6.7 Сравнение с открытыми средствами статического анализа

Разработанный инструмент сравнивался с пятью открытыми средствами на десяти проектах тестового набора (раздел 6.2). Средства различаются по назначению: Verilator строит развернутую модель и выдает диагностику, svlint и Verible выполняют синтаксический анализ отдельных файлов, Surelog является языковым фронтендом, а Yosys является инфраструктурой логического синтеза. Если всем средствам подать одни и те же исходные тексты, часть средств окажется в несвойственных им условиях (например, инфраструктура синтеза получит тестовые окружения). Поэтому разработанный инструмент сравнивается с каждым средством отдельно: набор исходных текстов выбирается по назначению сравниваемого средства, и разработанный инструмент анализирует тот же набор. Численные результаты сравнений с разными средствами напрямую не сопоставляются.

6.7.1 Условия и методика сравнения

Условия. Сравнение выполнено на рабочей станции (раздел 6.1). Использовались Verilator 5.051 (ревизия 877fc1e81), svlint 0.9.5, Verible v0.0-4082, Surelog 1.86 и Yosys 0.67+40 (ревизия 45ea2b8d6). Yosys собран с тем же фронендом slang, на котором построен разработанный инструмент, поэтому оба средства разбирают SystemVerilog одинаково. Средствам диагностики (Verilator, svlint, Verible, Surelog) подавался полный набор исходных текстов проекта с тестовым окружением. Инфраструктуре синтеза Yosys подавалась синтезируемая часть проекта с одним модулем верхнего уровня. Списки файлов, макроопределения и каталоги включения взяты из конфигураций системы непрерывной интеграции разработанного инструмента. Verilator запускался в режиме `--lint-only -Wall -Wno-fatal`. Surelog запускался с ключом `-synth`, svlint запускался с конфигурационным файлом отобранных правил, а Yosys выполнял сценарий из проходов `proc`, `check` и `fsm_detect`. Разработанный инструмент запускался с полным набором проверок (раздел 6.2). Все средства работали в одном потоке, на каждый запуск отводилось три часа. Версии проектов, таблица соответствия правил, матрица срабатываний и замеры по проектам приведены в приложении В.

Отбор правил. Учитывались только срабатывания диагностик, сопоставленных правилам набора. Кроме пяти сравниваемых средств, рассматривалось средство Icarus Verilog 14.0. По документации этих шести средств 58 правил набора проверяются хотя бы одним из них. Каждое сопоставление проверено на синтетических тестах (71 тест с нарушением и 58 парных тестов без нарушения) и считалось подтвержденным, если диагностика средства срабатывает на тесте с нарушением и не срабатывает на парном тесте. Подтверждены 63 сопоставления для 42 правил набора: 34 правила проверяются детекторами, семь правил проверяются диагностиками компилятора, а правилу FSM01 (извлечение автоматов) соответствует проход `fsm_detect` инфраструктуры Yosys. Для остальных 16 правил документированная диагностика на тесте не срабатывает либо проверяет иное свойство. Диагностика PINCONNECTEMPTY средства Verilator исключена из сопоставления с правилом PORT03 по срабатываниям на проектах: она отмечает намеренно неподключенные выходные порты (все 100 размеченных срабатываний из 13 989), то есть относится к правилу PORT02. Средство Icarus

Verilog разобрало только два наименьших проекта и сопоставленных срабатываний не дало, поэтому отдельное сравнение с ним не проводилось.

Единица сопоставления. В сравнении с открытыми средствами для правил о сигналах (SIGNAL07, SIGNAL09, SIGNAL16, SIGNAL17, LATCH01, RESET08) находкой считается сигнал, а не строка исходного текста, поскольку средства указывают на разные строки: на объявление сигнала либо на его присваивание. Для правила CASE01 находкой считается оператор case, для правила FSM01 находкой считается регистр состояния. По правилу ASSIGN03 средство svlint сообщает о процедурном блоке, а разработанный инструмент о каждом присваивании, поэтому находка разработанного инструмента считается найденной svlint, если присваивание находится в блоке, о котором сообщил svlint.

Точность и полнота. Точность и взаимная полнота оценивались по методике раздела 6.1. Границы доверительного интервала для средства получены подстановкой границ интервалов отдельных диагностик. Кроме того, на синтетических тестах определялась доля правил с документированным аналогом, нарушение которых средство обнаруживает. Находки сравниваемого средства учитывались только в модулях, вошедших в развернутую модель, то есть имеющих экземпляры в иерархии модуля верхнего уровня: разработанный инструмент анализирует развернутую модель, тогда как синтаксические средства проверяют текст каждого файла. Показатели сравнения с каждым средством вычисляются по тем проектам, на которых получены результаты обоих средств.

6.7.2 Сравнение с Verilator

Verilator является ближайшим по возможностям средством: он строит развернутую модель и обладает наибольшим среди открытых средств пересечением правил с разработанным набором. По документации его диагностикам соответствуют 50 правил набора, на синтетических тестах подтверждены сопоставления для 33 правил (21 диагностика). Точность и взаимная полнота по правилам приведены в таблице 12, а по проектам в таблице 13.

Таблица 12 — Точность и взаимная полнота SVAN и Verilator по правилам

Правило SVAN	Диагностика Verilator	Точность, %		Истинные находки Verilator		Истинные находки SVAN	
		SVAN	Verilator	всего	найдено SVAN	всего	найдено Verilator
SIGNAL17	UNUSED SIGNAL	99,1	27,4	1029	670 (65 %)	1632	669 (41 %)
WIDTH07	WIDTHTRUNC	99,1	98,2	311	195 (63 %)	1035	187 (18 %)
WIDTH04	WIDTHEXPAND, WIDTHTRUNC	63,9	84,3	171	79 (46 %)	358	65 (18 %)
SIGNAL16	UNDRIVEN	100	99,1	112	111 (99 %)	160	111 (69 %)
SIGNAL07	UNDRIVEN	100	75,8	91	2 (2 %)	20	2 (10 %)
WIDTH05	WIDTHEXPAND	100	28,3	73	36 (49 %)	71	51 (72 %)
ASSIGN03	BLKSEQ	100	100	52	50 (96 %)	109	51 (47 %)
RESET08	SYNCA SYNCNET	100	100	16	0 (0 %)	2	0
SYNTH13	STMTDLY	97,4	100	15	15 (100 %)	1013	21 (2 %)
CASE01	CASEINCOMPLETE	100	100	10	6 (60 %)	58	6 (10 %)
WIDTH02	WIDTHTRUNC	100	100	8	7	7	7
CASE02	CASEOVERLAP	100	100	4	3	45	7 (16 %)
CASE03	WIDTHEXPAND	100	100	4	3	42	7 (17 %)
LATCH01	LATCH	100	100	2	2	26	2 (8 %)
LOOP01	INFINITE LOOP	—	0	0	—	0	—
SIGNAL09	MULTIDRIVEN	100	1,1	1	1	4	1
ASSIGN04	COMBDLY	—	0	0	—	0	—
DESIGN01	UNOPTFLAT	—	0	0	—	0	—
Остальные правила (7)*	истинных находок нет	100	—	0	—	240	0 (0 %)
Всего		94,3	38,8	1899	1180 (62 %)	4822	1187 (25 %)

Сообщения WIDTHEXPAND и WIDTHTRUNC отнесены к правилу по виду операции: присваивание к WIDTH07, арифметическая или поразрядная операция к WIDTH04, сравнение к WIDTH05, подключение порта к WIDTH02, метка case к CASE03. Диагностика UNDRIVEN отнесена к правилу SIGNAL16, если объект без источника является портом, и к правилу SIGNAL07 в остальных случаях. Точность Verilator по правилу оценена по сообщениям, отнесенным к этому правилу. * Правила SIGNAL03, VARIABLE04, ARITH01, WIDTH06, CASE07, CASE08 и CASE05.

Анализ Verilator на проекте CVA6 прерван ошибкой разбора, поэтому для этого проекта получена только часть находок. Разработанный инструмент находит **62 %** истинных находок Verilator, а Verilator находит **25 %** истинных находок разработанного инструмента (26 % по правилам, по которым истинные находки есть у обоих средств).

Таблица 13 — Точность и взаимная полнота SVAN и Verilator по проектам

Проект	Срабатываний		Точность, %		Взаимная полнота, %	
	SVAN	Verilator	SVAN	Verilator	найдено SVAN	найдено Verilator
AES	4	0	100	—	—	0
PicoRV32	81	36	96,3	100	84	29
SCR1	310	147	98,1	47,6	65	11
VeeR EH1	458	495	91,9	33,5	52	23
VeeR EL2	532	988	97,7	43,0	42	32
Ibex	131	9	99,2	23,9	90	2
BlackParrot	422	968	97,4	57,6	38	48
CVA6	1578	581	90,8	52,5	90	16
OpenC910	1058	1548	99,8	20,7	86	51
OpenTitan	1149	192	90,5	23,0	43	2
Bcero	5723	4964	94,3	38,8	62	25

Точность. Оценка точности Verilator (38,8 %) отражает прежде всего различие проверяемых свойств: его диагностики шире сопоставленных правил. Сообщения WIDTHEXPAND и WIDTHTRUNC истинны относительно правил о разрядности в 41 и 64 % случаев. Остальные относятся к сравнениям с операндом типа `int`, которые определение правила WIDTH05 исключает, к значениям параметров по умолчанию, сдвигам и другим конструкциям, на которые сопоставленные правила не распространяются. Диагностика UNUSED SIGNAL истинна относительно правила SIGNAL17 в 27 % случаев, большинство остальных ее сообщений относится к неиспользуемым входным портам (правило PORT01). Диагностики MULTIDRIVEN и UNOPTFLAT анализируют сигнал целиком, а не по разрядам, и поэтому сообщают о разных элементах одного массива и о мнимых комбинационных петлях. Истинно одно сообщение MULTIDRIVEN из 95, оно указывает на дефект проекта OpenC910, найденный и детектором правила SIGNAL09. Сообщения INFINITE LOOP и COMBDLY под определения правил LOOP01 и ASSIGN04 не подпадают. Ложные срабатывания разработанного инструмента относятся к классам раздела 6.2.

Полнота. На синтетических тестах Verilator обнаруживает нарушения 33 правил из 50 с документированным аналогом. Основная часть находок, обнаруженных только Verilator, приходится на пять правил, каждая причина пропуска воспроизведена на минимальном тесте. По правилу SIGNAL17 (359 находок)

не диагностируются сигналы, получающие значение через выходной порт экземпляра или в объявлении цепи с инициализатором. По правилам WIDTH07 и WIDTH04 (116 и 92 находки) не проверяются объявления цепей с инициализатором и выражения в объявлениях параметров. По правилу SIGNAL07 (89 находок) о большинстве таких сигналов разработанный инструмент сообщает по другим правилам (PORT03, PORT12, PORT13 и VARIABLE06), которые в оценке полноты не учитываются. По правилу WIDTH05 (37 находок) не диагностируется сравнение сигнала с константой заданной разрядности. Остальные 26 находок относятся к правилам RESET08, CASE01, ASSIGN03, CASE02, CASE03, WIDTH02 и SIGNAL16.

Находки разработанного инструмента, пропущенные Verilator, объясняются в основном соглашением проектов lowRISC: диагностика UNUSED SIGNAL не выдается для сигналов, имя которых содержит слово `unused`. Такое имя имеют 685 из 968 сигналов, о которых по правилу SIGNAL17 сообщил только разработанный инструмент. Кроме того, ряд диагностик Verilator не выдается для подпрограмм и блоков тестового окружения, а его диагностика несинтезируемых конструкций (STMTDLY) ограничена задержками.

6.7.3 Сравнение со svlint

svlint выполняет синтаксический анализ отдельных файлов без построения развернутой модели, поэтому свойства, требующие развернутой модели (межмодульные связи, типы, достижимость), ему недоступны. На синтетических тестах подтверждены семь правил svlint, соответствующие пяти правилам набора из шести с документированным аналогом. Отказы svlint вызваны переполнением стека при разборе проекта OpenTitan и включаемыми файлами проекта SCR1 в кодировке, отличной от UTF-8. На OpenC910 и CVA6 часть файлов не разобрана, поскольку макроопределения из включаемых файлов при пофайловом разборе не определены. Результаты сравнения по правилам приведены в таблице 14, а по проектам в таблице 15.

Таблица 14 — Точность и взаимная полнота SVAN и svlint по правилам

Правило SVAN	Правило svlint	Точность, %		Истинные находки svlint		Истинные находки SVAN	
		SVAN	svlint	всего	найдено SVAN	всего	найдено svlint
CASE01	*_case_default	100	38,8	40	27 (68 %)	56	27 (48 %)
ASSIGN03	blocking_assignment_in_always_*	100	100	13	11 (85 %)	77	71 (92 %)
SYNTH10	operator_case_equality	100	100	46	46 (100 %)	46	46 (100 %)
SYNTH13	procedural_continuous_assignment	97,4	—	0	—	816	0 (0 %)
Всего		97,8	45,9	99	84 (85 %)	995	144 (14 %)

Точность. Размечены все 1226 срабатываний svlint. Ложны 663 сообщения об операторах case без ветви default (правила explicit_case_default и implicit_case_default): 250 относятся к операторам unique и priority case, 413 относятся к операторам, метки которых покрывают все значения селектора. По определению правила CASE01 такие операторы нарушением не являются. Правило implicit_case_default выдает сообщение на каждое присваивание внутри оператора (976 сообщений на 59 операторов). Сообщения об операторах === и о блокирующих присваиваниях в последовательностной логике истинны.

Таблица 15 — Точность и взаимная полнота SVAN и svlint по проектам

Проект	Срабатываний		Точность, %		Взаимная полнота, %	
	SVAN	svlint	SVAN	svlint	найдено SVAN	найдено svlint
AES	0	4	—	50,0	0	—
PicoRV32	30	71	100	64,8	47	97
VeeR EH1	174	90	91,4	47,2	75	8
VeeR EL2	152	3	100	100	67	9
Ibex	40	2	100	100	100	5
BlackParrot	90	212	100	73,8	96	30
CVA6	511	756	98,2	40,9	100	13
OpenC910	100	88	100	2,3	100	3
Всего	1097	1226	97,8	45,9	85	14

Для части сообщений об операторах case доля истинных известна только по всем проектам в целом, она перенесена на сообщения каждого проекта.

Полнота. Проверка svlint, сопоставленная правилу SYNTH13, срабатываний не дала, поэтому ни одна из 816 истинных находок детектора этого

правила средством svlint не найдена. По правилу SYNTH10 находки в модулях развернутой модели совпадают полностью, еще 65 находок svlint относятся к модулям, которые в развернутую модель не вошли. Расхождение по правилу CASE01 разобрано по операторам case без ветви default, найденным svlint, Verible или Verilator, но не разработанным инструментом. Таких операторов 101 (81 найден svlint, 49 найдены Verible, 4 найдены Verilator, часть найдена несколькими средствами). Из них 28 являются операторами unique и priority case, 43 полностью покрывают значения селектора, восемь находятся в модулях, не вошедших в развернутую модель. Пропусками разработанного инструмента являются 22 оператора: 14 операторов вида case (1'b1) с метками-выражениями, покрытие которых статически не устанавливается, семь операторов с селектором перечислимого типа, в которых перечислены все именованные значения, но не все значения его разрядности (раздел 6.7.2), и оператор с атрибутом full_case проекта PicoRV32 (раздел 6.6.3). В свою очередь, svlint не обрабатывает операторы case во включаемых файлах и в файлах, которые ему не удалось разобрать.

6.7.4 Сравнение с Verible

Verible является синтаксическим средством диагностики стиля и не поддерживает препроцессирование с внешними макроопределениями и каталогами включения. На синтетических тестах подтверждены пять его правил, соответствующие пяти правилам набора из шести с документированным аналогом. На проекте OpenC910 часть файлов не разобрана из-за синтаксических ошибок на подключениях портов после нераскрытых макроопределений. Результаты сравнения по правилам приведены в таблице 16, а по проектам в таблице 17.

Точность. Размечены все 184 срабатывания Verible. Ложны 40 срабатываний правила case-missing-default: 37 относятся к операторам, метки которых покрывают все значения селектора, в трех средство не распознает ветвь default в операторе case (...) inside (проект CVA6). Ложны все девять срабатываний правила truncated-numeric-literal на литерале 1'hX в утверждениях, где цифра X формально занимает четыре разряда, но значение не усекается. Ложны и все 72 срабатывания правила module-port: оно отмечает любое позици-

онное подключение портов, тогда как сопоставленное правило PORT09 относится к экземплярам, у которых подключено меньше входов, чем объявлено.

Таблица 16 — Точность и взаимная полнота SVAN и Verible по правилам

Правило SVAN	Правило Verible	Точность, %		Истинные находки Verible		Истинные находки SVAN	
		SVAN	Verible	всего	найдено SVAN	всего	найдено Verible
CASE01	case-missing-default	100	43,7	25	19 (76 %)	58	19 (33 %)
ASSIGN03	always-ff-non-blocking	100	100	24	20 (83 %)	109	20 (18 %)
PORT09	module-port	—	0	0	—	0	—
WIDTH09	truncated-numeric-literal	—	0	0	—	0	—
Всего		100	34,2	49	39 (80 %)	167	39 (23 %)

Полнота. Пропуски разработанного инструмента составляют шесть операторов case тех же классов, что и в сравнении со svlint, и четыре блокирующих присваивания строковой переменной в блоке always_ff. Низкая полнота Verible объясняется тем, что его правило о блокирующих присваиваниях ограничено блоками always_ff и не охватывает блоки always с событием фронта, а также задачи, вызываемые из таких блоков.

Таблица 17 — Точность и взаимная полнота SVAN и Verible по проектам

Проект	Срабатываний		Точность, %		Взаимная полнота, %	
	SVAN	Verible	SVAN	Verible	найдено SVAN	найдено Verible
AES	0	2	—	0	—	—
PicoRV32	29	0	100	—	—	0
SCR1	34	7	100	42,9	67	6
VeeR EH1	19	4	100	63,0	0	0
VeeR EL2	29	2	100	100	0	0
BlackParrot	2	64	100	1,6	0	0
CVA6	48	99	100	55,0	90	77
OpenC910	6	6	100	0	—	0
Всего	167	184	100	34,2	80	23

Для части сообщений об операторах case доля истинных известна только по всем проектам в целом, она перенесена на сообщения каждого проекта.

6.7.5 Сравнение с Surelog

Surelog является языковым фронтом (разбор и построение развернутой модели в формате UHDM), диагностика качества описаний не входит в его основное назначение. По документации его диагностикам соответствуют 16 правил набора, на синтетических тестах подтверждены три: несинтезируемые конструкции (SYNTH13), повторное определение макроса (PP02) и деление на ноль при вычислении константы (ARITH02). Анализ Surelog завершен без ошибок на пяти проектах. На OpenC910 не раскрыты макроопределения из включаемых файлов (903 ошибки препроцессора), на OpenTitan разбор остановлен неподдерживаемой конструкцией и неразрешенной иерархической ссылкой. На BlackParrot, VeeR EL2 и CVA6 анализ завершен частично: на BlackParrot из-за синтаксической ошибки во включаемом файле, на VeeR EL2 и CVA6 из-за остановки на системных задачах \$error, которые выполняются при построении развернутой модели. Результаты сравнения по правилам приведены в таблице 18, а по проектам в таблице 19.

Точность. Размечены все 392 срабатывания Surelog. Все 386 сообщений о несинтезируемых конструкциях истинны, как и четыре из пяти сообщений о повторном определении макроса. Ложных сообщений два: деление на ноль, возникающее при вычислении производного параметра от значения параметра по умолчанию, а не от значения, заданного при создании экземпляра, и определение макроса после `undef, принятое за повторное. Высокая точность Surelog достигается на трех правилах, проверка которых не требует приближений.

Таблица 18 — Точность и взаимная полнота SVAN и Surelog по правилам

Правило SVAN	Диагностика Surelog	Точность, %		Истинные находки Surelog		Истинные находки SVAN	
		SVAN	Surelog	всего	найдено SVAN	всего	найдено Surelog
SYNTH13	UHDM_NON_SYNTHESIZABLE	97,2	100	303	242 (80 %)	921	262 (28 %)
PP02	PP_MULTIPLY_DEFINED_MACRO	—	80,0	4	0	0	—
ARITH02	UHDM_DIVIDE_BY_ZERO	—	0	0	—	0	—
Всего		97,2	99,5	307	242 (79 %)	921	262 (28 %)

Таблица 19 — Точность и взаимная полнота SVAN и Surelog по проектам

Проект	Срабатываний		Точность, %		Взаимная полнота, %	
	SVAN	Surelog	SVAN	Surelog	найдено SVAN	найдено Surelog
PicoRV32	1	0	100	—	—	0
SCR1	213	149	97,7	100	96	94
VeeR EH1	154	110	90,3	100	73	28
VeeR EL2	123	43	100	100	59	27
Ibex	38	0	100	—	—	0
BlackParrot	58	1	100	0	—	0
CVA6	443	89	98,0	98,9	53	5
Всего	1030	392	97,2	99,5	79	28

Полнота. Пропуски разработанного инструмента по правилу SYNTH13 (61 находка) образуют четыре класса конструкций, которые детектор не относит к несинтезируемым: системные задачи вывода (`$display`, `$write`, `$fwrite`), параллельные утверждения и объявления свойств, оператор `wait` и импортируемые функции DPI. Еще 74 находки Surelog относятся к модулям, не вошедшим в развернутую модель. О четырех повторных определениях макросов проекта CVA6 разработанный инструмент не сообщает, поскольку диагностика `-wredef`-макро выдается только при определении с иным телом, а здесь макросы определены одинаково в двух файлах. Surelog, в свою очередь, не сообщает о системных задачах `$fatal` и `$error`, функции `$sformatf` и блоках `initial`, составляющих основную часть находок разработанного инструмента.

6.7.6 Сравнение с Yosys

Yosys является инфраструктурой логического синтеза, поэтому сравнение выполнено на синтезируемой части проектов. Для этого сравнения детекторы правил разработанного инструмента запускались отдельно на тех же исходных текстах, что и Yosys (для пяти проектов синтезируемая часть совпадает с полным набором исходных текстов). По документации сообщениям Yosys соответствуют 17 правил набора, на синтетических тестах подтверждены 12. Четыре из них проявляются у Yosys не как предупреждения, а как ошибки синтеза, прерывающие

обработку. На проектах тестового набора сработали три вида сообщений Yosys (таблица 20): извлечение автоматов (1934 автомата у разработанного инструмента и 1069 у Yosys), защелки (13 срабатываний разработанного инструмента и 91 сообщение Yosys) и сигналы без источника (36 и 384 соответственно). По правилам SYNTH13 и PORT13 сработал только разработанный инструмент (440 и 61 срабатывание). Оценка точности Yosys по всем 1544 сообщениям составляет **92,5 %**. Результаты сравнения по правилам приведены в таблице 20, а результаты извлечения автоматов по проектам в таблице 21.

Таблица 20 — Точность и взаимная полнота SVAN и Yosys по правилам

Правило SVAN	Диагностическое сообщение Yosys об обнаружении	Точность, %		Истинные находки Yosys		Истинные находки SVAN	
		SVAN	Yosys	всего	найдено SVAN	всего	найдено Yosys
FSM01	конечного автомата	91,7	89,1	953	953 (100 %)	1774	885 (50 %)
LATCH01	защелки	100	100	—	—	—	—
SIGNAL07, SIGNAL16	сигнала без источника	100	100	—	—	—	—

Сообщения Yosys о защелках и сигналах без источника не привязаны к исходному тексту, поэтому взаимная полнота по ним не оценивалась.

Точность. Автоматы обоих средств размечены по критерию автомата раздела 6.3. Из 205 размеченных автоматов Yosys ложны 16. Из них 13 (12 в OpenC910 и один в OpenTitan) являются регистрами, которые загружаются константами по внешним условиям и не зависят от собственного значения. Штатный проход fsm_detect такие регистры не отсеивает, тогда как фильтр зависимости разработанного инструмента их отбрасывает. Остальные три ложных автомата проекта OpenTitan извлечены обоими средствами. Поскольку все 13 размеченных автоматов, найденных только Yosys, ложны, эта доля перенесена на остальные такие автоматы (их состав разобран ниже). Автоматы Yosys, совпадающие с размеченными автоматами разработанного инструмента, оценивались по разметке этих автоматов. Регистры выбора, условия загрузки которых зависят от текущего значения (например, port_sel_r и wb_instr_type_q), истинны и извлекаются обоими средствами. Оценки точности извлечения близки: **91,7 %** у разработанного инструмента и **89,1 %** у Yosys. Сообщения Yosys о защелках (все 91) и о сигналах без источника (100 размеченных из 384) истинны, однако выдаются

на каждый бит: 384 сообщения о сигналах без источника относятся к одному входному порту, не подключенному в экземплярах одного модуля проекта BlackParrot, а 68 сообщений о защелках в PicoRV32 относятся к трем сигналам (оператор с атрибутом `full_case`, раздел 6.6.3).

Таблица 21 — Точность и взаимная полнота SVAN и Yosys по проектам

Проект	Автоматов		Точность, %		Взаимная полнота, %	
	SVAN	Yosys	SVAN	Yosys	найдено SVAN	найдено Yosys
AES	4	4	100	100	100	100
PicoRV32	2	2	100	50,0	100	50
SCR1	26	8	100	75,0	100	23
VeeR EH1	15	4	100	100	100	27
VeeR EL2	12	5	100	100	100	42
Ibex	6	5	100	80,0	100	67
BlackParrot	18	18	100	100	100	100
CVA6	25	9	96,0	100	100	36
OpenC910	1201	432	92,0	81,9	100	29
OpenTitan	625	582	89,8	94,1	100	91
Vcero	1934	1069	91,7	89,1	100	50

Полнота. Из 1934 автоматов разработанного инструмента и 1069 автоматов Yosys общими по имени регистра состояния являются 971. Остальные 98 автоматов Yosys являются регистрами выбора, режимов и кодов операций того же вида, что и размеченные ложные автоматы Yosys: 78 в OpenC910 (48 экземпляров регистра `data_sel` модуля `ct_ciu_snb_sab_entry`, 28 регистров `is_dis*_sel` и два регистра `special_ex1_func`), 16 в OpenTitan (например, `reset_cause_q` и `io_mode_outclk`) и четыре в PicoRV32, Ibex и SCR1. Поэтому разработанный инструмент находит все 953 истинных автомата Yosys, а Yosys находит 885 из 1774 истинных автоматов разработанного инструмента (**50 %**): от 23 % на SCR1 до всех автоматов на AES и BlackParrot. Штатный проход `fsm_detect` отвергает автоматы, у которых логика следующего состояния вынесена в отдельный процедурный блок либо в другой модуль, что и устраняют доработки главы 3.

6.7.7 Сводные результаты

Применимость. Результаты сравнения с каждым из открытых средств сведены в таблице 22. Разработанный инструмент и Yosys завершили анализ всех десяти проектов. Verilator и Verible завершили анализ девяти проектов, svlint завершил анализ шести проектов, Surelog завершил анализ пяти. Отказы и частичное завершение с ошибками разбора отмечены в таблице 26.

Таблица 22 — Результаты сравнения с открытыми средствами

Показатель	Verilator	svlint	Verible	Surelog	Yosys
Проектов завершено (из 10)	10 / 9	10 / 6	10 / 9	10 / 5	10 / 10
Сопоставлено правил набора	33	5	5	3	12
Из них со срабатываниями	22 / 25	4 / 3	2 / 4	1 / 3	6 / 4
Срабатываний	5723 / 4964	1097 / 1226	167 / 184	1030 / 392	1934 / 1069*
Точность, %	94,3 / 38,8	97,8 / 45,9	100 / 34,2	97,2 / 99,5	91,7 / 89,1*
Взаимная полнота, %	62 / 25	85 / 14	80 / 23	79 / 28	100 / 50*

В ячейке первое число относится к разработанному инструменту, второе к сравниваемому средству.

Точность оценена относительно сопоставленных правил набора. * Для Yosys приведены число извлеченных автоматов (FSM01), точность и взаимная полнота их извлечения. Взаимная полнота: первое число равно доле истинных находок средства, найденных разработанным инструментом, второе равно доле истинных находок разработанного инструмента, найденных средством.

Точность. Точность разработанного инструмента в сравнениях составляет от **91,7** до **100 %**. Она близка к точности Surelog (**99,5 %**) и Yosys (**89,1 %** при извлечении автоматов) и выше точности Verilator, svlint и Verible (**38,8**, **45,9** и **34,2 %**). По всем 42 правилам, которые проверяются хотя бы одним открытым средством, разработанный инструмент дал 5975 срабатываний по 26 правилам. Оценка точности по 4043 размеченным срабатываниям составляет **94,5 %** (95 %-й доверительный интервал 87,2–96,1 %), ложные срабатывания относятся к классам раздела 6.2.

Полнота. Разработанный инструмент находит от **62** до **100 %** истинных находок каждого открытого средства, а открытые средства находят от **14** до **50 %** его находок. На синтетических тестах разработанный инструмент обнаруживает нарушения всех 58 правил, сопоставленных открытым средствам, и не дает срабатываний на парных тестах без нарушений. Открытые средства обнаруживают нарушения от 3 из 16 (Surelog) до 33 из 50 (Verilator) правил с документированным аналогом. Пропуски разработанного инструмента на проектах тестового набора

образуют небольшое число классов, каждый из которых воспроизведен на минимальном тесте. Часть пропусков следует из определений правил (исключения для операторов `unique case`, операндов типа `int` и перечислимых селекторов), остальные связаны с реализацией детекторов.

Ресурсоемкость. Замеры времени и памяти сведены в таблице 23. Синтаксические средства Verible и svlint быстрее разработанного инструмента по максимуму времени более чем на порядок. Полный набор проверок разработанного инструмента медленнее Verilator по медиане в 4 раза, по максимуму в 12 раз и требует больше памяти (до 12,7 ГБ), оставаясь экономнее Surelog (до 18,5 ГБ). Анализ автоматов сопоставим по времени с синтезом в Yosys на малых проектах и в 2,5–3 раза медленнее на крупнейших.

Таблица 23 — Ресурсоемкость средств по десяти проектам (медиана / максимум)

Средство	Время	Память
Verible	0,2 с / 1,0 с	12 МБ / 73 МБ
Verilator	1,9 с / 46 с	0,2 ГБ / 3,0 ГБ
svlint	5,9 с / 34 с	64 МБ / 0,6 ГБ
Surelog	11 с / 145 с	1,4 ГБ / 18,5 ГБ
Yosys	8,1 с / 27 мин	0,4 ГБ / 5,7 ГБ
SVAN, детекторы правил	7,8 с / 9,1 мин	1,4 ГБ / 12,7 ГБ
SVAN, анализ автоматов	8,2 с / 67 мин	0,5 ГБ / 8,1 ГБ

Для запусков, завершившихся отказом, учтено время до отказа. Замеры по проектам приведены в приложении В, для средств анализа автоматов SVAN в таблице 7.

Итоги. Из 118 правил набора 42 правила проверяются хотя бы одним открытым средством. Ни одно из рассмотренных средств не проверяет правила модели подсистемы питания (UPF) и правила, использующие библиотеки ячеек Liberty, и не выполняет поиск тупиковых и недостижимых состояний и взаимоблокировок автоматов. Разработанный инструмент охватывает эту область.

6.8 Внедрение результатов

Реализованная система статического анализа SVAN внедрена в научно-исследовательские и учебные проекты Института системного программирования им. В.П. Иванникова РАН, а также в опытно-конструкторские проекты

АО «МНТЦ МИЭТ» для верификации описаний цифровой аппаратуры на ранних этапах проектирования. Разработка выполнена в рамках опытно-конструкторской работы «Разработка системы статического анализа для языка описания аппаратуры» (шифр «САПР-Анализ»). На основе результатов работы подготовлена дополнительная профессиональная программа повышения квалификации «Применение системы статического анализа SVAN для языка описания аппаратуры», утвержденная директором ИСП РАН 2 сентября 2025 г. Свидетельства о государственной регистрации программ для ЭВМ приведены в приложении Г.

6.9 Выводы по главе

Проведена экспериментальная оценка системы SVAN на десяти открытых проектах цифровых СБИС суммарным объемом свыше 1,8 млн строк кода. Точность полного набора детекторов правил и диагностик компилятора без шумного правила PORT04 составила от **90,5** до **100 %** по проектам и **96,3 %** в целом (таблица 5). Разработчикам открытых проектов сообщено о 24 нарушениях. Разработчики подтвердили **13** нарушений, из них **10** уже исправлены, три ожидают исправления. Нагрузочный тест масштабируемости успешно выполнен на проекте NVDLA объемом около 1,5 млн строк. Средства анализа конечных автоматов выявили все четыре вида дефектов графа автомата на 49 синтетических и регрессионных тестах. На проектах тестового набора точность анализа автоматов по всем диагностикам и извлеченным автоматам составила **95,5 %**, все ложные срабатывания относятся к извлечению автоматов (14 из 255 размеченных автоматов). Подсистема интерпретации UPF и анализа модели питания проверена на 830 синтетических тестах, 20 проектах проверки модели и открытых проектах с UPF-описаниями. Анализатор SDF проверен на 94 модульных тестах, включающих все примеры раздела 32 стандарта IEEE 1800-2023, детекторы на основе библиотек Liberty проверены на синтетических тестах. Механизм сопоставителей узлов АСД сократил полное время работы анализатора в **1,2–2,4** раза при одиночном запуске детектора, всего на сопоставителях реализовано 18 детекторов.

В сравнениях с другими инструментами точность разработанного инструмента вычислялась только по правилам, сопоставленным правилам сравниваемого инструмента, и на проектах, анализ которых завершили оба инструмента,

поэтому она отличается от точности полного набора проверок. Коммерческий инструмент SpyGlass завершил анализ шести проектов из десяти. На этих проектах точность без шумных правил PORT04 и W287b составила **95,7 %** у разработанного инструмента и **79,8 %** у SpyGlass, с шумными правилами **99,4** и **93,9 %** соответственно. Разработанный инструмент находит **49 %** истинных находок SpyGlass без учета правила W287b, а SpyGlass находит **29 %** находок разработанного инструмента без учета правила PORT04. Все автоматы SpyGlass более чем с двумя состояниями найдены разработанным инструментом, тогда как SpyGlass обнаружил **7 %** автоматов, извлеченных разработанным инструментом. Сравнение с инструментами VCS и SpyGlass показало, что разработанный инструмент поддерживает стандарт UPF IEEE 1801-2018 полнее.

Сравнение с пятью открытыми средствами выполнено по 42 правилам, которые проверяются хотя бы одним из них. Точность разработанного инструмента по этим правилам составила **94,5 %**. В сравнении с Surelog, который проверяет три правила набора, точность обоих средств близка (**97,2** и **99,5 %**). Точность Verilator, svlint и Verible относительно сопоставленных правил набора ниже (**38,8**, **45,9** и **34,2 %**). Точность извлечения автоматов у разработанного инструмента и Yosys близка (**91,7** и **89,1 %**). Разработанный инструмент находит от **62** до **100 %** истинных находок каждого открытого средства, а открытые средства находят от **14** до **50 %** его находок. Классы пропусков детекторов установлены и воспроизведены на минимальных тестах. Ни одно из рассмотренных открытых средств не проверяет модель подсистемы питания и правила на основе библиотек Liberty и не выполняет поиск тупиковых и недостижимых состояний и взаимоблокировок автоматов. На рабочей станции детекторам правил разработанного инструмента требуется до 12,7 ГБ памяти и до 9,1 мин.

Результаты главы соответствуют четвертому положению, выносимому на защиту.

Заключение

Основные результаты работы заключаются в следующем.

1. Разработан метод статического анализа описаний цифровой аппаратуры на основе внутреннего представления проекта и набора из 118 правил для раннего выявления структурных и семантических дефектов и нарушений свойств безопасности без логического синтеза и имитационного моделирования.
2. Разработан метод извлечения и статического анализа конечных автоматов из RTL-описаний на основе сети логических элементов и графового представления, выявляющий независимо от стиля кодирования недостижимые и тупиковые состояния, мертвые переходы, взаимоблокировки и взаимодействующие автоматы.
3. Разработан метод интеграции сопутствующих проектных форматов в статический анализ на основе единой информационной модели, связывающей описание аппаратуры с объектами Liberty, UPF и SDF, для анализа схем с библиотечными ячейками, модели подсистемы питания и SDF-описаний.
4. На основе предложенных методов реализованы программные средства статического анализа описаний цифровой аппаратуры, включающие детекторы правил разработанного набора, средства анализа конечных автоматов и анализатор модели подсистемы питания. Экспериментальная оценка на открытых проектах цифровых СБИС и сравнение с коммерческим и открытыми инструментами статического анализа показали эффективность разработанных методов.

К дальнейшим направлениям развития относятся расширение набора правил и поддержка стандартов кодирования (например, STARC), поддержка версий стандарта UPF, отличных от IEEE 1801-2018, завершение поддержки языка VHDL и исследование гибридных схем, сочетающих детерминированный статический анализ с методами машинного обучения.

Таким образом, поставленная цель работы достигнута.

В заключение автор выражает глубокую благодарность научному руководителю кандидату технических наук Р. А. Бучацкому за поддержку, помощь и научное руководство, а также коллективам отделов компиляторных техноло-

гий и технологий программирования Института системного программирования им. В.П. Иванникова РАН и соавторам совместных работ за плодотворное сотрудничество и обсуждение результатов.

Список сокращений и условных обозначений

CDC	Clock Domain Crossing, пересечение доменов синхронизации
DDG	Data Dependence Graph, граф зависимостей по данным
DFA	Data Flow Analysis, анализ потоков данных
DPI	Direct Programming Interface, интерфейс прямого вызова программного кода
FSM	Finite State Machine, конечный автомат
HDL	Hardware Description Language, язык описания аппаратуры
IFT	Information Flow Tracking, отслеживание потоков информации
LLM	Large Language Model, большая языковая модель
ML	Machine Learning, машинное обучение
PLA	Programmable Logic Array, программируемая логическая матрица
PLI	Programming Language Interface, программный интерфейс языка
PMP	Physical Memory Protection, физическая защита памяти
PST	Power State Table, таблица состояний питания
RACL	Register Access Control List, список управления доступом к регистрам
RDC	Reset Domain Crossing, пересечение доменов сброса
RTL	Register Transfer Level, уровень регистровых передач
RTLIL	RTL Intermediate Language, промежуточное представление уровня регистровых передач
SARIF	Static Analysis Results Interchange Format, формат обмена результатами статического анализа
SCC	Strongly Connected Component, сильно связанная компонента
SDC	Synopsys Design Constraints, формат проектных ограничений
SDF	Standard Delay Format, стандартный формат задержек
SMT	Satisfiability Modulo Theories, выполнимость формул в теориях
SVA	SystemVerilog Assertions, утверждения SystemVerilog
TLB	Translation Lookaside Buffer, буфер ассоциативной трансляции адресов
TSCC	Terminal Strongly Connected Component, терминальная сильно связанная компонента
UDP	User Defined Primitive, пользовательский примитив
UHDM	Universal Hardware Data Model, универсальная модель аппаратных данных

UPF	Unified Power Format, унифицированный формат описания питания
UVM	Universal Verification Methodology, универсальная методология верификации
VHDL	VHSIC Hardware Description Language, язык описания аппаратуры
VPI	Verilog Procedural Interface, процедурный интерфейс Verilog
АСД	абстрактное синтаксическое дерево (Abstract Syntax Tree)
ПЗУ	постоянное запоминающее устройство
САПР	система автоматизированного проектирования
СБИС	сверхбольшая интегральная схема
СнК	система на кристалле

Словарь терминов

Уровень регистровых передач (RTL) : уровень абстракции описания цифровой схемы в терминах регистров, сигналов и логических операций между ними в пределах одного такта.

Статический анализ : анализ описания, проводимый без его выполнения, логического синтеза и имитационного моделирования.

Дефект проектирования : свойство RTL-описания, которое приводит или может привести к некорректному функционированию схемы, к расхождению ее поведения при имитационном моделировании и логическом синтезе либо к нарушению свойств безопасности.

Внутреннее представление : совокупность структур, построенных системой анализа по описанию, над которыми выполняется проверка правил: уровни, которые строит компилятор (дерево разбора, абстрактное синтаксическое дерево, предварительная и развернутая модели RTL-описания), внутренние структуры, граф конечного автомата и единая информационная модель.

Развернутая модель : уровень внутреннего представления, в котором вычислены значения параметров, созданы экземпляры модулей, разрешены иерархические имена и раскрыты блоки генерации (раздел 2.1.1).

Правило : проверяемое свойство описания, то есть условие, которому должно удовлетворять описание или его части.

Структурное правило : правило, нарушение которого определяется по структуре описания (наличию, расположению, атрибутам и взаимосвязям конструкций) без вычисления значений сигналов и анализа достижимости путей.

Нарушение правила : конкретное несоответствие описания свойству, заданному правилом. Нарушение может относиться к отдельной конструкции, к отсутствию ожидаемой конструкции или к отношению между несколькими конструкциями.

Детектор : алгоритм, проверяющий по внутреннему представлению описания заданное правилом свойство и сообщаящий о каждом обнаруженном нарушении диагностическим предупреждением с локализацией нарушения в исходном тексте.

Сопоставитель : выражение предметно-ориентированного языка, задающее условие на узел абстрактного синтаксического дерева, его атрибуты или связанные с ним узлы (раздел 2.8).

Шумное правило : правило с большим числом срабатываний, которое отражает практику проектирования, а не дефект (например, PORT04).

Семантический шаблон : заданная структура конструкции, соответствие которой определяет ее функциональное назначение (например, строки таблицы состояний пользовательского примитива, задающие переходы защелки или D-триггера).

Конечный автомат : модель последовательностной логики, описывающая поведение через конечное множество состояний, переходов и выходов.

Взаимодействующие конечные автоматы : автоматы, для которых сигнал, формируемый одним автоматом, достигает логики переходов другого по пути, содержащему произвольную комбинационную логику и не более одного последовательностного элемента.

Сеть логических элементов (netlist) : описание схемы в терминах логических элементов (триггеров, мультиплексоров, логических операций) и соединений между ними, получаемое трансляцией RTL-описания средствами логического синтеза. В работе используется для извлечения конечных автоматов (глава 3).

Логическая сеть : ориентированный граф зависимостей между сигналами, который строится по развернутой модели без логического синтеза на уровне объявлений и вхождений переменных (раздел 2.6).

Подсистема питания : совокупность элементов схемы, обеспечивающих распределение питания и управление им (сети и домены питания, ключи питания, ячейки изоляции и преобразования уровня напряжения, регистры сохранения состояния). Ее описание (power intent) задается в формате UPF.

Единая информационная модель : часть внутреннего представления, связывающая узлы развернутой модели с объектами сопутствующих описаний: ячейками библиотек Liberty, моделью подсистемы питания, полученной интерпретацией UPF-описания, и записями SDF-описания.

Защелка (latch) : элемент памяти, прозрачный по уровню управляющего сигнала, в отличие от триггера, срабатывающего по фронту.

Список литературы

1. Система статического анализа для языка описания аппаратуры SystemVerilog [Текст] / Я. А. Чуркин, Р. А. Бучацкий [и др.] // Труды Института системного программирования РАН. — 2025. — Т. 37, № 1. — С. 7—40.
2. Статический анализ модели подсистемы питания цифровой аппаратуры [Текст] / А. В. Егиазарян, Я. А. Чуркин [и др.] // Труды Института системного программирования РАН. — 2026. — Т. 38, № 3. — С. 45—70.
3. Автоматическая проверка синтезируемости описаний аппаратуры [Текст] / А. А. Шевцов, Я. А. Чуркин [и др.] // Труды Института системного программирования РАН. — 2026. — Т. 38, № 4, ч. 2. — С. 45—66.
4. Чуркин, Я. А. Статический анализ описаний цифровой аппаратуры: методы, инструменты и перспективы [Текст] / Я. А. Чуркин, Р. А. Бучацкий // Программные продукты и системы. — 2026. — Т. 39, № 2. — С. 264—274.
5. SVAN: Static Analysis System for SystemVerilog Hardware Descriptions [Text] / Y. Churkin, R. Buchatskiy, [et al.] // 2026 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (US-BEREIT). — Yekaterinburg, Russian Federation : IEEE, 2026. — P. 1—4.
6. SVAN-FSM: Static Analysis of Finite State Machines in SystemVerilog [Text] / A. Viazovtsev, Y. Churkin, [et al.] // 2026 IEEE International Test Conference in Asia (ITC-Asia). — Ningbo, China : IEEE, 2026. — P. 1—6.
7. Китаев, К. Н. Разработка средств спецификации и верификации подсистемы питания в статическом анализаторе для языка описания аппаратуры SystemVerilog [Текст] / К. Н. Китаев, Я. А. Чуркин // Труды 67-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. — М. : Физматкнига, 2025. — С. 31—32.
8. Трунов, В. В. Разработка предметно-ориентированного языка для сопоставления узлов абстрактного синтаксического дерева языка описания аппаратуры SystemVerilog [Текст] / В. В. Трунов, Я. А. Чуркин // Труды 67-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. — М. : Физматкнига, 2025. — С. 42—43.
9. Свидетельство о государственной регистрации программы для ЭВМ № 2026615835 Рос. Федерация. Система статического анализа для языка описания аппаратуры [Текст] / Я. А. Чуркин [и др.] ; Российская Федерация,

- от имени которой выступает Министерство промышленности и торговли Российской Федерации; АО «МНТЦ МИЭТ». — № 2025692899 ; заявл. 26.11.2025 ; опубл. 03.03.2026, Бюл. № 3 (Рос. Федерация).
10. *Свидетельство о государственной регистрации программы для ЭВМ № 2026683339 Рос. Федерация*. Система статического анализа описаний аппаратуры на языке VHDL [Текст] / Я. А. Чуркин [и др.] ; Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. — № 2026681213 ; заявл. 09.07.2026 ; опубл. 30.07.2026 (Рос. Федерация).
 11. *IEEE*. IEEE Standard for Verilog Hardware Description Language [Text] / IEEE. — 2006. — IEEE Std 1364-2005.
 12. *IEEE*. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language [Text] / IEEE. — 2018. — IEEE Std 1800-2017.
 13. *IEEE*. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language [Text] / IEEE. — 2024. — IEEE Std 1800-2023.
 14. *Sutherland, S.* Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them [Text] / S. Sutherland, D. Mills. — Springer, 2010. — 218 p.
 15. *Nielson, F.* Principles of Program Analysis [Text] / F. Nielson, H. R. Nielson, C. Hankin. — Springer, 1999. — 452 p.
 16. *MITRE Corporation*. CWE VIEW 1194: Hardware Design [Text] / MITRE Corporation. — URL: <https://cwe.mitre.org/data/definitions/1194.html> (visited on 07/12/2026).
 17. NuSMV: A new symbolic model checker [Text] / A. Cimatti [et al.] // International Journal on Software Tools for Technology Transfer. — 2000. — Vol. 2. — P. 410—425.
 18. Qihe: A general-purpose static analysis framework for Verilog [Text] / Q. Chen [et al.]. — 2026. — arXiv preprint arXiv:2601.11408.
 19. *Kildall, G. A.* A unified approach to global program optimization [Text] / G. A. Kildall // Proc. 1st ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL). — 1973. — P. 194—206.
 20. *Kam, J. B.* Monotone data flow analysis frameworks [Text] / J. B. Kam, J. D. Ullman // Acta Informatica. — 1977. — Vol. 7. — P. 305—317.
 21. *Cousot, P.* Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints [Text] / P. Cousot,

- R. Cousot // Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL). — 1977. — P. 238—252.
22. Dependence graphs and compiler optimizations [Text] / D. J. Kuck [et al.] // Proc. 8th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). — 1981. — P. 207—218.
 23. Program slicing of hardware description languages [Text] / E. M. Clarke [et al.] // Proc. Correct Hardware Design and Verification Methods (CHARME), Lecture Notes in Computer Science, vol. 1703. — 1999. — P. 298—313.
 24. *Hymans, C.* Checking safety properties of behavioral VHDL descriptions by abstract interpretation [Text] / C. Hymans // Proc. 9th International Static Analysis Symposium (SAS), Lecture Notes in Computer Science, vol. 2477. — 2002.
 25. *Hymans, C.* Design and implementation of an abstract interpreter for VHDL [Text] / C. Hymans // Proc. Correct Hardware Design and Verification Methods (CHARME 2003). LNCS, vol. 2860. — 2003. — P. 263—269.
 26. *Schlickling, M.* Semi-automatic derivation of timing models for WCET analysis [Text] / M. Schlickling, M. Pister // Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). — 2010.
 27. *King, J. C.* Symbolic execution and program testing [Text] / J. C. King // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385—394.
 28. *Ryan, K.* Countering the path explosion problem in the symbolic execution of hardware designs [Text] / K. Ryan, C. Sturton. — 2023. — arXiv:2304.05445.
 29. *Ryan, K.* SylQ-SV: Scaling symbolic execution of hardware designs with query caching [Text] / K. Ryan, J. Rogers, C. Sturton // Proc. ASPLOS. Vol. 3. — 2025. — P. 195—211.
 30. *Ardeshiricham, A.* RTLIFT: Register transfer level information flow tracking [Text] / A. Ardeshiricham, W. Hu, R. Kastner // Proc. Design, Automation and Test in Europe (DATE). — 2017. — P. 1691—1696.
 31. *Ryan, K.* SEIF: Augmented symbolic execution for information flow in hardware designs [Text] / K. Ryan, M. Gregoire, C. Sturton // Proc. Hardware and Architectural Support for Security and Privacy (HASP). — 2023.
 32. A hardware design language for timing-sensitive information-flow security [Text] / D. Zhang [et al.] // Proc. 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). — 2015. — P. 503—516.

33. Caisson: A hardware description language for secure information flow [Text] / X. Li [et al.] // Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). — 2011.
34. Sapper: A language for hardware-level security policy enforcement [Text] / X. Li [et al.] // Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). — 2014.
35. *Ardeshiricham, A.* Clepsydra: Modeling timing flows in hardware designs [Text] / A. Ardeshiricham, W. Hu, R. Kastner // Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD). — 2017.
36. VeriSketch: Synthesizing secure hardware designs with timing-sensitive information flow properties [Text] / A. Ardeshiricham [et al.] // Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS). — 2019.
37. QIF-Verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment [Text] / X. Guo [et al.] // Proc. IEEE International Symposium on Hardware Oriented Security and Trust (HOST). — 2019. — P. 91—100.
38. *Ardeshiricham, A.* Verification and Synthesis of Information Flow Secure Hardware Designs [Text] : PhD thesis / Ardeshiricham A. — University of California, San Diego, 2020. — URL: <https://escholarship.org/uc/item/1669k7m9>.
39. *Смолов, С. А.* Обзор методов извлечения моделей из HDL-описаний [Текст] / С. А. Смолов // Труды ИСП РАН. — 2015. — Т. 27, № 1. — С. 97—124.
40. *Giomi, J. C.* Finite state machine extraction from hardware description languages [Text] / J. C. Giomi // Proc. Eighth International ASIC Conference. — 1995. — P. 353—357.
41. *Смолов, С. А.* Метод построения расширенных конечных автоматов по HDL-описанию на основе статического анализа кода [Текст] / С. А. Смолов, А. С. Камкин // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. — 2015. — № 1. — С. 60—73.
42. FSMx: Finite state machine extraction from flattened netlist with application to security [Text] / R. Kibria [et al.] // Proc. IEEE 40th VLSI Test Symposium (VTS). — 2022.
43. *Kibria, R.* FSMx-Ultra: Finite state machine extraction from gate-level netlist for security assessment [Text] / R. Kibria, F. Farahmandi, M. Tehranipoor // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. — 2023. — Vol. 42, no. 11. — P. 3613—3627.

44. RTL-FSMx: Fast and accurate finite state machine extraction at the RTL for security applications [Text] / R. Kibria [et al.] // Proc. IEEE International Test Conference (ITC). — 2022. — P. 165—174.
45. *Meade, T.* Netlist reverse engineering for high-level functionality reconstruction [Text] / T. Meade, S. Zhang, Y. Jin // Proc. 21st Asia and South Pacific Design Automation Conference (ASP-DAC). — 2016. — P. 655—660.
46. On the difficulty of FSM-based hardware obfuscation [Text] / M. Fyrbiak [et al.] // IACR Transactions on Cryptographic Hardware and Embedded Systems. — 2018. — No. 3. — P. 293—330.
47. *IEEE.* IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems [Text] / IEEE. — 2019. — IEEE Std 1801-2018 (UPF).
48. *Khondkar, P.* Low-Power Design and Power-Aware Verification [Text] / P. Khondkar. — Cham : Springer, 2018.
49. *Sharafinejad, R.* UPF-based formal verification of low power techniques in modern processors [Text] / R. Sharafinejad, B. Alizadeh, M. Fujita // Proc. IEEE 33rd VLSI Test Symposium (VTS). — 2015. — P. 1—6.
50. *IEEE.* IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process [Text] / IEEE. — 2001. — IEEE Std 1497-2001.
51. FLAG: Finding line anomalies (in RTL code) with generative AI [Text] / B. Ahmad [et al.] // ACM Transactions on Design Automation of Electronic Systems. — 2025. — Vol. 30, no. 6.
52. LintLLM: An open-source verilog linting framework based on large language models [Text] / Z. Fang [et al.] // Proc. International Conference on Generative LR (ICGLR). — 2025. — P. 673—680.
53. AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs [Text] / Z. Yan [et al.]. — 2025. — arXiv:2411.14436.
54. *Tsai, Y. D.* RTLFixer: Automatically fixing RTL syntax errors with large language model [Text] / Y. D. Tsai, M. Liu, H. Ren // Proc. Design Automation Conference (DAC). — 2024. — P. 1—6.
55. VeriDebug: A unified LLM for verilog debugging via contrastive embedding and guided correction [Text] / N. Wang [et al.] // Proc. International Conference on Computer-Aided Design (ICLAD). — 2025.
56. *Synopsys.* VC SpyGlass: static and formal verification [Text] / Synopsys. — URL: <https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass.html> (visited on 11/02/2025).

57. *Cadence*. JasperGold formal and static verification [Text] / Cadence. — URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html (visited on 11/02/2025).
58. *Aldec*. ALINT-PRO: functional verification / design rule checking [Text] / Aldec. — URL: https://www.aldec.com/en/products/functional_verification/alint-pro (visited on 11/02/2025).
59. *Siemens EDA*. Questa One smart verification solution [Text] / Siemens EDA. — URL: <https://eda.sw.siemens.com/en-US/ic/questa-one/> (visited on 11/02/2025).
60. *Snyder, W.* Verilator: the fastest Verilog/SystemVerilog simulator [Text] / W. Snyder. — URL: <https://github.com/verilator/verilator> (visited on 11/02/2025).
61. *Dalance*. svlint: SystemVerilog linter [Text] / Dalance. — URL: <https://github.com/dalance/svlint> (visited on 11/02/2025).
62. *Popoloski, M.* slang – SystemVerilog Language Services [Text] / M. Popoloski. — URL: <https://github.com/MikePopoloski/slang> (visited on 11/02/2025).
63. *PASCAL Lab*. Qihe: a static analysis platform for Verilog [Text] / PASCAL Lab. — URL: <https://qihe.pascal-lab.net/> (visited on 11/02/2025).
64. *Takamaeda-Yamazaki, S.* Pyverilog: A python-based hardware design processing toolkit for verilog HDL [Text] / S. Takamaeda-Yamazaki // Proc. Applied Reconfigurable Computing (ARC). Vol. 9040. — 2015. — P. 451—460.
65. *CHIPS Alliance*. Surelog: SystemVerilog 2017 pre-processor, parser, elaborator, UHDM compiler [Text] / CHIPS Alliance. — URL: <https://github.com/chipsalliance/Surelog> (visited on 11/02/2025).
66. *Dargelas, A.* Universal hardware data model [Text] / A. Dargelas, S. Korczynski // Proc. Workshop on Open-Source EDA Technology (WOSET). — 2020.
67. MLIR: Scaling compiler infrastructure for domain specific computation [Text] / C. Lattner [et al.] // Proc. International Symp. on Code Generation and Optimization (CGO). — 2021. — P. 2—14.
68. *CHIPS Alliance*. Verible: SystemVerilog parser, style-linter, and language server [Text] / CHIPS Alliance. — URL: <https://github.com/chipsalliance/verible> (visited on 11/02/2025).
69. *Williams, S.* Icarus Verilog [Text] / S. Williams, contributors. — URL: <https://github.com/steveicarus/iverilog> (visited on 02/01/2026).
70. *Wolf, C.* Yosys – a free verilog synthesis suite [Text] / C. Wolf, J. Glaser // Proc. Austrochip. — 2013.

71. *tree-sitter*. tree-sitter-verilog: Verilog grammar for tree-sitter [Text] / tree-sitter, contributors. — URL: <https://github.com/tree-sitter/tree-sitter-verilog> (visited on 02/01/2026).
72. *Larumbe, G.* tree-sitter-systemverilog: SystemVerilog grammar for tree-sitter [Text] / G. Larumbe, contributors. — URL: <https://github.com/gmlarumbe/tree-sitter-systemverilog> (visited on 02/01/2026).
73. *Takemoto, T.* Joint activity for semiconductor R&D and role of STARC [Text] / T. Takemoto // International Symp. on VLSI Technology, Systems, and Applications. — 2001. — P. 7—10.
74. *Gueguen, J. P.* Applying the OpenMORE assessment program for IP cores [Text] / J. P. Gueguen, P. Bricaud // Proc. International Symp. on Quality Electronic Design (ISQED). — 2000. — P. 379—381.
75. *Hilderman, V.* Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware) [Text] / V. Hilderman, T. Baghi. — Avionics Communications, 2007. — 244 p.
76. *Tarjan, R. E.* Depth-first search and linear graph algorithms [Text] / R. E. Tarjan // SIAM Journal on Computing. — 1972. — Vol. 1, no. 2. — P. 146—160.
77. *LLVM Project.* Clang-Tidy: clang-based C++ linter tool, Extra Clang Tools documentation [Text] / LLVM Project. — URL: <https://clang.llvm.org/extra/clang-tidy/> (visited on 07/03/2026).
78. *Wang, T.-H.* Practical FSM analysis for Verilog [Text] / T.-H. Wang, T. Edsall // Proc. International Verilog HDL Conference and VHDL International Users Forum. — 1998. — P. 52—58.
79. AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs [Text] / A. Nahiyani [et al.] // Proc. 53rd ACM/EDAC/IEEE Design Automation Conference (DAC). — 2016. — P. 1—6.
80. A highly efficient method for extracting FSMs from flattened gate-level netlist [Text] / Y. Shi [et al.] // Proc. IEEE International Symposium on Circuits and Systems (ISCAS). — 2010. — P. 2610—2613.
81. *Moon, I.-H.* Abstraction-based livelock/deadlock checking for hardware verification [Text] / I.-H. Moon, K. Harer. — 2012. — U.S. Patent 8 302 044.
82. *Bryant, R. E.* Graph-based algorithms for Boolean function manipulation [Text] / R. E. Bryant // IEEE Transactions on Computers. — 1986. — Vol. 35, no. 8. — P. 677—691.

83. *Gansner, E. R.* An open graph visualization system and its applications to software engineering [Text] / E. R. Gansner, S. C. North // *Software: Practice and Experience*. — 2000. — Vol. 30, no. 11. — P. 1203—1233.
84. *Povik, M.* yosys-slang: SystemVerilog frontend for Yosys based on slang [Text] / M. Povik, contributors. — URL: <https://github.com/povik/yosys-slang> (visited on 02/01/2026).
85. *Wilson, E. B.* Probable inference, the law of succession, and statistical inference [Text] / E. B. Wilson // *Journal of the American Statistical Association*. — 1927. — Vol. 22, no. 158. — P. 209—212.
86. *Zelenov, S.* Automated generation of positive and negative tests for parsers [Text] / S. Zelenov, S. Zelenova // *Proc. International Workshop on Formal Approaches to Software Testing (FATES)*. — 2005. — P. 187—202.
87. *jsamarth.* GuitarPedalBoard: an FPGA project on guitar effects [Text] / jsamarth. — URL: <https://github.com/jsamarth/GuitarPedalBoard> (visited on 10/01/2026).

Список рисунков

1.1	Маршрут проектирования цифровой интегральной схемы и уровни абстракции описания	15
2.1	Внутреннее представление: уровни, которые строит компилятор, и внутренние структуры	35
3.1	Конвейер извлечения и статического анализа конечных автоматов . . .	68
3.2	Граф конечного автомата с отмеченными дефектами и соответствующее описание на SystemVerilog	71
3.3	Два экземпляра одного извлеченного автомата модуля <code>bp_uce</code> проекта BlackParrot с различными значениями параметров: выделенные состояния недостижимы в экземпляре <code>icache_uce</code> , но достижимы в <code>dcache_uce</code>	78
4.1	Передача сигналов между доменами с разными уровнями напряжения через преобразователи уровня напряжения (<code>low_to_high</code> и <code>high_to_low</code>)	84
4.2	Построение единой информационной модели из HDL-описания, библиотек Liberty, UPF- и SDF-описаний	85
4.3	Граница выключенного и включенного доменов питания: нарушение и корректная реализация	92
5.1	Структурная схема системы статического анализа SVAN	100
6.1	Логика следующего состояния и регистр состояния в разных экземплярах модулей, соединенных через порты (ядро VeeR EL2) . . .	140

Список таблиц

1	Уровни представления описания в средствах анализа	18
2	Сравнение открытых инструментов анализа и обработки RTL”=описаний с разработанной системой	28
3	Сопоставление инструментов анализа конечных автоматов	67
4	Источники единой информационной модели, используемые проверками	96
5	Срабатывания детекторов и диагностик компилятора системы SVAN на проектах с открытым исходным кодом, без правила PORT04	125
6	Срабатывания детекторов и диагностик компилятора на проекте CVA6 (без правила PORT04)	126
7	Результаты извлечения и анализа конечных автоматов на проектах с открытым исходным кодом (число диагностик), сквозное время анализа и пиковая память	128
8	Полное время работы анализатора с детекторами на обходчиках и на сопоставителях, с (в ячейке: обходчики / сопоставители, ускорение выделено полужирным)	132
9	Время и пиковая память разработанного инструмента и SpyGlass (время / память)	135
10	Точность и взаимная полнота SVAN и SpyGlass по проектам	136
11	Точность и взаимная полнота SVAN и SpyGlass по правилам	137
12	Точность и взаимная полнота SVAN и Verilator по правилам	144
13	Точность и взаимная полнота SVAN и Verilator по проектам	145
14	Точность и взаимная полнота SVAN и svlint по правилам	147
15	Точность и взаимная полнота SVAN и svlint по проектам	147
16	Точность и взаимная полнота SVAN и Verible по правилам	149
17	Точность и взаимная полнота SVAN и Verible по проектам	149
18	Точность и взаимная полнота SVAN и Surelog по правилам	150
19	Точность и взаимная полнота SVAN и Surelog по проектам	151
20	Точность и взаимная полнота SVAN и Yosys по правилам	152
21	Точность и взаимная полнота SVAN и Yosys по проектам	153
22	Результаты сравнения с открытыми средствами	154
23	Ресурсоемкость средств по десяти проектам (медиана / максимум) . . .	155

Приложение А

Каталог правил статического анализа

В настоящем приложении приведен полный каталог сформулированных в работе правил статического анализа: 118 правил, сгруппированных по 19 категориям, введенным в разделе 2.2 (там же приведен состав набора по языкам и подсистемам анализа). Для каждого правила указаны техника анализа, достаточная для его проверки, проверяемое свойство описания и последствия его нарушения. Правила, отмеченные знаком [†], проверяются штатными диагностиками компилятора SystemVerilog, остальные — детекторами модуля анализа или отдельными подсистемами, указанными обозначением техники (глава 5). Идентификаторы детекторов, их настраиваемые параметры и исключения, а также примеры нарушающих и корректных фрагментов кода приведены в эксплуатационной документации системы и в настоящем приложении опущены.

Техника анализа указана в квадратных скобках после идентификатора правила по реализации его детектора или диагностики компилятора. Обозначения: А — обход АСД (разделы 2.3 и 2.8), Т — классификация типов логики процедурных блоков (раздел 2.4.1), П — анализ путей выполнения в процедурных блоках (раздел 2.4), Ш — распознавание семантического шаблона (раздел 2.5), С — анализ на основе логической сети (раздел 2.6), Д — анализ потоков данных (раздел 2.7), И — учет установленных разрядов и источников сигналов (раздел 2.7.1), Л — сведения библиотек ячеек Liberty (раздел 4.5), КА — анализ конечных автоматов (глава 3), МП — анализ модели подсистемы питания (глава 4). Все техники опираются на обход АСД, поэтому знак А указан только у правил, для проверки которых его достаточно. Несколько знаков означают, что проверка использует техники совместно. Прочерк у правил категории `synс` означает, что их проверка выполняется отдельным анализатором.

Категория `arith`: арифметические операции

ARITH01 [А] — операция сдвига влево не должна приводить к потере старших битов: требуемая разрядность результата не должна превышать разрядность приемника.

ARITH02 [Д] — делитель в операциях `/` и `%` не должен принимать нулевое значение, иначе результат имитационного моделирования принимает неопределенное значение `X`. Возможные значения делителя оцениваются анализом потока данных и интервальным анализом диапазонов.

Категория **assign**: операции присваивания

ASSIGN01 [A] — правая часть присваивания не должна быть литералом, содержащим значение X: явное присваивание неопределенного значения, как правило, является ошибкой проектирования.

ASSIGN02 [T] — в последовательностном блоке `always` одному и тому же триггеру не должно присваиваться значение несколько раз (за исключением асинхронного сброса): поведение такого описания зависит от порядка выполнения присваиваний.

ASSIGN03 [T] — в последовательностной логике не должны использоваться блокирующие присваивания: в синхронных блоках они порождают состояние гонки и несоответствие результатов имитационного моделирования до и после синтеза.

ASSIGN04 [T, П] — в комбинационном блоке `always_comb` за неблокирующим присваиванием не должно следовать блокирующее: комбинационную логику следует описывать блокирующими присваиваниями, а смешение видов присваивания приводит к расхождению поведения модели до и после синтеза.

ASSIGN05 [A] — в пределах одной области видимости присваивание не должно переопределять предыдущее присваивание тому же сигналу (за исключением задания значения по умолчанию). Ветви `if/else` и `case` со взаимоисключающими условиями нарушением не считаются.

ASSIGN06[†] [A] — знаковые и беззнаковые переменные и константы не должны смешиваться в одном выражении, сравнении или присваивании.

Категория **case**: операторы **case**

CASE01 [A] — оператор `case` должен содержать ветвь `default`. Исключение составляют операторы с полным покрытием значений селектора и операторы с квалификаторами `unique/priority`.

CASE02 [A] — метки оператора `case` не должны дублироваться или перекрываться: часть таких меток заведомо недостижима.

CASE03 [A] — разрядность метки оператора `case` должна соответствовать разрядности селектора: несоответствие приводит к неполному или некорректному сравнению.

CASE04 [A] — оператор `case`, помеченный как `full_case` или снабженный квалификатором `unique/priority`, не должен содержать ветвь `default`: для таких операторов она избыточна и может маскировать ошибки покрытия.

CASE05 [A] — метки оператора, помеченного как `full_case`, должны покрывать все возможные значения селектора либо оператор должен содержать ветвь `default`: несоблюдение приводит к непреднамеренному выводу защелки и расхождению до и после синтеза.

CASE06 [A] — в операторе, помеченном как `parallel_case`, метки не должны перекрываться: перекрытие противоречит указанному свойству параллельности и приводит к расхождению до и после синтеза.

CASE07 [И] — значение третьего состояния (`z`) не должно использоваться в качестве метки оператора `case` (за исключением оператора `case inside`): при синтезе такое сравнение трактуется как ложное.

CASE08 [И] — в метках оператора `case` без модификаторов `z` и `x` не должны использоваться вещественные значения, а также значения `X`, `Z` и `?`.

CASE09 [И] — селектор операторов `casez` и `casex` не должен быть сигналом с третьим состоянием.

CASE10 [A] — значения «безразлично» (?) допускаются только в метках операторов `casez`, `casex` и `case inside`: их использование в иных местах приводит к расхождениям между синтезом и имитационным моделированием.

Категория **design**: логика и семантика описания

DESIGN01 [C] — в описании не должно быть комбинационных петель, то есть замкнутых комбинационных цепей без элементов памяти: их поведение не определено и может приводить к автоколебаниям.

DESIGN02 [П, Д] — в описании не должно быть недостижимого (мертвого) кода, в том числе внутри блоков `generate`: он является признаком ошибки проектирования и увеличивает сложность сопровождения.

DESIGN03 [T, И] — сигнал разрешения триггера не должен быть константным (подключенным к `supply0`, `supply1` или к константе): константное разрешение делает триггер бесполезным либо неуправляемым.

DESIGN04 [A] — процедурный блок `always` должен содержать список чувствительности: его отсутствие приводит к бесконечному закликиванию при имитационном моделировании.

DESIGN05 [A] — в одном списке чувствительности не должны смешиваться сигналы, чувствительные к фронту, и сигналы, чувствительные к уровню (например, `always @(posedge clk or reset)`): такое описание неоднозначно интерпретируется инструментами синтеза.

Категория **fsm**: конечные автоматы

FSM01 [KA] — каждый конечный автомат описания должен быть выявлен и извлечен в виде графа состояний и переходов. Правило информационное: срабатывание сообщает о найденном автомате, а не о дефекте.

FSM02 [KA] — каждая пара взаимодействующих конечных автоматов должна быть выявлена (раздел 3.6). Правило информационное: срабатывание сообщает о выявленной паре, а не о дефекте.

FSM03 [KA] — конечные автоматы не должны содержать недостижимых состояний, мертвых переходов, тупиковых состояний и взаимоблокировок (раздел 3.4).

Категория **latch**: защелки

LATCH01 [П] — защелки не должны выводиться непреднамеренно: в комбинационном описании переменная должна получать значение на всех путях управления. Непреднамеренный вывод защелки является одной из наиболее распространенных ошибок описания комбинационной логики.

LATCH02 [Ш, Л] — RS-защелки не должны строиться из примитивных логических вентилей (`and`, `or`, `not`, `nand`, `nor`): такие конструкции плохо поддаются временному анализу и синтезу (раздел 4.5).

LATCH03 [П] — сигнал разрешения защелки должен быть одноразрядным: многоразрядный сигнал разрешения порождает неоднозначную семантику стробирования.

LATCH04 [T, Д] — выход защелки не должен подаваться на вход другой защелки, стробируемой той же фазой сигнала разрешения: возникает «прозрачный» путь и нарушается временная модель.

LATCH05 [T, Д, С] — не должно быть комбинационной обратной связи с выхода защелки на ее вход данных или сигнал разрешения: такая конструкция образует неуправляемый автоколебательный контур.

Категория **loop**: операторы цикла

LOOP01 [A] — условие завершения цикла `for` не должно быть константным: константное условие приводит либо к бесконечному циклу, либо к циклу, не выполняющемуся ни разу.

LOOP02 [A] — индекс цикла должен иметь целочисленный тип.

LOOP03 [И] — в условии цикла не должны использоваться значения X и Z: результат сравнения с неопределенным значением непредсказуем.

Категория **port**: порты и соединения

PORT01[†] [A] — объявленный входной порт должен использоваться.

PORT02[†] [A] — выходной порт экземпляра должен быть подключен.

PORT03 [И] — входные порты экземпляра должны иметь подключенные и инициализированные фактические аргументы.

PORT04 [Т, Д, И] — входные и выходные порты модуля должны быть регистровыми: запись в выходные порты и чтение входных должны выполняться в тактируемых блоках, что обеспечивает предсказуемую синхронизацию при повторном использовании модуля.

PORT05 [И] — порт, объявленный как `inout` или `input`, не должен принудительно приводиться к направлению `output` вследствие способа его подключения.

PORT06 [И] — порт, объявленный как `inout` или `output`, не должен принудительно приводиться к направлению `input`.

PORT07 [И] — порт, объявленный как `input` или `output`, не должен принудительно приводиться к направлению `inout`.

PORT08 [A] — оператор `force` не должен применяться к портам: такая конструкция несинтезируема.

PORT09[†] [A] — число подключенных входных портов экземпляра должно соответствовать определению модуля.

PORT10 [A] — шины при создании экземпляра модуля не должны подключаться в обратном порядке разрядов (например, [15:0] к [0:15]): это приводит к трудноуловимым ошибкам передачи данных (раздел 2.3).

PORT11[†] [A] — идентификатор, ранее объявленный как порт, не должен переопределяться.

PORT12[†] [A] — входному порту должно быть сопоставлено значение.

PORT13 [И] — все биты входного порта должны получать соответствующие биты фактического аргумента.

PORT14[†] [A] — диапазоны в объявлениях портов ввода-вывода должны быть совместимы.

Категория **power**: домены питания

POW01 [МП] — сигналы, пересекающие границы доменов питания с разными уровнями напряжения, должны быть защищены преобразователями уровня напряжения в соответствии с состояниями питания, указанными в спецификации подсистемы питания (раздел 4.4).

POW02 [МП] — сигналы из отключаемых или частично отключаемых доменов питания должны быть изолированы для предотвращения функциональных и электрических нарушений (раздел 4.4).

Категория **pp**: директивы препроцессора

PP01 [A] — директивы препроцессора не должны размещаться внутри комментариев, где они не обрабатываются.

PP02[†] [A] — макрос не должен переопределяться.

Категория **range**: выход за границы диапазонов

RANGE01 [Д] — обращения к массивам и битовым векторам не должны выходить за их границы.

Категория **reset**: сигналы сброса

RESET01 [T] — первым оператором блока `always` с асинхронным сбросом должен быть оператор `if`, содержащий этот сброс в условии, иначе сброс может оказаться несинтезируемым.

RESET02 [T] — условие сброса не должно быть сложным: инструменты синтеза распознают лишь ограниченный набор форм асинхронного сброса, поэтому сравнение сигнала сброса с другим сигналом, переменной или выражением, не являющимся константой, считается нарушением.

RESET03 [T] — условие асинхронного сброса должно быть простым идентификатором либо его отрицанием (оператором `!` или `~`). Модификация сигнала сброса иным оператором (например, операцией свертки) делает сброс несинтезируемым.

RESET04 [T] — сигнал сброса в условии `if` должен использоваться только как простой идентификатор или в сравнении с помощью операторов `==` либо `!=`.

RESET05 [T] — в одном модуле не должны использоваться сигналы сброса (или установки) разной полярности: это затрудняет понимание и верификацию схемы.

RESET06 [T, И] — сигналы сброса или установки триггеров не должны находиться постоянно в активном или неактивном состоянии (быть подключенными к `supply0`, `supply1` или к константе): постоянно активный или неактивный сброс делает триггер бесполезным.

RESET07 [T] — использование сигнала сброса в выражениях блока `always` должно соответствовать фронту этого сигнала, объявленному в списке чувствительности.

RESET08 [T] — сигнал асинхронного сброса (или установки) должен использоваться именно как сигнал сброса, а не как обычный сигнал данных или синхронный сброс.

Категория **signal**: использование сигналов

SIGNAL01 [A] — переменным типа `supply0` и `supply1` не должны присваиваться значения: сетью питания невозможно управлять напрямую.

SIGNAL02 [T, И] — тактовый сигнал триггера не должен быть константным (подключенным к `supply0`, `supply1` или к константе): постоянный тактовый сигнал делает триггер неработоспособным.

SIGNAL03 [A] — глобальные сигналы, определенные вне тела подпрограммы (`function` или `task`), не должны использоваться в ее теле: они могут изменяться в любой части кода, что усложняет понимание поведения подпрограммы.

SIGNAL04 [П] — одному сигналу не должно выполняться несколько присваиваний в пределах одного такта имитационного моделирования. Учитываются списки чувствительности процедурных блоков и взаимоисключаемость путей условных конструкций.

SIGNAL05 [T] — список чувствительности комбинационного блока должен быть полным: все используемые в блоке сигналы должны быть в нем указаны, иначе поведение до и после синтеза расхожётся.

SIGNAL06 [Т, И] — управляющими сигналами буферов с третьим состоянием должна выступать только простая логика: сложная логика на линии разрешения изменяет момент переключения буфера и может вызвать конфликт на шине.

SIGNAL07 [И] — используемый сигнал (или переменная) должен быть предварительно установлен: чтение неинициализированного бита возвращает неопределенное значение, делая неопределенным и результат всего выражения (раздел 2.7.1).

SIGNAL08[†] [А] — многоразрядный сигнал не должен использоваться в списке чувствительности с указанием фронта.

SIGNAL09 [И] — сигнал без третьего состояния должен иметь единственный источник: наличие нескольких источников ведет к электрическому конфликту.

SIGNAL10 [А] — сигналы не должны сравниваться со значениями, содержащими Х или Z: такое сравнение допускается при имитационном моделировании, но игнорируется при синтезе.

SIGNAL11 [Т, И] — одновременно должен быть активен не более чем один управляющий сигнал шины с третьим состоянием: одновременная активация нескольких драйверов вызывает конфликт и может повредить устройство.

SIGNAL12 [Т, Л] — тактовые сигналы не должны буферизоваться вручную библиотечными ячейками или примитивами: буферизация эффективнее выполняется средствами физического проектирования, а ручные буферы снижают переносимость (раздел 4.5).

SIGNAL13 [А] — в одном блоке `always` не должно использоваться несколько тактовых сигналов: такая конструкция может оказаться несинтезируемой.

SIGNAL14 [Т, И] — тактовые сигналы последовательностной логики не должны использоваться как нетактовые: это порождает гонки, усложняет статический временной анализ и снижает переносимость.

SIGNAL15 [И] — порты `inout` и `output` не должны подключаться к сетям питания (`supply0`, `supply1`) или к константе: возможен электрический конфликт и повреждение ячейки.

SIGNAL16[†] [А] — используемый входной сигнал должен быть подключен и управляться корректным значением.

SIGNAL17 [И] — каждый установленный бит сигнала или переменной должен впоследствии использоваться: неиспользуемые установленные биты образуют избыточную логику.

SIGNAL18[†] [А] — в условии не должен использоваться вектор там, где ожидается одноразрядный сигнал.

SIGNAL19 [И] — сигнал или переменная с третьим состоянием не должны использоваться в условии оператора `if`: значение `z` после прохождения через логический вентиль обычно превращается в `x`, что приводит к расхождению поведения при имитационном моделировании и синтезе.

Категория **sync**: домены синхронизации и сброса

SYNC01 [—] — пересечения доменов синхронизации (CDC) должны быть выявлены и защищены синхронизаторами (раздел 1.5).

SYNC02 [—] — пересечения доменов сброса (RDC) должны быть выявлены и защищены (раздел 1.5).

Категория **synth**: синтезируемость описания

SYNTH01 [T] — в цепочке `if-else-if` последовательностного блока сначала должны проверяться все асинхронные сигналы (кроме тактового), а затем синхронные: иначе невозможно однозначно определить тактовый сигнал и сброс регистра.

SYNTH02 [A] — неявная последовательностная логика внутри тела блока `always` синтезируема, только если все переходы состояния происходят на одном и том же фронте одного тактового сигнала.

SYNTH03 [A] — списки чувствительности событий внутри тела блока `always` не должны использовать более одного фронта тактового сигнала.

SYNTH04 [A] — функции с типом результата `real` не должны использоваться: их результаты не поддерживаются рядом инструментов синтеза.

SYNTH05 [A] — список чувствительности процедурного блока не должен содержать несколько различных тактовых сигналов с указанием фронта: такие блоки несинтезируемы рядом инструментов.

SYNTH06 [A] — выражение в операторе повторения `repeat` должно быть константой, иначе оператор несинтезируем.

SYNTH07 [A] — в рамках одного процедурного блока не должны использоваться одновременно блокирующее и неблокирующее присваивания: блокирующие применяются в комбинационных блоках, неблокирующие — в последовательностных, а их смешение приводит к расхождению до и после синтеза.

SYNTH08 [A] — набор переменных блока шага цикла должен соответствовать набору переменных в условии его остановки, иначе цикл может оказаться несинтезируемым.

SYNTH09 [A] — набор переменных блока шага цикла должен соответствовать набору переменных блока инициализации: различие наборов делает цикл потенциально несинтезируемым.

SYNTH10 [A] — операторы точного сравнения `==/!=` не должны использоваться в логических выражениях: они сравнивают нефизические значения (X, Z), а их сведение к `==/!=` при синтезе порождает расхождение поведения.

SYNTH11 [Ш] — пользовательские примитивы (UDP) должны быть синтезируемыми: несинтезируемыми являются примитивы с более чем одним тактовым сигналом, со слишком малой таблицей состояний, сочетающие триггер и защелку либо содержащие переходы вида `?1/?0`, не позволяющие вывести сигналы сброса, установки или разрешения (раздел 2.5).

SYNTH12 [A] — сигнал или переменная не должны сравниваться со значением третьего состояния в условии `if`: средства синтеза трактуют такое сравнение как ложное, нарушая семантику схемы.

SYNTH13 [A] — в описании не должно быть несинтезируемых конструкций: блоков `initial` и `final`, задержек, подтягивающих вентилей, МОП-ключей, двунаправленных ключей, вызовов несинтезируемых системных функций (синтезируемыми считаются `$signed`, `$unsigned`, `$clog2`, `$bits`, `$countones`, `$onehot`, `$onehot0` и функции запроса размерностей), параллельных блоков `fork`, выражений `disable fork` и `disable` для внешних именованных блоков. Перечисленные конструкции не имеют физического аналога и предназначены только для имитационного моделирования.

Категория **timing**: временные задержки

TIMING01 [A] — задержки должны использоваться только при заданной директиве `timescale`, иначе единица и точность времени не определены.

TIMING02 [A] — значения задержек не должны быть отрицательными: отрицательная задержка лишена физического смысла и снижает эффективность имитационного моделирования.

Категория **type**: система типов

TYPE01 [A] — операция выбора бита не должна применяться к переменным типов `integer` и `time`, для которых она не определена.

TYPE02 [A] — операция выбора диапазона битов не должна применяться к переменным типов `integer` и `time`. При необходимости значение следует предварительно скопировать в переменную типа `reg` или `logic` требуемой разрядности.

TYPE03 [A] — операнды типа `real` не должны использоваться в операции сравнения `==`: сравнение вещественных значений на точное равенство дает непредсказуемый результат.

TYPE04[†] [A] — выражения в условии должны вычисляться как логические значения.

Категория **variable**: переменные и подпрограммы

VARIABLE01 [A] — в списке чувствительности блока `always` не должны одновременно использоваться передний и задний фронты одной переменной: такое описание несинтезируемо рядом инструментов.

VARIABLE02[†] [A] — в качестве индекса массива не должны использоваться переменные, содержащие значения `X` или `Z`.

VARIABLE03 [A] — переменные, объявленные в списке чувствительности, не должны модифицироваться внутри блока `always`: обновление сигнала из списка чувствительности приводит к ошибочным результатам имитационного моделирования.

VARIABLE04 [A] — глобальным переменным, определенным вне тела функций и задач, не должны присваиваться значения внутри них: это усложняет понимание значений переменной в произвольных участках кода.

VARIABLE05 [Д] — все биты возвращаемого значения функции должны присваиваться на всех путях ее тела: иначе результат формируется неполно, что становится источником трудноуловимых ошибок.

VARIABLE06[†] [A] — переменной должно быть присвоено значение.

VARIABLE07[†] [A] — системная функция `$size` не должна использоваться при объявлении упакованного массива.

VARIABLE08[†] [A] — правой части присваивания должно быть сопоставлено значение.

Категория **width**: разрядность

WIDTH01[†] [A] — разрядность аргументов функции должна соответствовать разрядности передаваемых значений.

WIDTH02[†] [A] — разрядность подсоединяемого сигнала должна соответствовать разрядности порта.

WIDTH03[†] [A] — разрядность сигнала, присоединяемого к порту экземпляра, должна соответствовать разрядности в определении порта.

WIDTH04[†] [A] — операнды арифметических и поразрядных операций (+, -, /, %, *, ~, ^, &) должны иметь одинаковую разрядность.

WIDTH05 [A] — операнды арифметического сравнения (>, >=, <, <=, ==, !=, ===, !==) должны иметь одинаковую разрядность: для части значений операнда большей разрядности результат сравнения оказывается константой, не зависящей от операнда меньшей разрядности.

WIDTH06 [A] — правая и левая части логических операторов должны иметь одинаковую разрядность, иначе значение выражения может вычисляться неправильно. Операторы логического сдвига составляют исключение.

WIDTH07 [A] — разрядность левой части присваивания не должна быть меньше подразумеваемой разрядности правой части: иначе происходит усечение и потеря данных.

WIDTH08[†] [A] — разрядность параметра должна быть не меньше разрядности присваиваемого ему значения.

WIDTH09[†] [A] — в задании констант не должно быть усечения лишних битов (например, 2 'b111).

Приложение Б

Примеры дефектов, выявленных в проектах с открытым исходным кодом

В настоящем приложении приведены примеры дефектов, выявленных системой SVAN в проектах с открытым исходным кодом (итоги обращений к разработчикам подведены в разделе 6.2). Статусы всех обращений сведены в таблице 24. Сначала подробно рассматриваются наиболее показательные случаи, затем приводится сводный перечень остальных обращений.

Выход за границы битового вектора в проекте OpenTitan

Детектор RANGE01 выявил выход за границы битового вектора в автоматически генерируемом пакете конфигурации списков управления доступом (RACL) проекта OpenTitan (обращение № 25913): функции `automatic` извлекают из вектора `rsvd` биты `[11:8]` и `[7:0]`, тогда как фактическая разрядность вектора, зависящая от параметра целевой платформы, меньше используемых индексов, поэтому биты `[11:8]` не могут быть извлечены ни при какой конфигурации (листинг Б.1). Дефект затрагивал пакеты всех целевых платформ, был подтвержден и исправлен разработчиками в генерирующем шаблоне.

Листинг Б.1: Нарушение правила RANGE01 в пакете `top_racl_pkg` (OpenTitan)

```

1 function automatic rac_l_role_t
2   tlul_extract_racl_role_bits(
3     logic [tlul_pkg::RsvdWidth-1:0] rsvd);
4   // фактическая разрядность rsvd меньше 12
5   return rac_l_role_t'(rsvd[11:8]); // выход за границы вектора
6 endfunction

```

Мертвый код в декодере команд ядра Ibex

Детектор правила DESIGN02 («в описании не должно быть недостижимого (мертвого) кода») выявил недостижимый код в декодере команд ядра Ibex (обращение № 2268): присваивание в ветви `else` вложенного условного оператора никогда не выполняется, поскольку внешняя ветвь `else` уже гарантирует условие `!instr[26]` (листинг Б.2). Дефект подтвержден и исправлен разработчиками: недостижимая ветвь удалена.

Листинг Б.2: Мертвый код в декодере `ibex_decoder.sv` (Ibex)

```

1  unique case (instr[14:12])
2      3'b101: begin
3          if (instr[26]) begin
4              ...
5          end else begin                // здесь instr[26] == 1'b0
6              unique case (instr[31:27])
7                  ...
8                  5'b0_0001: begin
9                      if (instr[26] == 1'b0) begin // условие всегда истинно
10                         ...
11                     end else begin          // ветвь недостижима
12                         ...
13                     end
14                 end

```

Мертвые переходы конечного автомата в проекте OpenC910

Подсистема анализа конечных автоматов (глава 3) выявила мертвые переходы в автомате модуля `st_ciu_snb_sab_entry` процессора OpenC910 (обращение № 49): локальный провод, которому присваивается константа 0 и который нигде не переопределяется и не подключается к выходам экземпляров, используется в условиях переходов автомата. Поэтому соответствующие условия всегда ложны, и переходы в состояние MEMR никогда не происходят. Аналогичные случаи мертвого кода из-за константных проводов обнаружены детекторами и в модулях `st_mmu_jtlb` (обращение № 45) и `st_lsu_snoop_snoq_entry` (обращение № 48) того же проекта: условия операторов `if`, проверяющие такие провода, всегда ложны.

Непреднамеренная защелка в модуле трассировки команд ядра CVA6

Детектор LATCH01 выявил непреднамеренную защелку в модуле трассировки команд ядра CVA6 (обращение № 3105): в блоке `always_comb` выходному сигналу `valid_o` не присваивается значение по умолчанию перед условным оператором, поэтому по части путей сигнал сохраняет прежнее значение и синтезируется защелка (листинг Б.3). Дефект подтвержден и исправлен разработчиками добавлением присваивания по умолчанию.

Листинг Б.3: Нарушение правила LATCH01 в модуле трассировки команд `iti.sv` (CVA6)

```

1  always_comb begin
2      for (int i = 0; i < CVA6Cfg.NrCommitPorts; i++) begin

```

```

3      // отсутствует присваивание по умолчанию valid_o[i] = 1'b0;
4      if (itt_out[i].valid) begin
5          valid_o[i] = itt_out[i].valid;    // выводится защелка
6      end
7  end
8 end

```

Неинициализированный сигнал в проекте BlackParrot

Детектор правила SIGNAL07 выявил в проекте BlackParrot переменную orcb, инициализируемую побитово в цикле, число итераций которого зависит от значения параметра: при некоторых конфигурациях параметр обращается в ноль, цикл не выполняется, и ни один бит переменной не инициализируется (листинг Б.4). Дефект был независимо найден другими разработчиками и сообщен ими в проект, поэтому в перечень обращений (таблица 24) не включен.

Листинг Б.4: Нарушение правила SIGNAL07 (BlackParrot)

```

1 localparam dword_width_gp = 64;
2 localparam num_bytes_lp   = dword_width_gp >> 8; // == 0
3 logic [num_bytes_lp-1:0][7:0] orcb;
4 for (genvar i = 0; i < num_bytes_lp; i++)
5     assign orcb[i] = {8{rs1[8*i +: 8]}}; // цикл не выполняется ни разу

```

Выход за границы провода в проекте systolic-array

Детектор RANGE01 выявил в модуле PE_array проекта systolic-array (обращение № 1) выборку диапазона, выходящую за границы провода: провод объявлен как wire [num1*16*num2-1:0] out_sum, однако в непрерывном присваивании используется выборка out_sum[num1*16*num2:(num1-1)*16*num2], старшая граница которой на единицу превышает разрядность провода. Дефект подтвержден разработчиком.

Выход за границы массива конфигурации защиты памяти в ядре CVA6

Детектор RANGE01 выявил в модуле регистров управления csr_regfile ядра CVA6 (обращение № 3452) обращение за границы массива конфигурации физической защиты памяти (PMP): массив pmpcfg_q объявлен из 64 элементов, а при записи регистра pmpaddr63 индекс index, вычисляемый как смещение адреса регистра относительно pmpaddr0, равен 63, и проверка блокировки следующей записи обращается к элементу pmpcfg_q[index+1], то есть к несуществующему элементу с номером 64 (листинг Б.5). Диапазон индекса детектор получает по меткам оператора case (регистры pmpaddr0–pmpaddr63), откуда для выражения index+1 следует интервал 1–64, выходящий за границы массива (раздел 2.7).

Дефект подтвержден и исправлен разработчиками: проверка следующей записи выполняется только при `index` меньше 63, для последней записи запись адреса разрешена без нее, а в набор тестов ядра добавлена проверка граничного регистра `pmpaddr63`.

Листинг Б.5: Нарушение правила RANGE01 в модуле `csr_regfile.sv` (CVA6)

```

1 | riscv::pmpcfg_t [63:0] pmpcfg_q, pmpcfg_d, pmpcfg_next;
2 | ...
3 | riscv::CSR_PMPADDR0, ..., riscv::CSR_PMPADDR63: begin
4 |     // индекс вычисляется как смещение от PMPADDR0: 0..63
5 |     automatic logic [11:0] index =
6 |         csr_addr.address[11:0] - riscv::CSR_PMPADDR0;
7 |     // при index == 63 обращение к pmpcfg_q[64] за границами массива
8 |     if (!pmpcfg_q[index].locked &&
9 |         !(pmpcfg_q[index+1].locked &&
10 |          pmpcfg_q[index+1].addr_mode == riscv::TOR)) begin
11 |         pmpaddr_d[index] = csr_wdata[CVA6Cfg.PLEN-3:0];
12 |     end
13 | end

```

Сводный перечень обращений к разработчикам открытых проектов

В таблице 24 перечислены все обращения к разработчикам открытых проектов, включая рассмотренные выше. Номера обращений приведены по системам отслеживания ошибок репозиторий проектов, адреса записей даны в подстрочных ссылках, статусы приведены на сентябрь 2026 г. Исправленным считается обращение, закрытое исправлением в проекте, подтвержденным — признанное разработчиками дефектом, отклоненным — закрытое без исправления с разъяснением, что конструкция намеренная или не используется.

Таблица 24 — Сводный перечень обращений к разработчикам открытых проектов

Проект	Обращение	Суть нарушения	Статус
OpenTitan	№ 25913 ¹	выход за границы битового вектора в пакете конфигурации RACL	исправлено
Ibex	№ 2268 ²	недостижимый код в декодере команд	исправлено
OpenC910	№ 45 ³	мертвый код в модуле <code>ct_mmu_jtlb</code>	нет ответа
OpenC910	№ 48 ⁴	мертвый код в модуле <code>ct_lsu_snoop_snq_entry</code>	нет ответа

¹URL: <https://github.com/lowRISC/opentitan/issues/25913> (дата обращения: 24.09.2026).

²URL: <https://github.com/lowRISC/ibex/issues/2268> (дата обращения: 24.09.2026).

³URL: <https://github.com/XUANTIE-RV/openc910/issues/45> (дата обращения: 24.09.2026).

⁴URL: <https://github.com/XUANTIE-RV/openc910/issues/48> (дата обращения: 24.09.2026).

Продолжение таблицы 24

Проект	Обращение	Суть нарушения	Статус
OpenC910	№ 49 ⁵	мертвые переходы автомата модуля <code>st_ciu_snb_sab_entry</code>	нет ответа
CVA6	№ 3105 ⁶	непреднамеренная защелка в модуле трассировки команд	исправлено
systolic-array	№ 1 ⁷	выход за границы провода в модуле <code>PE_array</code>	подтверждено
SCR1	№ 63 ⁸	множественное управление статической переменной функции, вызываемой из двух блоков <code>always_ff</code> (тестовое окружение)	отклонено
SCR1	№ 71 ⁹	использование переменной, которой нигде не присваивается значение (тестовое окружение)	подтверждено
SCR1	№ 72 ¹⁰	непокрытая метка оператора <code>case</code> при отсутствии ветви <code>default</code>	отклонено
BlackParrot	№ 1230 ¹¹	разрядность метки оператора <code>case</code> (4 бита) не соответствует разрядности селектора (3 бита)	исправлено
CVA6	№ 2591 ¹²	оператор <code>case inside</code> покрывает лишь значения 0–31 шестизрядного селектора (значения 32–63 не обрабатываются)	исправлено
CVA6	№ 2618 ¹³	некорректные аргументы системной задачи <code>\$fatal</code> (строка сообщения на месте кода завершения, раздел 20.10 стандарта)	исправлено
CVA6	№ 2680 ¹⁴	оператор <code>case</code> без ветви <code>default</code> не покрывает один из уровней привилегий из пакета <code>riscv</code>	отклонено
CVA6	№ 2681 ¹⁵	значения «безразлично» в метках обычного оператора <code>case</code> (допустимы лишь в <code>casex/casez</code> и <code>case inside</code>)	нет ответа
CVA6	№ 3446 ¹⁶	выход за границы массивов из 11 элементов в цикле по 100 элементам в модуле трассировки тестового окружения	исправлено

⁵URL: <https://github.com/XUANTIE-RV/openc910/issues/49> (дата обращения: 24.09.2026).

⁶URL: <https://github.com/openhwfoundation/cva6/issues/3105> (дата обращения: 24.09.2026).

⁷URL: <https://github.com/Dazhuzhu-github/systolic-array/issues/1> (дата обращения: 24.09.2026).

⁸URL: <https://github.com/syntacore/scr1/issues/63> (дата обращения: 24.09.2026).

⁹URL: <https://github.com/syntacore/scr1/issues/71> (дата обращения: 24.09.2026).

¹⁰URL: <https://github.com/syntacore/scr1/issues/72> (дата обращения: 24.09.2026).

¹¹URL: <https://github.com/black-parrot/black-parrot/issues/1230> (дата обращения: 24.09.2026).

¹²URL: <https://github.com/openhwfoundation/cva6/issues/2591> (дата обращения: 24.09.2026).

¹³URL: <https://github.com/openhwfoundation/cva6/issues/2618> (дата обращения: 24.09.2026).

¹⁴URL: <https://github.com/openhwfoundation/cva6/issues/2680> (дата обращения: 24.09.2026).

¹⁵URL: <https://github.com/openhwfoundation/cva6/issues/2681> (дата обращения: 24.09.2026).

¹⁶URL: <https://github.com/openhwfoundation/cva6/issues/3446> (дата обращения: 24.09.2026).

Продолжение таблицы 24

Проект	Обращение	Суть нарушения	Статус
CVA6	№ 3452 ¹⁷	обращение к элементу <code>index+1</code> массива конфигурации защиты памяти из 64 элементов при индексе 63 в модуле регистров управления	исправлено
IHP-Open-PDK	№ 208 ¹⁸	пути задержек в блоках <code>specify</code> , начинающиеся с выходного порта (нарушение ограничений путей, раздел 30.4.1 стандарта)	исправлено
IHP-Open-PDK	№ 209 ¹⁹	недопустимое использование условия <code>ifnone</code> (раздел 30.4.4.4 стандарта)	подтверждено
OpenC910	№ 52 ²⁰	использование общей переменной цикла в двух параллельных процедурных блоках — потенциальная гонка (раздел 12.7.1 стандарта)	нет ответа
OpenROAD	№ 7363 ²¹	использование необъявленного объекта в тестовом UPF-описании	исправлено
x-heep	№ 687 ²²	отсутствие обязательной опции <code>-update</code> команды <code>add_power_state</code> в UPF-описании	нет ответа
8b10b	№ 2 ²³	несуществующая опция <code>-control_ports</code> команды <code>create_power_switch</code> в UPF-описании	нет ответа
Low-Power-Async-CDC-FIFO-IP	№ 1 ²⁴	использование необъявленной логической цепи управляющего порта в команде <code>create_power_switch</code> UPF-описания	нет ответа

¹⁷URL: <https://github.com/openhwfoundation/cva6/issues/3452> (дата обращения: 24.09.2026).

¹⁸URL: <https://github.com/IHP-GmbH/IHP-Open-PDK/issues/208> (дата обращения: 24.09.2026).

¹⁹URL: <https://github.com/IHP-GmbH/IHP-Open-PDK/issues/209> (дата обращения: 24.09.2026).

²⁰URL: <https://github.com/XUANTIE-RV/openc910/issues/52> (дата обращения: 24.09.2026).

²¹URL: <https://github.com/The-OpenROAD-Project/OpenROAD/issues/7363> (дата обращения: 24.09.2026).

²²URL: <https://github.com/x-heep/x-heep/issues/687> (дата обращения: 24.09.2026).

²³URL: <https://github.com/insularity1337/8b10b/issues/2> (дата обращения: 24.09.2026).

²⁴URL: <https://github.com/Olajide-Badejo/Low-Power-Async-CDC-FIFO-IP/issues/1> (дата обращения: 24.09.2026).

Приложение В

Материалы сравнительного исследования с открытыми средствами

В настоящем приложении приведены материалы сравнительного исследования с открытыми средствами (раздел 6.7): версии проектов тестового набора, замеры времени и памяти по проектам, таблица соответствия диагностик средств правилам набора и матрица числа срабатываний (таблица 29).

Версии (реvisions репозитория) проектов тестового набора зафиксированы в таблице 25.

Таблица 25 — Версии проектов тестового набора

Проект	Ревизия	Проект	Ревизия
AES	80dc471	CVA6	v5.3.0-150-g8e694e64
PicoRV32	v1.0-70-g87c89ac	BlackParrot	a54252c4
SCR1	2024.11	OpenC910	b91c909
VeeR EH1	v1.9-13-g915fb34	OpenTitan	Earlgrey-PROD-M6-RC0
VeeR EL2	2.0-246-gab93a36	Ibex	pulpissimo-v1.0.1-2107

Время анализа и пиковая память по проектам приведены в таблицах 26 и 27.

Таблица 26 — Время анализа по проектам, с

Проект	SVAN	Verilator	svlint	Verible	Surelog	Yosys
AES	0,1	0,2	0,3	0,0	0,2	0,3
PicoRV32	3,0	0,1	0,4	0,1	1,6	0,2
SCR1	4,2	0,3	0,2*	0,1	3,3	0,9
Ibex	8,5	0,5	2,9	0,1	4,9	0,6
VeeR EH1	7,0	2,4	5,1	0,2	8,9	7,5
VeeR EL2	38,4	5,6	6,8	0,2	144,7 [±]	8,6
OpenC910	148,6	45,1	34,3 [±]	1,0 [±]	26,8*	1607,1
BlackParrot	7,0	7,3	19,2	0,2	13,7 [±]	12,9
CVA6	229,3	1,5 [±]	11,1 [±]	0,3	28,5 [±]	42,4
OpenTitan	544,0	46,3	12,3*	0,8	86,2*	591,2

* — отказ (время до отказа), [±] — анализ завершен частично. Для SVAN приведено время детекторов правил на полном наборе исходных текстов, Yosys запускался на синтезируемой части проектов. Время и память средств анализа автоматов SVAN по проектам приведены в таблице 7.

Таблица 27 — Пиковая память по проектам

Проект	SVAN	Verilator	svlint	Verible	Surelog	Yosys
AES	95 МБ	29 МБ	27 МБ	6 МБ	32 МБ	47 МБ
PicoRV32	82 МБ	21 МБ	45 МБ	13 МБ	130 МБ	34 МБ
SCR1	4,8 ГБ	49 МБ	25 МБ*	8 МБ	343 МБ	80 МБ
Ibex	541 МБ	65 МБ	46 МБ	10 МБ	540 МБ	76 МБ
VeeR EH1	423 МБ	178 МБ	59 МБ	12 МБ	736 МБ	395 МБ
VeeR EL2	1,9 ГБ	350 МБ	70 МБ	15 МБ	18,5 ГБ [±]	514 МБ
OpenC910	4,4 ГБ	3,0 ГБ	214 МБ [±]	26 МБ [±]	4,9 ГБ*	5,7 ГБ
BlackParrot	985 МБ	400 МБ	155 МБ	31 МБ	2,1 ГБ [±]	798 МБ
CVA6	7,9 ГБ	236 МБ [±]	68 МБ [±]	11 МБ	2,2 ГБ [±]	1,3 ГБ
OpenTitan	12,7 ГБ	2,4 ГБ	605 МБ*	73 МБ	6,0 ГБ*	4,1 ГБ

В таблицу 28 включены только сопоставления, подтвержденные на синтетических тестах (раздел 6.7.1). Тип соответствия: «полное» — диагностика по документации проверяет то же свойство, «частичное» — родственное или более широкое свойство.

Таблица 28 — Подтвержденное соответствие диагностик открытых средств правил набора

Диагностика средства	Правило(а) набора	Соответствие
<i>Verilator</i>		
ASSIGNIN	PORT05	частичное
BLKSEQ	ASSIGN03	полное
CASEINCOMPLETE	CASE01, CASE05	частичное
CASEOVERLAP	CASE02, CASE06	частичное
CASEWITHX	CASE07, CASE08	частичное
CASEX	CASE09	частичное
COMBDLY	ASSIGN04	частичное
IMPURE	SIGNAL03, VARIABLE04	частичное
INFINITEL00P	LOOP01	частичное
LATCH	LATCH01	полное
MULTIDRIVEN	SIGNAL09	частичное
NORETURN	VARIABLE05	частичное
PINMISSING	PORT02, PORT09	полное
REDEFMACRO	PP02	полное
STMTDLY	SYNTH13	частичное
SYNCASYNCTNET	RESET08	полное
UNDRIVEN	SIGNAL07, SIGNAL16	частичное
UNOPTFLAT	DESIGN01	частичное
UNUSED SIGNAL	SIGNAL17	частичное

Продолжение таблицы 28

Диагностика средства	Правило(а) набора	Соответствие
WIDTH	CASE03, WIDTH04, WIDTH05, WIDTH06	частичное
WIDTHTRUNC	ARITH01, WIDTH02, WIDTH07, WIDTH09	частичное
<i>svlint</i>		
blocking_assignment_in_always_at_edge	ASSIGN03	полное
blocking_assignment_in_always_ff	ASSIGN03	полное
explicit_case_default	CASE01	полное
implicit_case_default	CASE01	полное
non_blocking_assignment_in_always_comb	ASSIGN04	частичное
operator_case_equality	SYNTH10	полное
procedural_continuous_assignment	SYNTH13	частичное
<i>Verible</i>		
always-comb-blocking	ASSIGN04	частичное
always-ff-non-blocking	ASSIGN03	полное
case-missing-default	CASE01	полное
module-port	PORT09	частичное
truncated-numeric-literal	WIDTH09	полное
<i>Surelog</i>		
PP_MULTIPLY_DEFINED_MACRO	PP02	полное
UHDH_DIVIDE_BY_ZERO	ARITH02	полное
UHDH_NON_SYNTHESIZABLE	SYNTH13	частичное
<i>Yosys</i>		
proc_dff: сложный асинхронный сброс (ошибка)	SYNTH01	частичное
proc_dff: сложный асинхронный сброс (ошибка)	RESET02	частичное
fsm_detect: найден регистр состояния	FSM01	частичное
check, proc_dlatch: защелка	LATCH01	полное
proc_dff: несколько событий фронта (ошибка)	SYNTH05	частичное
read_verilog: разрядность подключения порта	PORT13	частичное
check: сигнал без источника	SIGNAL07, SIGNAL16	частичное
check: комбинационная петля	DESIGN01	полное
check: конфликтующие источники	SIGNAL04, SIGNAL09	частичное
несинтезируемое управление временем (ошибка)	SYNTH13	частичное
<i>Icarus Verilog</i>		
-wfloating-nets	SIGNAL07	частичное

Продолжение таблицы 28

Диагностика средства	Правило(а) набора	Соответствие
-Winfloop	LOOP01	частичное
-Wportbind	PORT03	частичное

Число срабатываний по сопоставленным правилам (сумма по проектам, на которых анализ завершен полностью или частично) приведено в таблице 29. Для открытых средств учтены только сопоставленные диагностики, для SVAN — все правила, по которым были срабатывания. Столбец SVAN относится к запуску детекторов правил на полном наборе исходных текстов, кроме строки FSM01: автоматы извлекаются на синтезируемой части проектов. Срабатывания детекторов SVAN на входе Yosys приведены в разделе 6.7.6. Диагностики Verilator семейства WIDTH сопоставлены нескольким правилам набора, поэтому одно срабатывание учтено в нескольких строках: все 1757 срабатываний семейства (WIDTHEXPAND — 1091, WIDTHTRUNC — 663, WIDTHCONCAT — 3) входят в строки CASE03, WIDTH04, WIDTH05 и WIDTH06, а срабатывания WIDTHTRUNC — еще и в строки ARITH01, WIDTH02, WIDTH07 и WIDTH09. Аналогично срабатывания CASEINCOMPLETE, CASEOVERLAP, UNDRIVEN и PINMISSING учтены в двух строках каждое. У разработанного инструмента в нескольких строках учтены диагностики компилятора -wint-bool-conv (SIGNAL18 и TYPE04, 63 срабатывания), -wunassigned-variable (PORT12 и VARIABLE06, 92) и -wundriven-net (PORT12, SIGNAL16 и VARIABLE06, 39), поэтому сумма столбца SVAN на $63 + 92 + 2 \cdot 39 = 233$ больше числа срабатываний. Без повторного учета у Verilator 4964 срабатывания, у разработанного инструмента — 53 486 срабатываний детекторов и диагностик компилятора и 1934 извлеченных автомата (строка FSM01).

Таблица 29 — Число срабатываний по правилам набора (для открытых средств — по сопоставленным диагностикам)

Правило	SVAN	Verilator	svlint	Verible	Surelog	Yosys
ARITH01	35	663	—	—	—	—
ARITH02	—	—	—	—	1	—
ASSIGN01	710	—	—	—	—	—
ASSIGN02	35	—	—	—	—	—
ASSIGN03	109	53	19	32	—	—
ASSIGN04	—	1	—	—	—	—
ASSIGN05	390	—	—	—	—	—

Продолжение таблицы 29

Правило	SVAN	Verilator	svlint	Verible	Surelog	Yosys
ASSIGN06	3307	—	—	—	—	—
CASE01	58	10	1084	71	—	—
CASE02	49	4	—	—	—	—
CASE03	42	1757	—	—	—	—
CASE04	736	—	—	—	—	—
CASE05	2	10	—	—	—	—
CASE06	—	4	—	—	—	—
CASE07	5	—	—	—	—	—
CASE08	5	—	—	—	—	—
CASE10	404	—	—	—	—	—
DESIGN01	—	23	—	—	—	—
DESIGN02	26	—	—	—	—	—
DESIGN05	3	—	—	—	—	—
FSM01	1934	—	—	—	—	1069
LATCH01	26	2	—	—	—	91
LATCH03	2	—	—	—	—	—
LOOP01	—	2	—	—	—	—
LOOP02	5	—	—	—	—	—
PORT01	614	—	—	—	—	—
PORT02	—	1	—	—	—	—
PORT03	103	—	—	—	—	—
PORT04	40 322	—	—	—	—	—
PORT06	17	—	—	—	—	—
PORT09	—	1	—	72	—	—
PORT12	131	—	—	—	—	—
PORT13	94	—	—	—	—	—
PP02	—	—	—	—	5	—
RANGE01	8	—	—	—	—	—
RESET01	1	—	—	—	—	—
RESET05	5	—	—	—	—	—
RESET06	13	—	—	—	—	—
RESET08	8	17	—	—	—	—
SIGNAL03	224	—	—	—	—	—
SIGNAL04	2	—	—	—	—	—
SIGNAL05	1	—	—	—	—	—
SIGNAL07	37	234	—	—	—	384
SIGNAL09	4	95	—	—	—	—
SIGNAL16	160	234	—	—	—	384
SIGNAL17	1764	2750	—	—	—	—

Продолжение таблицы 29

Правило	SVAN	Verilator	svlint	Verible	Surelog	Yosys
SIGNAL18	63	—	—	—	—	—
SYNTH07	14	—	—	—	—	—
SYNTH08	11	—	—	—	—	—
SYNTH10	53	—	123	—	—	—
SYNTH13	1127	15	—	—	386	—
TIMING01	33	—	—	—	—	—
TYPE04	63	—	—	—	—	—
VARIABLE04	50	—	—	—	—	—
VARIABLE06	131	—	—	—	—	—
WIDTH01	691	—	—	—	—	—
WIDTH02	7	663	—	—	—	—
WIDTH04	749	1757	—	—	—	—
WIDTH05	71	1757	—	—	—	—
WIDTH06	5	1757	—	—	—	—
WIDTH07	1186	663	—	—	—	—
WIDTH08	8	—	—	—	—	—
WIDTH09	—	663	—	9	—	—

Приложение Г

Сведения о внедрении и регистрации результатов

Результаты диссертационной работы получены в рамках опытно-конструкторской работы «Разработка системы статического анализа для языка описания аппаратуры» (шифр «САПР-Анализ»), выполняемой при поддержке Министерства промышленности и торговли Российской Федерации. Главной исполнитель — АО «МНТЦ МИЭТ».

Сведения о внедрении системы приведены в разделе 6.8. Ниже приведены копии свидетельств о государственной регистрации программ для ЭВМ № 2026615835 [9] и № 2026683339 [10].

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2026615835

Система статического анализа для языка описания аппаратуры

Правообладатели: *Российская Федерация, от имени которой выступает Министерство промышленности и торговли Российской Федерации (RU), Акционерное общество «Международный научно-технологический центр МИЭТ» (RU)*

Авторы: *Камкин Александр Сергеевич (RU), Буздалов Денис Викторович (RU), Бучацкий Рубен Артурович (RU), Волохов Алексей Георгиевич (RU), Вязовцев Андрей Викторович (RU), Демаков Алексей Васильевич (RU), Егiazарян Арман Вагаршакович (AM), Зеленев Сергей Вадимович (RU), Коцыняк Артём Михайлович (RU), Литвинов Михаил Юрьевич (RU), Романов Никита Сергеевич (RU), Самоваров Дмитрий Олегович (RU), Смолов Сергей Александрович (RU), Чернявских Илья Игоревич (RU), Чупилко Михаил Михайлович (RU), Чуркин Ян Андреевич (RU)*

Заявка № 2025692899

Дата поступления 26 ноября 2025 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 03 марта 2026 г.



Руководитель Федеральной службы
по интеллектуальной собственности

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2026683339

«Система статического анализа описаний аппаратуры
на языке VHDL»

Правообладатель: *Федеральное государственное бюджетное учреждение науки Институт системного программирования им В.П. Иванникова Российской академии наук (RU)*

Авторы: *Бучацкий Рубен Артурович (RU), Вязовцев Андрей Викторович (RU), Камкин Александр Сергеевич (RU), Коцыняк Артём Михайлович (RU), Самоваров Дмитрий Олегович (RU), Чуркин Ян Андреевич (RU)*

Заявка № 2026681213

Дата поступления 09 июля 2026 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 30 июля 2026 г.



Руководитель Федеральной службы
по интеллектуальной собственности

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат 00a570e4f7add8d531b4b8818e75f29506
Владелец **Зубов Юрий Сергеевич**
Действителен с 04.09.2025 по 28.11.2026

Ю.С. Зубов