

Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
GRADUATION THESIS

Реализация модуля моделирования аппаратуры для SCADA-систем

**Обучающийся / Student** Тюрин Иван Николаевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники

**Группа/Group** Р34102

**Направление подготовки/ Subject area** 09.03.04 Программная инженерия

**Образовательная программа / Educational program** Системное и прикладное программное обеспечение 2021

**Язык реализации ОП / Language of the educational program** Русский

**Квалификация/ Degree level** Бакалавр

**Руководитель ВКР/ Thesis supervisor** Осипов Святослав Владимирович, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель практики")

**Консультант не из ИТМО / Third-party consultant** Нозик Александр Аркадьевич, кафедра общей физики МФТИ, центр научного программирования, директор Центра научного программирования, кандидат физико-математических наук, доцент

Обучающийся/Student


Документ подписан	
Тюрин Иван Николаевич	
19.05.2025	

(эл. подпись/ signature)

Тюрин Иван  
Николаевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Осипов Святослав Владимирович	
19.05.2025	

(эл. подпись/ signature)

Осипов  
Святослав  
Владимирович

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Тюрин Иван Николаевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники

**Группа/Group** P34102

**Направление подготовки/ Subject area** 09.03.04 Программная инженерия

**Образовательная программа / Educational program** Системное и прикладное программное обеспечение 2021

**Язык реализации ОП / Language of the educational program** Русский

**Квалификация/ Degree level** Бакалавр

**Тема ВКР/ Thesis topic** Реализация модуля моделирования аппаратуры для SCADA-систем  
**Руководитель ВКР/ Thesis supervisor** Осипов Святослав Владимирович, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель практики")

**Консультант не из ИТМО / Third-party consultant** Нозик Александр Аркадьевич, кафедра общей физики МФТИ, центр научного программирования, директор Центра научного программирования, кандидат физико-математических наук, доцент

**Характеристика темы ВКР / Description of thesis subject (topic)**

**Тема в области фундаментальных исследований / Subject of fundamental research:** нет / not

**Тема в области прикладных исследований / Subject of applied research:** да / yes

**Основные вопросы, подлежащие разработке / Key issues to be analyzed**

Техническое задание:

Реализовать программный модуль на языке программирования Kotlin для фреймворка "Controls.kt", предоставляющий поддержку моделирования аппаратуры на уровне цифровых схем для виртуальных устройств, и автоматизирующий процесс построения исполняемых моделей на основе языка описания аппаратуры.

Содержание работы:

Выпускная квалификационная работа предполагает разработку программного решения на языке программирования Kotlin, предоставляющего возможность использования виртуальных устройств в составе фреймворка "Controls.kt" в режиме моделирования работы цифровых схем, и анализ разработанного решения.

Цель: Повышение качества систем на базе фреймворка "Controls.kt" за счет их сквозного тестирования с применением моделей цифровых схем в виртуальных устройствах.

Задачи работы:

1. Исследование имеющихся решений для моделирования аппаратуры.
2. Разработка программного модуля на языке программирования Kotlin для работы с моделями аппаратуры.
3. Демонстрация применения моделей аппаратуры в системе на базе Controls-kt.
4. Анализ полученных результатов.

Рекомендуемые материалы и пособия для выполнения работы:

1. Документация и исходный код проекта Controls-kt.
2. Документация проекта Chisel.
3. Документация проекта Gradle.
4. Документация проекта Kotlin Poet.
5. Описание и документация FFM API в Java из проекта Panama.
6. Документация и исходный код проекта LLVM CIRCT.

**Форма представления материалов ВКР / Format(s) of thesis materials:**

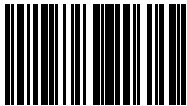
Пояснительная записка, презентация.

**Дата выдачи задания / Assignment issued on:** 14.10.2024

**Срок представления готовой ВКР / Deadline for final edition of the thesis** 25.05.2025

**СОГЛАСОВАНО / AGREED:**

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Осипов Святослав Владимирович	
06.05.2025	

(эл. подпись)

Осипов  
Святослав  
Владимирович

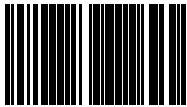
Задание принял к  
исполнению/ Objectives  
assumed BY

Документ подписан	
Тюрин Иван Николаевич	
06.05.2025	

(эл. подпись)

Тюрин Иван  
Николаевич

Руководитель ОП/ Head  
of educational program

Документ подписан	
Дергачев Андрей Михайлович	
12.05.2025	

(эл. подпись)

Дергачев  
Андрей  
Михайлович

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся / Student** Тюрин Иван Николаевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники  
**Группа/Group** P34102  
**Направление подготовки/ Subject area** 09.03.04 Программная инженерия  
**Образовательная программа / Educational program** Системное и прикладное программное обеспечение 2021  
**Язык реализации ОП / Language of the educational program** Русский  
**Квалификация/ Degree level** Бакалавр  
**Тема ВКР/ Thesis topic** Реализация модуля моделирования аппаратуры для SCADA-систем  
**Руководитель ВКР/ Thesis supervisor** Осипов Святослав Владимирович, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель практики")  
**Консультант не из ИТМО / Third-party consultant** Нозик Александр Аркадьевич, кафедра общей физики МФТИ, центр научного программирования, директор Центра научного программирования, кандидат физико-математических наук, доцент

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
DESCRIPTION OF THE GRADUATION THESIS**

**Цель исследования / Research goal**

Повышение качества систем на базе фреймворка "Controls.kt" за счет их сквозного тестирования с применением моделей цифровых схем в виртуальных устройствах.

**Задачи, решаемые в ВКР / Research tasks**

1. Исследование имеющихся решений для моделирования аппаратуры. 2. Разработка программного модуля на языке программирования Kotlin для работы с моделями аппаратуры. 3. Демонстрация применения моделей аппаратуры в системе на базе Controls.kt. 4. Анализ полученных результатов.

**Краткая характеристика полученных результатов / Short summary of results/findings**

Разработан принцип генерации программных моделей на основе языка описания аппаратуры. Разработан программный модуль, предоставляющий поддержку работы с моделями аппаратуры для фреймворка Controls.kt. На примере продемонстрирована возможность использования разработанного решения при сквозном тестировании, что позволяет повысить качество разрабатываемой системы за счет раннего выявления ошибок.

Обучающийся/Student

Документ подписан	
----------------------	--

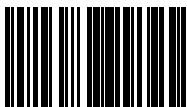
	
Тюрин Иван Николаевич	
19.05.2025	

(эл. подпись/ signature)

Тюрин Иван  
Николаевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Осипов Святослав Владимирович	
19.05.2025	

(эл. подпись/ signature)

Осипов  
Святослав  
Владимирович

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ .....	9
ВВЕДЕНИЕ .....	11
1 Обзор предметной области .....	12
1.1 Область применения .....	12
1.2 Языки описания аппаратуры .....	13
1.3 Промежуточное представление цифровых схем .....	14
1.4 Средства моделирования .....	16
1.4.1 Интегрированные решения .....	17
1.4.2 Математические модели .....	18
1.4.3 Точное моделирование .....	19
1.5 Способы взаимодействия с внешним кодом .....	20
2 Проектирование решения .....	23
2.1 Требования к проекту .....	23
2.2 Обзор выбранных технологий .....	24
3 Обзор разработанного решения .....	27
3.1 Общая архитектура проекта .....	27
3.2 Архитектура плагина .....	31
3.2.1 Конфигурация .....	31
3.2.2 Компиляция HDL .....	36
3.2.3 Процесс генерации кода .....	38
3.3 Использование модели .....	41
3.3.1 Условия использования .....	42
3.3.2 Простой пример .....	44
3.3.3 Пример с Controls.kt .....	45
4 Анализ результатов .....	49
4.1 Сравнение с альтернативными решениями .....	49
4.2 Способы применения .....	51
4.3 Перспективы развития .....	51
ЗАКЛЮЧЕНИЕ .....	53
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	54

ПРИЛОЖЕНИЕ А .....	56
ПРИЛОЖЕНИЕ Б .....	58
ПРИЛОЖЕНИЕ В .....	62

## ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

**Диалект** – условное название специфицированного промежуточного представления в технологии MLIR

**Мультиплатформенный** – (от англ. multiplatform) условное обозначение для программного кода или проекта, который может работать на множестве аппаратно-программных платформ

**Нативный** – (от англ. native — «родной») условное название для компонента, подразумевающее низкоуровневое бинарное представление специфичное для данной платформы, машинный код, который исполняется процессором напрямую без дополнительных слоев интерпретации или трансляции; в контексте платформы JVM также может значить «внешний», то есть сторонний программный код не на языке Java

**Промежуточное представление** – определенный формат данных, используемый для представления данных между разными уровнями системы

**Фреймворк** – инструмент для создания программных систем, который в большей степени контролирует процесс разработки, чем библиотека

**ABI** – двоичный (бинарный) интерфейс приложения (Application Binary Interface), набор правил и соглашения для взаимодействия программных приложений посредством платформозависимого двоичного кода

**API** – программный интерфейс приложения (Application Programming Interface), набор правил для взаимодействия разных программных приложений друг с другом

**CIRCT** – название проекта и набора технологий для работы с моделями цифровой техники, использующих технологии компиляторной инфраструктуры LLVM

**CIRCT IR** – промежуточное представление, набор диалектов задокументированных в проекте CIRCT

**Circulator** – публичное название разрабатываемого проекта.

**Gradle** – система сборки проектов, активно применяющаяся в экосистеме языка программирования Kotlin



**HDL** – язык описания аппаратуры (Hardware Definition Language, Hardware Design Language), язык программирования, позволяющий описывать дизайн моделей аппаратуры с использованием специфичных для этого конструкций; используется для дальнейшего производства аппаратных модулей

**JVM** – виртуальная машина Java (Java Virtual Machine), технология разработанная компанией Oracle

**JDK** – пакет инструментов разработки приложений на языке Java, специализируется версией

**Kotlin** – современный язык программирования, поддерживающий возможность мультиплатформенной разработки

**LLVM** – технология предоставляющая открытую инфраструктуру для построения промышленных компиляторов

**LLVM IR** – базовое промежуточное представление используемое в проекте LLVM.

**MLIR** – технология построения промежуточных представлений и выполнения преобразований между ними, основанная на базе компиляторной инфраструктуры LLVM

**Native** – условное название целевой платформы исполнения кода в мультиплатформенных проектах на базе Kotlin Multiplatform

**WASM** – название технологии и целевой платформы, часто используемой для выполнения кода в веб-браузере

## ВВЕДЕНИЕ

При построении сложных систем, немаловажным является вопрос их тестирования, при этом тестирование с использованием реального оборудования может быть очень рискованно — приборы могут быть дорогостоящими или даже уникальными, а выход из строя хотя бы одного из них приведет к значительным временным потерям, помимо финансовых затрат. Отсюда возникает необходимость в использовании «виртуальных устройств», которые создают уровень абстракции между системой управления и оборудованием, что позволяет незаметно для компонентов системы подменять реализацию устройства. Например, виртуальное устройство имитирующее поведение аппаратного модуля можно переключить на реальный прибор в нужный момент.

Моделирование работы прибора может производиться различными способами, но в любом случае эта задача требует дополнительных усилий по разработке модели и поддержания ее актуальности. Кроме того, чем точнее модель и сложнее устройство, тем труднее ее разработать человеческими усилиями, что серьезно затрудняет или даже делает невозможным сквозное тестирование модели системы, если же прибор во время построения системы находится в разработке, то затраты на поддержание актуальности модели также возрастают.

В данной выпускной квалификационной работе исследуются пути решения и рассматривается разработанное решение для задачи моделирования аппаратуры.

## 1 Обзор предметной области

В данной главе дается описание предметной области, рассматриваются имеющиеся решения и технологии, которые важны при проектировании собственного решения.

### 1.1 Область применения

Целевой областью применения разрабатываемого решения является аппаратно-программная система, обладающая следующими свойствами:

- поддерживает компонентное моделирование,
- не требовательная к реальному времени, то есть не критичная к производительности моделей,
- поддерживающая разработку на высокоуровневом языке программирования Kotlin или другом языке JVM-платформы.

В частности, такие системы разрабатываются с использованием фреймворка «Controls.kt».

«Controls.kt» — это фреймворк на языке программирования Kotlin, разрабатываемый Центром научного программирования МФТИ. [2] Фреймворк базируется на ядре фреймворка для работы с данными — DataForge, который разрабатывался совместно с подразделением JetBrains Research [9]. Controls.kt предназначен для создания «легковесных» SCADA-систем, а его особенность заключается в том, что он использует асинхронный подход для коммуникации между устройствами, что расширяет границы применимости фреймворка. [8, 10] Кроме того, фреймворк дает необходимые абстракции для создания и использования виртуальных устройств.

Наиболее известным примером применения «Controls.kt» является установка, использовавшаяся в научном эксперименте по изучению нейтрино «Troitsk nu-mass, experiment» [11].

В настоящий момент вектор развития фреймворка направлен на моделирования систем, примеры которых доступны в репозитории проекта. [5] Для обеспечения полного сквозного тестирования разрабатываемых систем, необходимо точное соответствие поведения модели и реального устройства, поэтому разработчики фреймворка нуждаются в подходе, позволяющем мо-

делировать устройства на уровне цифровых схем. Такой подход позволяет повысить качество работы разрабатываемой системы и снизить затраты за счет раннего обнаружения проблем в дизайне аппаратных компонентов.

Использование асинхронного режима работы в фреймворке Controls.kt накладывает сложности на его реализацию. Фреймворк берет на себя решение проблемы согласованности «виртуального» времени отдельных приборов, что позволяет интегрировать в одну систему устройства разных типов, в частности, это позволяет использовать модели цифровых схем приборов, несмотря на их низкую производительность и отзывчивость в сравнении с реальными приборами.

## 1.2 Языки описания аппаратуры

Языки описания аппаратуры — HDL — играют ключевую роль в проектировании и разработке цифровых систем. Они позволяют инженерам описывать поведение и структуру аппаратных компонентов на различных уровнях абстракции, начиная от логических вентилях и заканчивая сложными системами на кристалле.

Наиболее популярными языками HDL являются Verilog, SystemVerilog, VHDL, также встречаются примеры использования SystemC и Chisel. Кроме того, разрабатываются новые языки, например, Verik. Каждый из них имеет свои особенности и области применения.

**Verilog и SystemVerilog** — индустриальные стандарты, широко используемые для проектирования и верификации цифровых систем. SystemVerilog является расширением Verilog и предоставляет дополнительные возможности, благодаря чему имеет большую популярность.

**VHDL** — ещё один популярный язык описания аппаратуры и индустриальный стандарт с большой историей, который отличается от семейства Verilog большей сложностью чтения кода.

**SystemC** — библиотека и доммено-ориентированный язык описания систем на основе языка программирования C++. Инструмент предназначен для моделирования аппаратных систем на уровне системного проектирования. SystemC позволяет интегрировать моделирование аппаратуры и

программного обеспечения, что делает его особенно полезным для проектирования встраиваемых систем. Также он является языком высокоуровневого синтеза (HLS, High-level synthesis) для некоторых специализированных средств проектирования. [17]

**Chisel** — плагин для компилятора и доменно-ориентированный язык описания аппаратуры, основанный на языке программирования Scala. [7] Chisel является новым по сравнению с другими классическими HDL, он предоставляет высокоуровневый функциональный подход к проектированию цифровых систем и использует промежуточное представление для интеграции со сторонними инструментами. Этот язык широко применяется для генерации аппаратных блоков различных процессоров.

**Verik** — новый развивающийся язык описания аппаратуры, основанный на Kotlin. Verik ориентирован на упрощение проектирования цифровых систем и интеграцию с современными инструментами разработки. Инструмент использует язык SystemVerilog как промежуточное представление: позволяет импортировать модули на этом языке и использует этот язык как результат компиляции. Однако он не позволяет использовать разработанные модели как либо еще. [15]

Представленный список языков для описания аппаратуры не является полным, но он содержит наиболее популярные и значимые для разрабатываемого проекта языки.

### 1.3 Промежуточное представление цифровых схем

В силу специфики области разработки аппаратуры, в ней слабо развиты технологии использования промежуточных представлений, в сравнении с областью разработки программного обеспечения. В связи с этим, современные языки описания аппаратуры предлагают свои варианты промежуточного представления.

Основным промежуточным представлением, используемым в области синтеза цифровых схем является **Netlist**, который представляет собой довольно примитивное представление соединений операционных элементов схемы. Открытые средства работы с таким представлением не развиты. [16]

Другим вариантом промежуточного представления может служить сам язык **Verilog** (или SystemVerilog), который является достаточно низкоуровневым. Для него имеется обширный набор средств анализа, как проприетарных, так и открытых. Однако, этот язык не разрабатывался в качестве промежуточного представления, из-за чего его обработка и разносторонний анализ оказываются затруднительны. Тем не менее, некоторые языки используют его как результат компиляции, например, Verik, о котором упоминалось, язык Chisel для симуляции схем, SystemC для синтеза.

Примером более гибкого и современного подхода к созданию промежуточного представления служит проект LLVM CIRCT. **LLVM CIRCT** (Circuit IR Compilers and Tools) — это проект, направленный на создание инфраструктуры для проектирования цифровых схем с использованием промежуточного представления (IR). CIRCT базируется на MLIR (Multi-Level Intermediate Representation), что позволяет использовать модульный и расширяемый подход к описанию и трансформации цифровых схем. [1] Диаграмма иллюстрирующая ключевые технологии проекта представлена на рисунке 1.

К основным компонентам CIRCT относятся следующие:

- **MLIR**: основа для создания диалектов, которые описывают различные аспекты цифровых схем.
- **Диалекты**: включают HW (Hardware), SV (SystemVerilog), FIRRTL и другие, которые позволяют описывать схемы на различных уровнях абстракции в зависимости от требований изначального языка.
- **Инструменты трансляции**: такие как firtool для преобразования FIRRTL в SystemVerilog или диалекты CIRCT.

CIRCT предоставляет мощные возможности для оптимизации и анализа цифровых схем, а также для интеграции с существующими инструментами проектирования. Так, например, для языка Chisel разрабатываются диалекты firrtl и cirrtl, которые позволяют точнее сохранить в семантику языка.

Кроме указанных инструментов, с проектом CIRCT можно рассмотреть отдельные инструменты для трансляции указанных языков описания аппаратуры в промежуточное представление диалектов CIRCT. Для языка Verilog

разрабатывается современный фронтенд SVlang, который используется для трансляции утилитой `circt-verilog`.

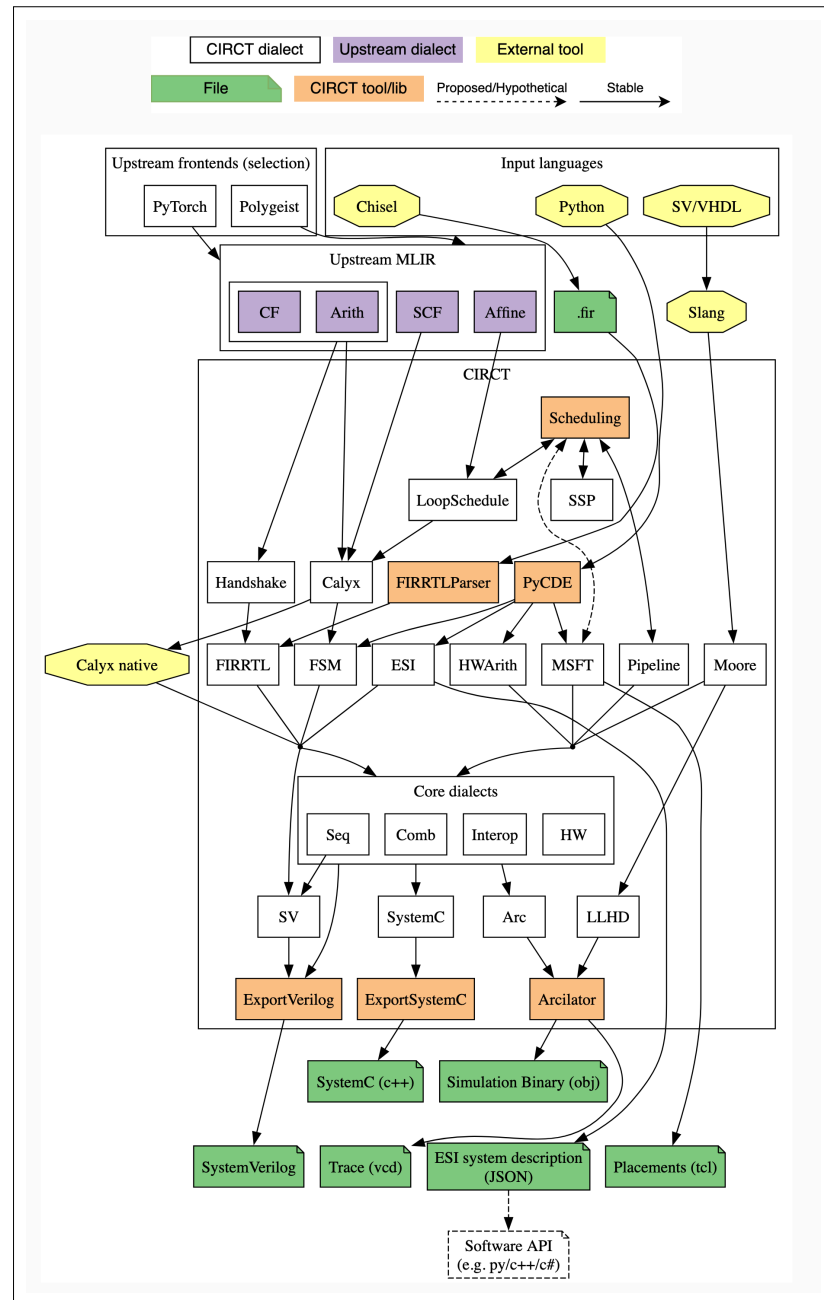


Рисунок 1 — Ключевые технологии проекта CIRCT, включая связанные сторонние инструменты

## 1.4 Средства моделирования

Для построения моделей аппаратуры существует множество готовых решений и подходов. Условно их можно разделить на категории:

- интегрированные решения, предоставляющие обширные возможности по разностороннему моделированию,
- простые математические и функциональные модели,
- точные модели, т.н. «cyclic-accurate».

#### 1.4.1 Интегрированные решения

Широко распространены интегрированные решения для моделирования, которые зачастую разрабатываются крупными компаниями — лидерами на своем рынке. Среди прочих выделяются следующие решения, пригодные для моделирования аппаратуры и программно-аппаратных систем.

- **Modelsim** — популярный инструмент для симуляции цифровых схем. Поддерживает Verilog, VHDL и SystemVerilog. Обладает высокой точностью моделирования и возможностью интеграции с другими инструментами проектирования. Стоимость лицензии может быть высокой, что делает его доступным в основном для крупных компаний.
- **MATLAB Simulink** — развитый инструмент для моделирования и симуляции систем. Поддерживает генерацию HDL через дополнительные модули. Хорошо интегрируется с другими инструментами MATLAB. Стоимость лицензии высокая, но инструмент популярен в академической и инженерной среде.
- **LabVIEW** — инструмент для визуального программирования и моделирования. Подходит для управления оборудованием и симуляции. Генерация HDL возможна через дополнительные модули. Стоимость лицензии высокая, но инструмент широко используется в промышленности.
- **Engae** — российская среда динамического моделирования и технических расчётов. Инструмент предоставляет возможности по моделированию систем как с помощью визуального программирования функциональными блоками, так и с применением языка программирования Julia. На официальном сайте представлены примеры использования системы для генерации кода на языке Verilog.
- **Quartus Prime** — инструмент от компании Intel для проектирования программируемых логических интегральных схем (ПЛИС). Поддерживает



генерацию HDL и интеграцию с аппаратными платформами. Бесплатная версия доступна, но с ограничениями. Популярен среди разработчиков FPGA.

- **Vivado** — инструмент от компании Xilinx для проектирования ПЛИС. Поддерживает генерацию HDL и интеграцию с аппаратными платформами. Бесплатная версия доступна, но с ограничениями. Широко используется в индустрии.

Каждое из этих решений имеет свои особенности, и выбор зависит от требований проекта, бюджета и уровня интеграции с существующими системами. Но все они так или иначе используют язык Verilog (SystemVerilog) как промежуточное представление для моделей и высокоуровневого синтеза. Эти решения являются проприетарными и у них ограниченные возможности по интеграции разработанных моделей в сторонние системы.

#### 1.4.2 Математические модели

Другим подходом к моделированию является математическое моделирование с применением т.н. передаточных функций. Основные качества подхода с использованием передаточных функций:

- **простота** — передаточные функции предоставляют компактное и удобное представление системы в виде алгебраического выражения, поддающегося методам математического анализа,
- **точность** — при правильной настройке передаточные функции могут точно описывать поведение системы в определенных условиях, однако точность зависит от уровня абстракции и предположений, сделанных при создании модели,
- **эффективность** — математические модели, как правило, обладают высокой вычислительной эффективностью, так как они оперируют упрощенными представлениями системы, что позволяет значительно сократить время расчетов. Математические модели могут быть выполнены быстрее за счет использования аналитических методов и упрощений,
- **ограниченность** — математическим моделям свойственно не учитывать всех деталей и нюансов работы реальных устройств, они ведут себя

как эталонная реализация, что мешает проводить сквозное тестирование системы с использованием таких моделей.

Таким образом, подход с использованием передаточных функций является мощным и удобным инструментом для моделирования приборов, при модульном, функциональном и интеграционном тестировании, но не подходит для целей сквозного тестирования.

#### 1.4.3 Точное моделирование

Точное моделирование популярно в сфере разработки аппаратуры. Многие интегрированные решения, в том числе описанные ранее, обладают такими возможностями. Однако, их использование в открытом проекте затруднительно. Среди свободных решений для точного («cyclic-accurate») моделирования можно выделить следующие.

SystemC, как уже говорилось ранее, подходит для создания моделей систем, он также позволяет производить точное моделирование работы аппаратных модулей и использует собственную реализацию операционного окружения для этого. В результате компиляции проекта на SystemC получается исполняемый файл, который при запуске выполняет симуляцию работы системы. [17] SystemC предоставляет возможность для интеграции с итоговыми моделями системы, но всё же этот подход требует высоких трудозатрат.

**Icarus Verilog** — это бесплатный инструмент для симуляции Verilog. Подходит для небольших проектов и обучения. Он очень ограничен в функциональности и возможности интеграции с ним.

**Verilator** — это бесплатный инструмент с открытым исходным кодом для симуляции Verilog. Инструмент позволяет транслировать код на языке Verilog в код на языке C++. В результате компиляции полученных исходных кодов на языке C++ получается исполняемый файл, который выполняет высокопроизводительное точное моделирование работы аппаратного модуля. Как и в случае с SystemC, инструмент предлагает интерфейс для взаимодействия с итоговыми моделями, который выглядит сложным для использования этого решения во внешней системе. Verilator является популярным решением

в своей сфере и активно развивается. Например, язык Chisel использует этот инструмент для симуляции работы модели.

**Arcilator** — это инструмент в составе проекта CIRCT, предназначенный для анализа и оптимизации цифровых схем и представляющий собой утилиту командной строки `arcilator`. Он предоставляет возможности для трансформации и симулирования работы цифровых схем, описанных на различных уровнях абстракции с использованием ключевых диалектов MLIR из проекта CIRCT. [6, 14]

Инструмент позволяет конфигурировать процесс трансляции промежуточного представления CIRCT в промежуточное представление LLVM, в зависимости от требований к нативной модели, например, можно указать опции, добавляющие возможность наблюдения за внутренними состояниями аппаратного модуля, что может быть важно при его тестировании и отладке.

Также немаловажным является то, что итоговая модель представляет собой функцию, которая получает исходное состояние модели и обновляет его в соответствии с логикой работы устройства, т.е. не содержит дополнительной среды выполнения, с которой пришлось бы взаимодействовать, а поддерживает ABI для прямых вычислений. Такая особенность инструмента позволяет использовать его из внешних систем, например, загружая скомпилированный модуль динамически с любым наперед неизвестным количеством приборов: нужно лишь создать структуру состояния для каждого.

### 1.5 Способы взаимодействия с внешним кодом

Область применения фреймворка определяет основную целевую платформу разрабатываемого решения — JVM. Средства симуляции предоставляют модели в виде нативных модулей. Поэтому важно найти способ взаимодействия между этими компонентами.

Интеграция с нативным кодом из Kotlin/JVM может быть выполнена различными способами, в зависимости от требований проекта и используемых технологий.

**JNI** (Java Native Interface) — технология платформы JVM предоставляющая стандартный способ взаимодействия между Java (и Kotlin/JVM) и

нативным кодом, написанным на C или C++. Этот подход позволяет вызывать нативные функции из JVM и наоборот.

– Преимущества:

1. Широкая поддержка и документация;
2. Возможность работы с любыми библиотеками на C/C++;
3. Доступно на старых версиях JDK.

– Недостатки:

1. Требуется написания кода на C/C++ обрабатывающего нативные вызовы;
2. Требуется ручного управления памятью.

**IPC** (Inter-Process Communication) — межпроцессное взаимодействие используется для взаимодействия между процессами, когда нативный код выполняется в отдельном процессе. Это может быть полезно для повышения безопасности и изоляции.

– Преимущества:

1. Изоляция процессов повышает стабильность;
2. Подходит для взаимодействия с нативными сервисами.

– Недостатки:

1. Более высокая задержка из-за межпроцессного взаимодействия;
2. Сложность настройки.

**Kotlin/Native** позволяет компилировать Kotlin-код в нативный бинарный код, что упрощает взаимодействие с нативными библиотеками. Это особенно полезно для мультиплатформенных проектов.

– Преимущества:

1. Естественная интеграция с Kotlin;
2. Существуют инструменты автоматизирующие интеграцию с JVM.

– Недостатки:

1. Ограниченная поддержка JVM-специфичных функций;
2. Такие же сложности как и с обычным JNI;
3. Требуется использования среды исполнения Kotlin/Native.

**FFM API** — это современный способ взаимодействия с нативным кодом, официально предоставляемый в JDK, начиная с версии 22. Он

позволяет работать с внешней памятью и вызывать внешние функции без использования JNI.

– Преимущества:

1. Повышенная производительность;
2. Упрощенная работа с нативной памятью.

– Недостатки:

1. Требуется использование последних версий JDK.

Прямое использование FFM API позволяет вручную вызывать нативные функции и управлять памятью, но в силу особенностей языка Java требуется довольно много дополнительного программного кода.

Проект Panama по развитию возможностей работы с внешним кодом в JVM, частью которого является FFM API, предоставляет дополнительный инструментарий для повышения удобства работы с нативными библиотеками. [12]

**Jextract** — инструмент для автоматической генерации кода на языке Java, выполняющего обращение к нативным библиотекам. В качестве основы он берет имеющиеся заголовочные файлы на языке C. Код, генерируемый этим инструментом, оказывается громоздким и плохо читаемым, и поэтому инструмент лучше всего подходит в тех случаях, когда нужно автоматически адаптировать большое количество кода, что не получится сделать эффективно другими средствами.

Также существуют решения от сторонних разработчиков, позволяющие автоматически генерировать код для работы с FFM API на базе аннотаций — проект **Java Native Memory Access**. Это решение так же полагается на наличие заголовочного файла на языке C, описывающего интерфейс работы с нативным модулем.

Каждый из описанных подходов имеет свои особенности, и выбор сильно зависит от требований проекта и сценариев использования.

## 2 Проектирование решения

В данной главе рассматривается процесс выявления требований к проекту, анализ этих требований и архитектурные решения о выборе используемых для реализации проекта технологий.

### 2.1 Требования к проекту

Со стороны разработчикой фреймворка «Controls.kt» не выдвигалось явных требований к реализации модуля моделирования аппаратуры. Тем не менее, исходя из направленности фреймворка и общих представлений о разработке аппаратуры, разработчики дали следующие рекомендации:

1. Программному модулю следует использовать промежуточное представление цифровых схем для создания моделей.
2. Программный модуль следует сделать проектом, отдельным от «Controls.kt».

Первая рекомендация говорит о том, что программный модуль должен использовать промежуточное представление как интерфейс для возможности интеграции с ним. Вторая рекомендация говорит о том, что программному модулю не следует добавлять зависимость на «Controls.kt», ведь он может быть полезен вне контекста фреймворка.

Исходя из этих рекомендаций и свойств целевой области применения, были сформулированы следующие требования к программному модулю:

1. Программный модуль должен быть разработан на языке программирования Kotlin.
2. Программный модуль должен следовать принципам мультиплатформенной разработки на языке Kotlin.
3. Программный модуль должен предоставлять абстракции для работы с моделями аппаратуры.
3. Программный модуль должен использовать промежуточное представление для описания аппаратуры.
4. Программный модуль должен генерировать код на языке Kotlin для использования конкретных моделей аппаратуры на базе промежуточного представления.

5. Программный модуль должен давать возможность конфигурации процесса генерации кода.

Первое требование отвечает ключевой технологии с помощью которой реализован фреймворк «Controls.kt». Язык Kotlin выбран потому, что это современный язык программирования, предоставляющий простой интерфейс взаимодействия с JVM платформой и обладающий большей выразительностью в сравнении с языком Java. [18] Закономерно выдвинуто второе требование: целевой фреймворк следует принципам мультиплатформенной разработки, и существует потенциал реализовать проект с поддержкой платформ, отличных от JVM, например, Native или WASM.

Третье требование отвечает первоначальной рекомендации; четвертое требование является осмысленным по той причине, что промежуточного представления вполне достаточно для автоматического определения интерфейса взаимодействия с моделируемым устройством, и создание корректного программного интерфейса является сложной задачей для человека. Пятое требование призвано добавить гибкости процессу генерации кода, что должно позволить расширить возможности по использованию полученных программных моделей и удобство структурирования проекта разрабатываемой системы.

Готовые решения для моделирования, такие как Modelsim, LabVIEW и другие описанные в разделе 1.4, решено не использовать по той причине, что они являются проприетарными, не дают требуемой гибкости в создании и использовании моделей, и их использование приведет к ограничению возможностей по расширению разрабатываемого программного решения, например, потребует использования определенного средства моделирования.

В соответствии с описанными требованиями был определен набор технологий для реализации проекта.

## **2.2 Обзор выбранных технологий**

Сборка проекта происходит с использованием инструмента Gradle, так как он рекомендуется в официальной документации к языку Kotlin для мультиплатформенных проектов. [13] По той же причине этот инструмент

выбран для реализации компонента выполняющего генерацию кода: компонент должен иметь возможность встраиваться в сборочный процесс. [3] Непосредственно для генерации кода используется программная библиотека Kotlin Poet, которая предоставляет программный интерфейс для управления процессом генерации кода. [4]

В качестве промежуточного представления принято решение использовать семейство диалектов MLIR описанных в проекте CIRCT по той причине, что проект активно развивается и такое промежуточное представление является наиболее гибким для анализа, оптимизаций и другой обработки с помощью открытых компиляторных технологий из проекта LLVM. Кроме того, имеются или находятся в разработке инструменты для получения этого промежуточного представления из популярных языков описания аппаратуры, например, SystemVerilog и Chisel, что также позволяет использовать другие языки, такие, как Verik. Существуют инструменты позволяющие транслировать это промежуточное представление в SystemVerilog.

Для получения нативной модели из промежуточного представления используется инструмент Arcilator, так как он предоставляет готовый пользовательский интерфейс для использования набора преобразований необходимых для получения LLVM IR из CIRCT IR, который далее используется для компиляции в нативный код либо статической, либо динамической библиотеки. Этот инструмент выбран в большей степени в силу своей простоты использования, но он не является критичным для реализации; Arcilator вполне можно заменить на другие инструменты предоставляющие возможность получения нативных моделей из описания аппаратуры, например, более популярный Verilator или SystemC, но они не рассчитаны на работу с промежуточным представлением, из-за чего очень специализированны для конкретного языка описания аппаратуры и, кроме того, влекут использование их собственной среды исполнения с усложненным ABI.

Компиляция LLVM IR в нативный код может происходить с использованием стандартных инструментов проекта LLVM. Лучше всего для этого подходит драйвер clang, который предоставляет пользовательский интерфейс



для работы с другими утилитами, такими как оптимизатор и компилятор LLVM IR. Также он позволяет сразу получить нативные модели в виде динамически загружаемых библиотек.

Взаимодействие с нативным кодом может происходить различным образом, в зависимости от требований проекта и платформы, используемой для реализации системы. Так как фреймворк «Controls.kt» в основе своей нацелен на использование платформы JVM, в разрабатываемом решении так же сделан упор на эту платформу.

Для обращения из JVM к нативному коду моделей принято решение использовать FFM API, так как он является наиболее современным и производительным. Эта технология в большей степени выбрана в силу своей простоты с точки зрения использования, но она не является критичной. Для ее удобного использования решено было реализовать слой абстракций, реализующих логику работы с внешним кодом и предоставляющих упрощенный программный интерфейс; эта разработка оказывается оправданна, так как Arcilator предоставляет стабильный ABI и для готовых инструментов требуется наличие заголовочных файлов на языке C.

Тогда общий принцип получения программных моделей на базе языка описания аппаратуры состоит в следующем:

1. Трансляция языка описания аппаратуры в промежуточное представление.
2. Обработка и анализ промежуточного представления с дальнейшей компиляцией его в LLVM IR.
3. Генерация программного кода на языке Kotlin на основе полученного анализа.
4. Подключение нативных библиотек, реализующих логику работы устройств к системе.

Таким образом, для реализации проекта выбраны технологии, которые позволяют выполнить поставленные требования наиболее оптимально для его целевой области применения.

### 3 Обзор разработанного решения

В данной главе рассматривается реализация разработанного инструмента Circulator. Также в ней рассматриваются условия и примеры его использования для проведения сквозного тестирования.

Проект выполнен в соответствии с принципами мультиплатформенной разработки на языке Kotlin и представляет собой программную библиотеку и плагин для системы сборки Gradle. Проект структурирован как композитный проект в системе сборки Gradle и содержит следующие модули.

- `circulator-core` — ядро проекта, включаемое во все подпроекты Circulator. Оно реализуется как программная библиотека, предоставляющая абстракции для описания и использования моделей аппаратуры, а также структуры данных для работы со средствами моделирования (Arcilator для указанной версии).
- `circulator-plugin` — реализация плагина для системы сборки Gradle. Он принимает конфигурацию из сценария сборки, дополнительно конфигурирует проект и генерирует код на языке Kotlin.
- `demo` — проект с подпроектами, демонстрирующими возможности использования инструмента.

#### 3.1 Общая архитектура проекта

Модуль `demo` включает в свой процесс сборки корневой проект, что важно для использования актуальной версии плагина выполняющего генерацию кода. Несмотря на неочевидную зависимость подпроектов, такая конфигурация сборочного процесса оказывается удобной во время разработки, так как при использовании демонстрационного проекта в качестве корневого, среда разработки и система сборки правильно разрешает зависимости.

Визуализация размещения модулей представлена на диаграмме 2. Пунктиром на ней изображены коллекции исходных файлов отвечающие за конкретную платформу, а стрелки указывают направление зависимости.

Центральный модуль проекта `circulator-core` предоставляет базовые абстракции для создания модели прибора:

- класс `Model` для определения интерфейса для работы с моделью прибора,
- класс `ModelLibrary` для определения интерфейса для работы с нативной библиотекой,
- классы делегатов для получения внутреннего состояния прибора, следующие интерфейсу `StateProjection<T>`.

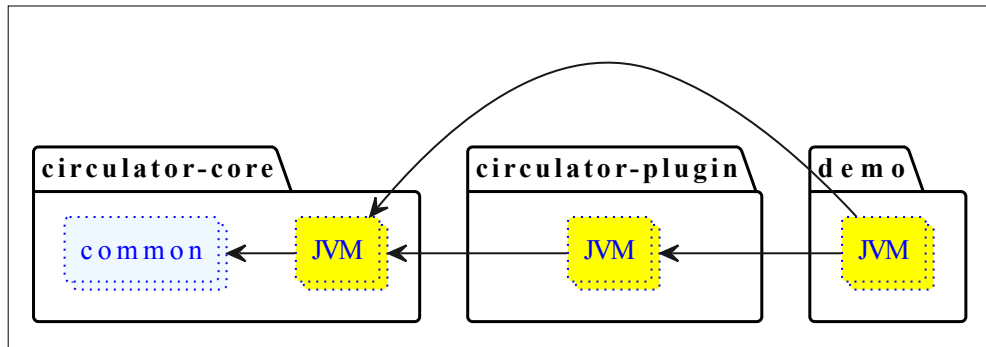


Рисунок 2 — Диаграмма структуры проекта Circulator, иллюстрирующая зависимости между модулями

Диаграмма классов, визуализирующая взаимосвязь между компонентами, представлена на рисунке 3. Классы реализующие интерфейс `StateProjection<T>` выполняют доступ к данным, расположенными во внешней памяти, и к ним делегируется обращение к свойствам класса модели. Для примитивных типов таких, как `Byte`, `Int`, `Long` и пр., отдельно реализованы классы, потому, как доступ к сегменту памяти происходит не полиморфным, а зависимым от типа, образом.

Указанные классы оказываются сильно зависимыми от платформы JVM, так как полагаются на пакет `java.lang.foreign` для работы с нативным кодом. В результате, в мультиматформенной части проекта оказывается класс определяющий схему данным для работы с утилитой `arcilator` и базовый набор типов состояний, наблюдаемых в модели. При дальнейшем развитии проекта предполагается реализация обобщенного мультиплатформенного подхода для определения классов моделей, что также может быть полезно для использования с другими инструментами для создания нативных моделей.

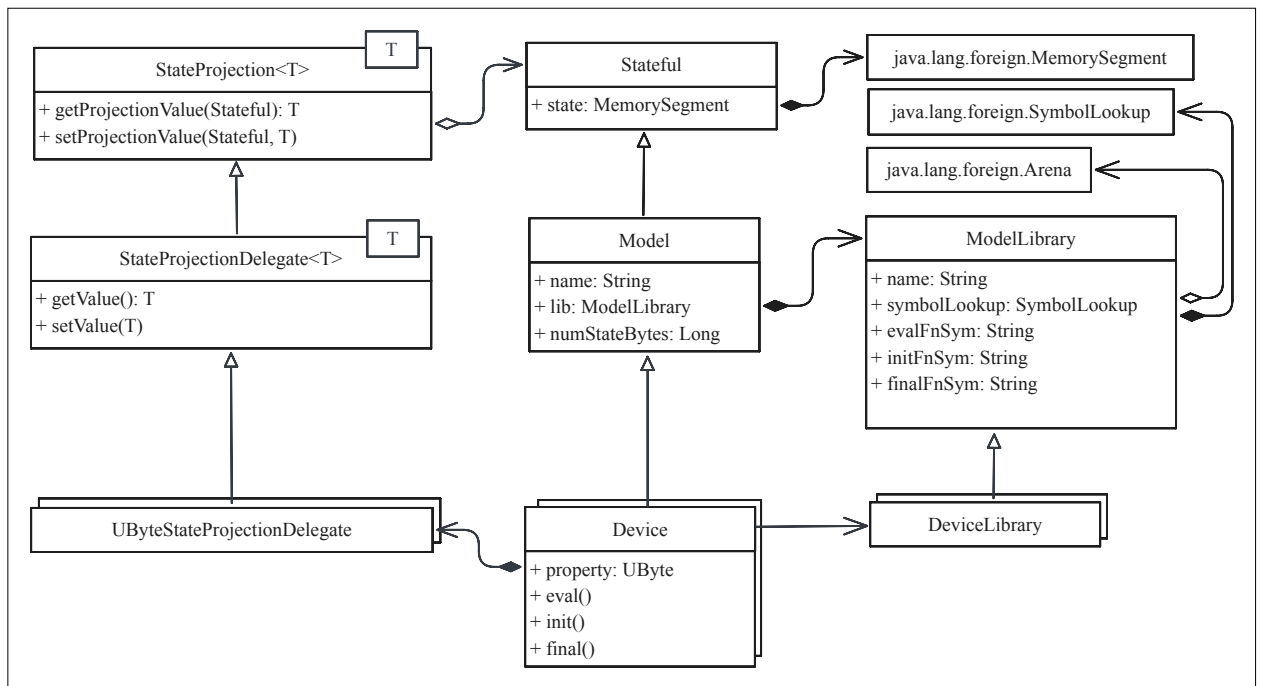


Рисунок 3 — Диаграмма ключевых классов, лежащих в основе Circulator

Модель устройства содержит сегмент нативной памяти, в котором сохраняется состояние модели. Размер этого сегмента определяется из конфигурации. Непосредственно с памятью работают объекты-делегаты следующие интерфейсу `StateProjection<T>`, где `T` является параметром типа, соответствующим типу проекции состояния из конфигурации.

Для каждого стандартного типа проекции, унаследованного из проекта Arcilator, имеется соответствующая функция-конструктор, общий вид сигнатуры которых можно видеть на листинге 1. Такая функция позволяет указать номер байта в сегменте памяти, соответствующего началу данных, относящихся данной проекции. Используя эти функции, становится возможно компактно определить интерфейс модели — методы для чтения и записи данных («getter» и «setter») свойства класса соответствующие проекции состояния делегируются объекту (средствами языка), полученному из функции.

```

fun <T> Model.input(offset: Long) = // ...
fun <T> Model.output(offset: Long) = // ...
fun <T> Model.register(offset: Long) = // ...
fun <T> Model.memory(offset: Long, layout: /* ... */) = // ...
fun <T> Model.wire(offset: Long) = // ...

```

Листинг 1 — сигнатура функций, использующихся для создания проекций состояния

Соответственно количеству бит для хранения значения в проекции состояния, указанному в конфигурации, используется наименьший по размеру тип языка Kotlin допускающий весь диапазон значений. Соответствие типов можно видеть в таблице 1.

Таблица 1 — Соответствие типов данных при генерации кода

Количество бит	Тип Kotlin
[1; 8]	UByte
[9; 16]	UShort
[17; 32]	UInt
[33; 64]	ULong

После изменения значения свойства объекта необходимо выполнить расчет нового состояния модели, вызвав метод `eval()`. Последовательность операций производимых в системе можно представить в виде диаграммы на рисунке 4.

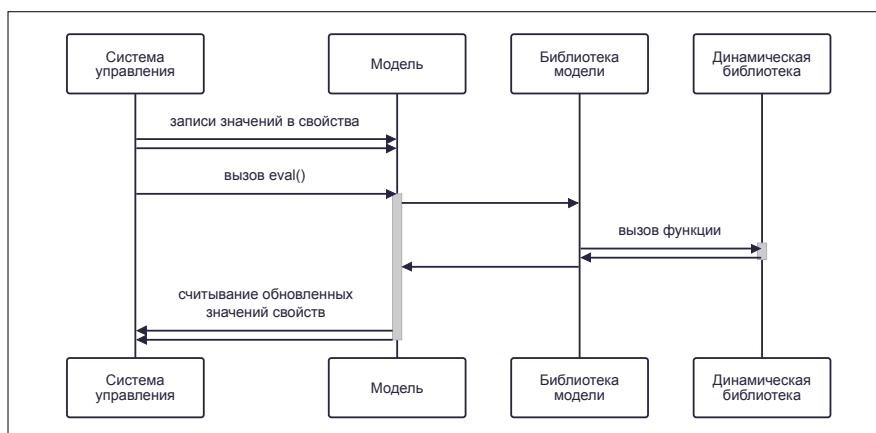


Рисунок 4 — Диаграмма последовательности выполнения операций при работе с моделью

## 3.2 Архитектура плагина

Плагин для системы Gradle является ключевой составляющей разрабатываемого инструмента. Именно плагин выполняет необходимые действия для получения рабочей модели устройства. Процесс работы с плагином состоит из следующих этапов:

1. Конфигурация плагина;
2. Компиляция языка описания аппаратуры;
3. Генерация кода;
4. Использование полученной модели.

Для каждого этапа были проработаны нюансы его прохождения и разработана методика действий; все они нашли отражение в архитектуре плагина.

### 3.2.1 Конфигурация

Конфигурация плагина производится через интерфейс системы сборки Gradle, то есть в файле конфигурации `build.gradle.kts`, который находится в корне проекта подлежащего сборке, согласно документации Gradle.

Плагин необходимо подключить в процесс сборки, указав его идентификатор в соответствующей секции файла конфигурации, то есть так, как показано на листинге 2.

```
plugins {  
    // ... other plugins  
    id("io.github.elturin.circulator.plugin") version "0.1.0"  
}
```

Листинг 2 — Подключение плагина в системе сборки Gradle

Плагин принимает конфигурацию в секции своего расширения так, как это показано на листинге 3. Конфигурация плагина сознательно вынесена в отдельный файл, потому что она может выглядеть громоздко в конфигурации сборки проекта, а также такое решение позволяет разделить конфигурации моделей и проекта.

```
circulator {  
    config = file("src/jvmMain/resources/circulator/config.json5")  
}
```

Листинг 3 — Настройка плагина Circulator в секции расширения

Схема данных конфигурации плагина ссылается на формат файла состояний утилиты `arcilator`. Файл можно получить запустив `arcilator` для модели в формате CIRCT IR с флагом `--state-file`, указав путь до выходного файла так, как это показано на листинге 4. Этот файл содержит описание характеристик модели полученной на вход утилиты. Пример содержимого файла представлен на листинге в приложении А.

```
arcilator --state-file=states.json --emit-llvm -o counter.ll counter.mlir
```

Листинг 4 — Пример использования утилиты `arcilator`

Формат файла не задокументирован и для того, чтобы понять его формат, потребовалось изучить исходный код утилиты. В результате была выяснена схема данных этого файла, которая зафиксирована в файле `ArcilatorStateFileSchema.kt` и `StateProjectionType.kt` проекта Circulator с использованием типов данных Kotlin. Содержимое указанных файлов можно видеть на листингах 5 и 6 соответственно.

Формат файла состояний содержит следующие важные атрибуты:

- `name` в описании модели — название модели;
- `numStateBytes` — размер структуры в байтах, используемой для хранения состояния;
- `initialFnSym`, `finalFnSym` — название символов в выходном LLVM IR модуле соответствующих функции инициализации и функции завершения работы модели;
- `states` — массив содержащий характеристики отдельных проекций состояния модели;
- `name` в описании проекции состояния — название проекции;

- `offset` — номер байта, начиная с которого располагается значение проекции состояния в структуре состояния произведенной модели в виде LLVM IR;
- `numBits` — количество бит требующихся для представления всех возможных значений проекции состояния.
- `type` — тип проекции состояния, отвечающий ее назначению.

```
public typealias StateFile = List<ModelInfo>

/**
 * from C++:
 * struct ModelInfo {
 *     std::string name;
 *     size_t numStateBytes;
 *     llvm::SmallVector<StateInfo> states;
 *     // ...
 * };
 */
public data class ModelInfo(
    val name: String,
    val numStateBytes: ULong,
    val initialFnSym: String,
    val finalFnSym: String,
    val states: List<StateInfo>
)

/**
 * from C++ :
 * struct StateInfo {
 *     enum Type { Input, Output, Register, Memory, Wire } type;
 *     std::string name;
 *     unsigned offset;
 *     unsigned numBits;
 *     unsigned memoryStride = 0;
 *     unsigned memoryDepth = 0;
 * };
 */
public data class StateInfo(
    val type: StateProjectionType,
    val name: String,
    val offset: UInt,
    val numBits: UInt,
    val memoryStrides: UInt = 0u,
    val memoryDepth: UInt = 0u,
)
```

Листинг 5 — Классы на языке Kotlin, определяющие схему данных файла состояний Arcilator



```

/**
 * from C++:
 * enum Type { Input, Output, Register, Memory, Wire };
 */
public enum class StateProjectionType {
    INPUT,
    OUTPUT,
    REGISTER,
    MEMORY,
    WIRE
}

```

Листинг 6 — Тип перечисления в языке Kotlin, соответствующий типу перечислени в реализации Arcilator

Расширение принимает путь до конфигурация в формате JSON, в частности JSON5, который позволяет использовать синтаксис для комментариев. В конфигурационном файле указываются настройки для генерации моделей отдельных устройств в формате представленном на листинге 7. Формат JSON выбран среди прочих форматов представления данных как наиболее популярный и простой для понимания.

Основные элементы формата конфигурации представлены с описанием в следующем списке.

- `<model_id_name>` — атрибут с ключом содержащим идентификатор прибора, в значении которого находится конфигурация, в версии Circulator 0.1.0 значение этого ключа не используется, но в дальнейшем его можно использовать для внутренних нужд. Этот атрибут намерено добавлен в значении атрибута `model`, а не атрибутом в элементе массива, чтобы гарантировать уникальность идентификаторов и иметь возможность в редакторах кода искать по ним.
- `package` — атрибут со строковым значением для указания названия пакета, в котором необходимо расположить сгенерированный Kotlin-класс.
- `state-file` — атрибут со строковым значением, содержащим путь до файла проекций состояний.
- `all-states-open`, `all-states-mutable`, `all-states-types` — атрибуты отвечающие конфигурации для всех проекций состояний модели сразу. Атрибуты соответственно отвечают возможности переопределять это свой-

ство класса, записывать в это свойство и набор свойств с фильтрацией по их типу. Значение этого атрибута можно переопределить для отдельной проекции состояния с помощью следующего атрибута.

- `<state_id_name>` — атрибут с ключом, содержащим название проекции состояния, и значением, содержащим конфигурацию для генерации её программного интерфейса.
- `<var_id_name>` — атрибут с ключом, содержащим уникальный идентификатор, и значением, на которое можно использовать в конфигурации. На этот атрибут можно ссылаться в конфигурации.

```
{
  "models": {
    "<model_id_name>": {
      "package": "com.example.generated",
      "state-file": "path/to/arcilator/state-file.json",
      "model": {
        "open": true,
        "all-states-open": true,
        "all-states-mutable": true,
        "all-states-type": [
          "input",
          "output",
          "register",
          "memory",
          "wire"
        ],
        "states": {
          "<state_id_name>": {
            "open": true,
            "mutable": true,
            "access": true
          },
          // ... other states
        }
      },
      "library": {
        "open": true
      },
    },
    // ... other models
  },
  "variables": {
    "<var_id_name>": "<var_value>"
  }
}
```

Листинг 7 — Файл конфигурации плагина Circulator

Большая часть опций обладает значениями по умолчанию, что позволяет сократить размер конфигурации и соблюсти логику работы модели, оставив возможность для изменения, например, по умолчанию все входные порты считаются доступными для записи, а выходные — только для чтения.

В рассматриваемой версии Circulator 0.1.0 отсутствует конфигурация для вспомогательных утилит участвующих в промежуточных этапах получения нативной модели, потому что это выходит за рамки функциональности разрабатываемого программного модуля. Эти этапы сильно зависят от конкретных используемых инструментов и операционной системы, в которой выполняется процесс компиляции. Тем не менее, в будущих версиях такая конфигурация может стать доступна.

### 3.2.2 Компиляция HDL

Процесс компиляции языков описания аппаратуры в нативный код тоже представляет собой многоэтапный процесс, который протекает независимо от Circulator версии 0.1.0. Плагин в составе Circulator создает зависимость своего этапа генерации кода на результат протекания процесса компиляции, в частности на выходной файл состояний утилиты Arcilator.

Можно выделить следующие этапы компиляции, представленные в таблице 2 с описанием промежуточных артефактов.

Таблица 2 — Этапы компиляции с их атрибутами

№	Атрибут	Описание
1	<b>Название</b>	Трансляция языка описания аппаратуры в промежуточное представление.
	<b>Вход</b>	Файл, содержащий исходный код на языке описания аппаратуры.
	<b>Выход</b>	Файл, содержащий промежуточное представление в зависимости от исходного языка.
	<b>Инструмент</b>	Компилятор языка, например, плагин компилятора Scala для Chisel.
	<b>Пример</b>	Chisel в результате компиляции создает файл DeviceName.fir содержащий промежуточное представление в виде FIRRTL; альтернативно язык можно сразу скомпилировать в промежуточное представление одно из диалектов CIRCT: hw, firrtl, или даже SystemVerilog.
	<b>Примечание</b>	Для некоторых языков этот этап может быть пропущен, например, Verilog уже может считаться своим промежуточным представлением.

№	Атрибут	Описание
2	<b>Название</b>	Трансляция промежуточного представления языка в один из основных диалектов CIRCT IR.
	<b>Вход</b>	Файл, содержащий промежуточное представление языка описания аппаратуры.
	<b>Выход</b>	Файл, содержащий промежуточное представление на одном из основных диалектов CIRCT IR.
	<b>Инструмент</b>	Транслятор промежуточных представлений, например, firtool или cirtc-verilog.
	<b>Пример</b>	Промежуточное представление FIRRTL с помощью утилиты firtool транслируется в CIRCT IR диалект hw. Описание аппаратуры на языке Verilog с помощью утилиты cirtc-verilog компилируется в hw диалект CIRCT IR.
	<b>Примечание</b>	Для некоторых языков описания аппаратуры этот этап может быть пропущен, если они уже используют CIRCT IR как промежуточное представление.
3	<b>Название</b>	Компиляция CIRCT IR в симуляционную модель в виде LLVM IR.
	<b>Вход</b>	Файл, содержащий промежуточное представление на одном из основных диалектов CIRCT IR.
	<b>Выход</b>	Файл, содержащий LLVM IR, соответствующий логике симуляции работы модели, и файл, содержащий характеристики симуляционной модели.
	<b>Инструмент</b>	Утилита arcilator.
	<b>Пример</b>	Промежуточное представление устройства в виде CIRCT IR диалекта hw с помощью утилиты arcilator транслируется в LLVM IR, а с опцией --state-file=states.json создается дополнительный файл states.json, содержащий характеристики симуляционной модели устройства.
	<b>Примечание</b>	На этом этапе с помощью отдельных опций можно контролировать процесс трансляции в LLVM IR, отчего будет зависеть набор наблюдаемых состояний в файле характеристики модели.
4	<b>Название</b>	Компиляция симуляционной модели в виде LLVM IR в нативный код динамически загружаемой библиотеки.
	<b>Вход</b>	Файл, содержащий LLVM IR.
	<b>Выход</b>	Файл динамически загружаемой нативной библиотеки для данной операционной системы.
	<b>Инструмент</b>	Утилита clang.
	<b>Пример</b>	Файл, содержащий нативный код симуляционной модели в виде LLVM IR, принимается в качестве аргумента утилитой clang с опциями для создания динамически загружаемой библиотеки (разделяемой библиотеки). Примеры команд с опциями для популярных операционных систем можно видеть на листинге 8.

№	Атрибут	Описание
	Примечание	Этот этап можно выполнить различным образом, в зависимости от требований к итоговому артефакту: можно выполнить дополнительную серию оптимизаций LLVM IR, можно скомпилировать модель в статическую библиотеку, можно объединить несколько моделей в один нативный модуль и т.д.

Соответственно описанной последовательности этапов, после прохождения третьего из них появляется файл состояний симуляционной модели, требующийся для генерации кода на следующем этапе работы с плагином Circulator.

```
# Linux
clang -shared -o libcounter.so counter.ll

# Windows with definition file
clang -shared -o counter.dll counter.ll -WL,/DEF:counter.def
# Windows with manual export
clang -shared -o counter.dll counter.ll -WL,/EXPORT:Counter_eval

# MacOS
clang -nostartfiles -nodefaultlibs -dynamiclib -o libcounter.dylib counter.ll -lSystem
```

Листинг 8 — Команды для компиляции LLVM IR в динамически загружаемую библиотеку в зависимости от операц системы

На последнем, четвертом, этапе получается файл динамически загружаемой библиотеки, который содержит функции, необходимые для работы приложения, использующего требуемые модели аппаратуры.

### 3.2.3 Процесс генерации кода

Процесс генерации кода заключается в создании файлов с исходным кодом на языке Kotlin, реализующих логику взаимодействия с нативными моделями, полученными на предыдущем этапе работы с плагином Circulator.

Генерация кода начинается с получения конфигурации, описывающей требования к результирующим моделям, из файла указанного в конфигурации плагина. Далее в соответствии с ней происходит генерация кода с исполь-

зованием специализированной библиотеки Kotlin Poet, предоставляющей программный интерфейс для описания желаемого кода на языке Kotlin.

При создании свойств генерируемого класса применяется правило именования, при котором входные и выходные порты называются так же, как задано в конфигурации, а имена отвечающих за внутреннее состояние свойств начинается с префикса `internal`, пример сгенерированных свойств можно видеть на листинге 9. Такой подход позволяет избавиться от коллизии имен при генерации класса и добавить семантическое обозначение в имя свойства.

```
public open var clk: UByte by input<UByte>(0)

public open var reset: UByte by input<UByte>(1)

public open val internalClk: UByte by wire<UByte>(2)

public open val internalReset: UByte by wire<UByte>(3)

public open val internalReg: UByte by register<UByte>(5)

public open val internal0: UByte by wire<UByte>(6)

public open val o: UByte by output<UByte>(7)
```

Листинг 9 — Пример сгенерированных свойств класса на языке Kotlin

Плагин Circulator предоставляет дополнительную задачу для системы сборки Gradle, которая находится в группе задач «circulator» и называется «generateKotlinClasses», что можно видеть на листинге 10. При конфигурации проекта плагин добавляет этой задаче зависимость от файла конфигурации и файлов указанных в файле конфигурации, что позволяет отслеживать изменения в используемых артефактах предыдущих этапов.

```

$ ./gradlew tasks --group=circulator
Calculating task graph as no cached configuration is available for tasks: tasks
--group=circulator

> Task :tasks

-----
Tasks runnable from root project 'demo'
-----

Circulator tasks
-----
...
generateKotlinClasses - Generate Kotlin classes for defined models
...

```

Листинг 10 — Вывод утилиты gradle со списком задач в группе «circulator»

Для удобства работы с плагином можно описать этапы компиляции HDL с помощью отдельных задач системы сборки Gradle и добавить задаче «generateKotlinClasses» зависимость на них так, как это показано для примера с Chisel на листинге 11. Подобным образом можно сконфигурировать процесс сборки с другим языком описания аппаратуры.

```

/* in subproject 'chisel' */
val compileChisel = tasks.register<JavaExec>("compileChisel") {
    /* ... */
}
val compileFirrtl = tasks.register<Exec>("compileFirrtl") {
    dependsOn(compileChisel) /* ... */
}
val compileCircuitMlir = tasks.register<Exec>("compileCircuitMlir") {
    dependsOn(compileFirrtl) /* ... */
}
val compileLlvm = tasks.register<Exec>("compileLlvm") {
    dependsOn(compileCircuitMlir) /* ... */
}
val runFullPipeline = tasks.register("runChiselPipeline") {
    dependsOn(compileLlvm) /* ... */
}

/* In parent project */
tasks.named("generateKotlinClasses") {
    dependsOn(":chisel:runChiselPipeline")
}

```

Листинг 11 — Пример конфигурации процесса компиляции в сборочной системе Gradle

В случае использования плагина с языком Chisel в одном проекте есть особенность, связанная с последовательностью выполнения стандартных для Gradle задач — возникает ошибка из-за циклической зависимости задач. Дело в том, что язык Scala, который используется в основе компилятора Chisel зависит от Java классов, которые получаются, в том числе, в результате компиляции классов Kotlin. Но компиляция Kotlin в силу использования плагина Circulator начинает зависеть от компиляции Chisel, и образуется циклическая зависимость «Kotlin — Chisel — Scala — Kotlin». Наиболее простой способ решить эту проблему состоит в том, чтобы сделать отдельный проект, отвечающий за компиляцию Chisel, а в проект использующий плагин Circulator добавить зависимость на результат исполнения его задачи, что продемонстрировано на листинге . Такой подход также решает проблемы возникающие при одновременном использовании плагина Scala и Kotlin Multiplatform для системы сборки Gradle в одном проекте.

### **3.3 Использование модели**

После выполненной генерации кода классы Kotlin становятся доступны для использования в программном коде разрабатываемого приложения по имени пакета, указанного в конфигурации плагина, и имени класса, указанному в файле состояний Arcilator.

В качестве основы для демонстрации работоспособности используется модель счетчика, реализованная на языке описания аппаратуры Chisel, и представленная на листинге 12. В разделе описываются примеры простого приложения командной строки и более сложного графического приложения, представляющего прибор интегрированный в простую систему на базе фреймворка Controls.kt.



```

class CounterChisel extends Module {
  val count = IO(Output(UInt(8.W)))

  val counter = RegInit(0.U(8.W))
  count := counter

  counter := counter + 1.U
}

```

Листинг 12 — Модель счетчика на языке Chisel

В обоих примерах создание модели происходит автоматически из описания прибора на языке Chisel благодаря настроенным задачам и зависимостям между ними, описанными в разделе про компиляцию HDL. Пример вывода утилиты `gradle`, иллюстрирующего сборочный процесс можно видеть на листинге 13.

```

Reusing configuration cache.
...
> Task :sandbox:chisel:compileCounterChisel
> Task :sandbox:chisel:compileFirrtl
> Task :sandbox:chisel:compileCircuitMlir
> Task :sandbox:chisel:compileLlvm
> Task :sandbox:chisel:runChiselPipeline
> Task :sandbox:generateKotlinClasses
> Task :sandbox:compileKotlinJvm
> Task :sandbox:compileJvmMainJava
> Task :sandbox:jvmMainClasses
> Task :sandbox:runJvmCounter
Hello JVM World!
...

```

Листинг 13 — Вывод утилиты `gradle`, демонстрирующий полностью автоматизированный процесс компиляции и генерации кода

### 3.3.1 Условия использования

Для работы с моделью, необходимо выполнить следующие условия.

1. При создании модели должно быть указано **платформонезависимое** имя библиотеки, полученной в результате компиляции на предыдущем этапе.
2. Динамическая библиотека должна быть расположена в известных для JVM путях файловой системы для поиска библиотек под ее **платформозависимым** именем.

3. При запуске JVM должна быть указана опция, разрешающая доступ к нативной памяти.
4. При создании модели должна быть создана арена (экземпляр класса `java.lang.foreign.Arena`) внешней памяти соответствующая цели использования модели.

Первое условие связано с независимым от платформы (операционной системы) исполнением кода в JVM, потому как JVM делегирует поиск и загрузку динамической библиотеки операционной системе, а формат именования библиотек отличается в зависимости от операционной системы.

Второе условие связано с тем, как работает динамический загрузчик библиотек — он ищет лишь в определенных директориях, в том числе тех, которые указываются при запуске JVM в свойстве `java.library.path` и/или переменной окружения в зависимости от операционной системы: `LD_LIBRARY_PATH` на Linux, `DYLD_LIBRARY_PATH` на MacOS. Модель при создании автоматически загружает библиотеку по имени.

Третье условие связано с политикой безопасности при работе с FFM API. Спецификация OpenJDK требует указывать опцию `--allow-native-access` с названием модулей обладающих разрешением.

Четвертое условие связано с поведением JVM при работе с аренами поддерживающими интерфейс `java.lang.foreign.Arena` в многопоточном приложении. FFM API предоставляет три вида арен обладающих различными свойствами, одно из которых возможность доступа из потока отличного от того, которым она была создана. С точки зрения работы моделей приборов, код выполняется в одном потоке независимо от конкретного типа используемой арены. Поэтому, если обращаться к модели из разных потоков, может возникнуть исключение `WrongThreadException`. Таким образом, нужно либо использовать специальный тип арен, доступных из фабричного метода `Arena::ofShared`, либо фиксировать поток, из которого происходит обращение к модели устройства.

### 3.3.2 Простой пример

В качестве простого примера использования счетчика можно привести тест его работоспособности: создание объекта модели, инициализация, использование, проверка количества отсчетов. Исходный код примера приведен на листинге 14.

Аппаратный модуль счетчика реализует функцию подсчета положительных фронтов входного тактового сигнала. Для моделирования поведения тактового сигнала используется функция `CounterChiselModel.tick()`, которая циклически сменяет значение входного сигнала, записывая значения 1 или 0, соответствующие положительному и отрицательному фронту сигналов, в свойство `clock` модели.

```
fun CounterChiselModel.tick() {
    clock = 1
    eval()
    clock = 0
    eval()
}

fun CounterChiselModel.init() {
    reset = 1
    for (i in 1..7) tick()
    reset = 0
}

fun main() {
    Arena.ofConfined().use { arena ->
        val counter = CounterChiselModel.instance(arena, "counterchisel")

        counter.init()

        for (i in 1..10) {
            counter.tick()
        }

        val result = counter.count.toInt()

        assert(result == 10)
    }
}
```

Листинг 14 — Исходный код простого примера, использующего модель счетчика

Инициализация счетчика вынесена в отдельную функцию `CounterChiselModel.init()`. Принцип ее работы соответствует принципу работы системы сброса устройства: на входной порт необходимо подать логическую единицу и дождаться следующего такта, для большей надежности обычно ожидают больше одного такта.

Так как модель используется в одном потоке, который и создает арену для аллокации сегмента памяти хранящего состояние модели, создается арена с ограниченной областью использования, т.е. с помощью фабричного метода `Arena::ofConfinded`.

Основная функция выполняет следующие действия:

1. Создает арену;
2. Создает объект модели счетчика, используя арену;
3. Инициализирует счетчик;
4. Ожидает 10 циклов тактового сигнала;
5. Считывает результат со счетчика;
6. Успешно проверяет результат на равенство ожидаемому значению 10.

### 3.3.3 Пример с Controls.kt

В качестве более сложного примера использования модели представляется счетчик в составе прибора на базе фреймворка Controls.kt с графическим интерфейсом реализованным с использованием фреймворка Compose Multiplatform. Основные элементы модели, реализующей эту систему можно видеть на листинге в приложении Б. Этот проект представляет собой модель системы, которая схематично изображена на рисунке 5.

Пример демонстрирует способ интеграции модели прибора из фреймворка Controls.kt с моделью прибора созданного с помощью Circulator. В примере создается класс, который реализует интерфейс прибора и определяет спецификацию работы с ним. Обращения к свойствам и методам модели прибора из Controls.kt делегируются модели прибора из Circulator. Легко видеть сходство этого примера с ранее продемонстрированным простым примером.

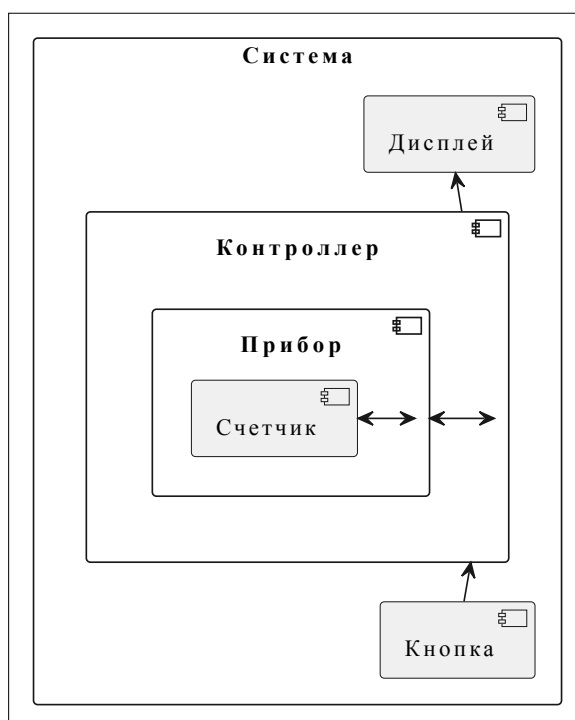


Рисунок 5 — Схема зависимости и размещения компонентов для примера системы с использованием фреймворка Controls.kt

По сути, приложение использует прибор, который еще не произведен и существует лишь в виде дизайна на языке описания аппаратуры, но уже в таком виде его можно использовать для построения системы и поиска ошибок в ней. Этот пример, несмотря на его примитивность, может служить демонстрацией выполнения сквозного тестирования системы от нажатия на кнопку до отображения счета на экране.

Модель счетчика создается в контроллере посредством менеджера устройств `DeviceManager` из Controls.kt, который подключает все устройства в одну систему. Далее контроллер используется при работе панели управления.

Функция `CounterDeviceController.Panel()` отрисовывает графический интерфейс приложения, реализующего возможность автоматического и ручного увеличения счета, а также сброса. Внешний вид этого приложения можно видеть на рисунке 6.

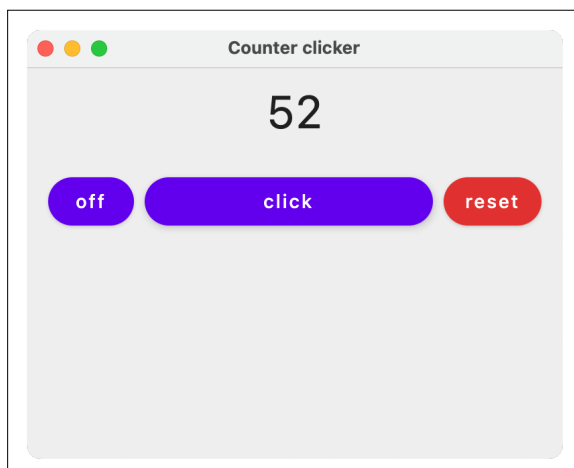


Рисунок 6 — Внешний вид графического приложения представляющего систему

Во время использования этого приложения было выяснено, что в системе возникает проблема представления данных: счетчик хранит беззнаковое представление числа, а экран ожидает для отображения число в знаковом представлении, из-за чего при достижении значения счета 128 происходит знаковое переполнение и отображается  $-128$ , что видно на рисунке 7. При этом счетчик напрямую с экраном не взаимодействует, потому что за передачу данных между компонентами отвечает контроллер.

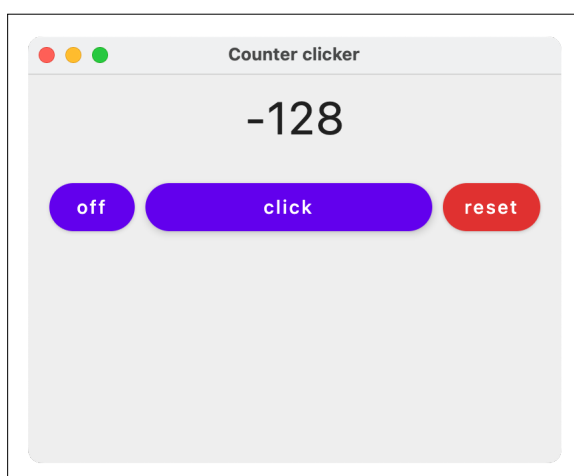


Рисунок 7 — Внешний вид приложения: возникло знаковое переполнение в системе

Более формально результат тестирования выглядит так:

– **Действие:** нажать кнопку «click» 128 раз.

- **Ожидаемый результат:** значение счетчика 128.
- **Фактический результат:** значение счетчика  $-128$ .
- **Проблема:** неверное значение счетчика.
- **Причина:** знаковое переполнение при отображении значения счетчика.

Простой путь исправления этого недостатка системы заключается в изменении интерфейса управления экраном: тип входных данных должен допускать полный диапазон значений счетчика. После использования типа `Int` для входного значения экрана система работает предсказуемо, и позволяет вести подсчет вплоть до значения 255 без переполнения, что можно видеть на рисунке 8. Таким образом, сквозное тестирование с применением моделей `Circulator` позволяет повышать качество разрабатываемых систем, за счет раннего обнаружения ошибок.

Среди важных аспектов работы моделируемой системы можно отметить, что арена, используемая для выделения сегмента памяти внутри модели, получается из фабричного метода `Arena::ofShared` по той причине, что оба фреймворка, используемых в примере, работают в многопоточном асинхронном режиме и обращения к модели устройства происходят из различных потоков. Фиксация потока, о которой говорится в разделе об условиях использования моделей `Circulator`, для простоты примера не производилась.

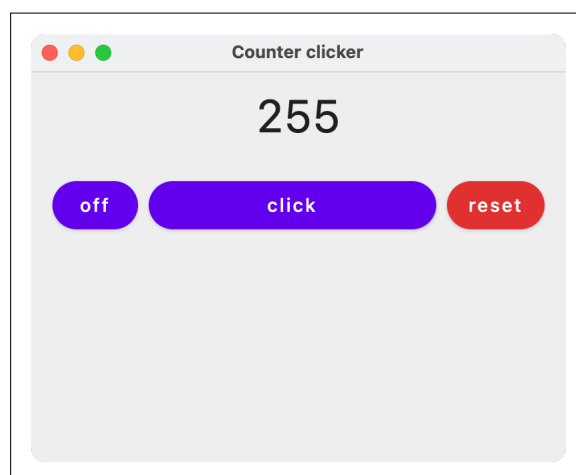


Рисунок 8 — Внешний вид приложения: знаковое переполнение в системе устранено

## 4 Анализ результатов

В данной главе производится анализ результатов выполнения выпускной квалификационной работы, разработанное решение сравнивается с другими готовыми решениями и подходами по достижению поставленной цели, предлагаются варианты использования инструмента Circulator, и рассматриваются перспективы его развития в будущих версиях.

### 4.1 Сравнение с альтернативными решениями

Сравнение разработанного решения Circulator с имеющимися на рынке программными решениями для моделирования аппаратуры по нескольким ключевым критериям:

- доступность,
- функциональность,
- расширяемость,

представлено в виде сводной таблицы 3.

В первую очередь, Circulator является свободным программным обеспечением, распространяемым под лицензией Apache 2.0.

Функциональность Circulator в актуальной версии 0.1.0 на порядок меньше, чем у других программных решений, в том числе коммерческих, и отсутствие пользовательского интерфейса, кроме программного кода, может вызывать трудности в его использовании у людей не знакомых с программированием на языке Kotlin и платформой JVM. Но разработанное решение отвечает заявленным для него требованиям и целевой аудитории пользователей.

В силу своей открытости, Circulator обладает возможностью «бесконечного расширения» функциональности, с помощью доработки программного кода, что не свойственно закрытым проприетарным продуктам, хотя некоторые из них предоставляют программный интерфейс на одном из языков общего назначения: Python, Julia, Lua и другие.

Проект Circulator не ограничивается конкретным набором совместимых технологий, вместо этого он предоставляет API для взаимодействия и полагается на промежуточное представление. Библиотека совместима с другими



проектами, работающими на платформе JVM и не обязательно использующими язык Kotlin.

Таблица 3 — Сравнение разработанного решения Circulator с другими решениями

Критерий	Circulator	Другие инструменты
Доступность	Бесплатный и с открытым исходным кодом	В основном проприетарные с дорогой лицензией
Функциональность	Сфокусирован на интеграции и генерации кода для JVM	Широкие возможности моделирования и симуляции
Расширяемость	Высокая расширяемость благодаря открытости кода	Ограниченная расширяемость, часто привязана к экосистемам поставщиков
Целевая аудитория	Разработчики, использующие Kotlin/JVM	Инженеры и исследователи в области проектирования аппаратуры
Интеграция	Бесшовная интеграция с системами на основе JVM	Автономные инструменты с ограниченными возможностями интеграции
Кривая обучения	Требует знаний Kotlin и JVM	Различается, часто требует специализированных знаний
Гибкость	Поддерживает несколько языков описания аппаратуры через промежуточное представление	Часто привязаны к конкретным языкам или фреймворкам

Разработанное решение не стремится заменить имеющиеся решения для создания и использования моделей аппаратуры, оно занимает позицию между ними и системами полагающимися на разработанные модели, т.е. Circulator выступает в роли интегратора готовых дизайнов или моделей аппаратуры в управляющую систему по типу Controls.kt — подобного рода «бесконечно расширяемую» систему. Инструмент нацелен на использование при сквозном тестировании не аппаратных модулей, а систем, использующих эти модули, что демонстрируется на примере в разделе 3.3.3.

## 4.2 Способы применения

Применение Circulator по целевому назначению предполагает совместное использование генерируемых моделей с виртуальными устройствами фреймворка Controls.kt при выполнении тестовых сценариев. Наличие API на языке Kotlin позволяет использовать популярные фреймворки для тестирования и современные подходы такие, как «fuzzing» и «Property-based» тестирование.

Кроме того, можно предложить другой сценарий использования разработанного инструмента — верификация готовой модели устройства с помощью сгенерированной Circulator модели. В таком виде получается производить модульное тестирование модели устройства.

Среди альтернативных направлений использования Circulator можно увидеть

- разработку учебных демонстрационных моделей устройств,
- модульное и функциональное тестирование дизайна аппаратного модуля рядовыми инженерами по автоматизированному тестированию программного обеспечения
- и разработка тестовых сценариев, называемых «testbench», в проектах на языке Chisel, с последующим их исполнением.

## 4.3 Перспективы развития

Реализованная версия 0.1.0 проекта Circulator обладает достаточной функциональностью для соответствия поставленным ей требованиям, тем не менее можно рассмотреть возможности для дальнейшего развития проекта.

Пути развития функциональности проекта связаны с расширением набора технологий доступных для использования, который сильно ограничен в данной версии, об этом говорилось в разделе 2.1 про требования к проекту. А именно, потенциал развития имеют следующие аспекты.

- Поддержка средств создания нативных моделей и симуляции, отличных от Arcilator. Например, Verilator и SystemC, которые не полагаются на промежуточное представление.

- Поддержка других способов взаимодействия с нативными библиотеками. Например, JNI или возможности Kotlin/Native.
- Поддержка платформ для запуска моделей, отличных от JVM. Примерами таких платформ могут быть Native и WASM.

Также можно рассмотреть варианты добавления в плагин сборочной системы Gradle возможностей по конфигурированию проектов, направленных на разработку дизайна аппаратных модулей, и создание входящего в проект Circulator доменно специфичного языка для описания аппаратных модулей на базе языка Kotlin.

## ЗАКЛЮЧЕНИЕ

В данной работе была спроектирована архитектура и разработан программный модуль предоставляющий возможности моделирования аппаратуры на уровне цифровых схем, который применим для использования с виртуальными устройствами фреймворка Controls.kt.

Разработанный программный модуль Circulator позволяет автоматически генерировать модели на языке Kotlin из имеющихся моделей устройств, разработанных на языке описания аппаратуры. .

Также в работе был продемонстрирован пример использования инструмента для системы на базе фреймворка Controls.kt, на котором показано, что **цель работы достигнута и поставленные задачи выполнены**: с использованием разработанного решения можно проводить сквозное тестирование системы для обнаружения недостатков, исправление которых ведет к повышению качества системы.

Разработанное решение планируется начать использовать в новых проектах разрабатывающих системы на базе фреймворка Controls.kt. Код проекта размещен в личном репозитории, ссылка на который приведена в приложении В. Развитие проекта Circulator планируется продолжить.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Chris Lattner. CIRCT: Lifting hardware development out of the 20th century / Andrew Lenharth and Chris Lattner. – SIFIVE, 2021. – URL: <https://llvm.org/devmtg/2021-11/slides/2021-CIRCT-LiftingHardwareDevOutOfThe20thCentury.pdf> (дата обращения: 19.05.2025). – Текст : электронный.
2. Controls-kt. – URL: <https://sciprogramcenter.org/projects/controls> (дата обращения: 19.05.2025). – Текст : электронный.
3. Implementing Binary Plugins. – URL: [https://docs.gradle.org/current/userguide/implementing\\_gradle\\_plugins\\_binary.html](https://docs.gradle.org/current/userguide/implementing_gradle_plugins_binary.html) (дата обращения: 06.04.2025). – Текст : электронный.
4. KotlinPoet. – URL: <https://square.github.io/kotlinpoet/> (дата обращения: 06.04.2025). – Текст : электронный.
5. kscience/controls-kt: Lightweight SCADA device servers and integrations in Kotlin-Multiplatform - controls-kt - SPC-git.
6. Martin Erhart. Arcilator. Fast and cycle-accurate hardware simulation in CIRCT / Martin Erhart, Fabian Schuiki, Zachary Yedidia, [и др.]. – URL: <https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf> (дата обращения: 24.03.2025). – Текст : электронный.
7. Martin Schoeberl. Digital Design with Chisel / Martin Schoeberl – 6 – Martin Schoeberl, – URL: <https://www.imm.dtu.dk/~masca/chisel-book.pdf> (дата обращения: 25.03.2025). – Текст : электронный – ISBN 9781689336031.
8. Mazalova, V. A Novel Solution for Controlling Hardware Components of Accelerators and Beamlines.
9. Nozik, Alexander. DataForge: Modular platform for data storage and analysis / Nozik, Alexander // EPJ Web Conf.. – 2018. – Т. 177. – С. 5003.
10. Nozik, A. Controls-kt, a Next Generation Control System / A. Nozik. – 2023.
11. Nozik, A. Declarative analysis in “Troitsk nu-mass” experiment / A. Nozik // Journal of Physics: Conference Series. – 2020. – Т. 1525. – № 1. – С. 12024.

12. Project Panama: Interconnecting JVM and native code. – URL: <https://openjdk.org/projects/panama/> (дата обращения: 25.03.2025). – Текст : электронный.
13. The basics of Kotlin Multiplatform project structure | Kotlin Multiplatform Development Documentation. – URL: <https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-discover-project.html#targets> (дата обращения: 29.03.2025). – Текст : электронный.
14. Théo Degioanni. Arcilator for ages five and up. Flexible self-contained hardware simulation made easy / Théo Degioanni. – 2024. – URL: <https://llvm.org/devmtg/2024-04/slides/QuickTalks/Degioanni-Arcilator.pdf> (дата обращения: 24.03.2025). – Текст : электронный.
15. Wang, F. Verik: Reinterpreting Kotlin as a Hardware Description Language / F. Wang. – Massachusetts Institute of Technology, 2022.
16. Антонов А. А. Цифровой синтез: практический курс / Антонов А. А., Барабанов А. В., Данчек Ч. Т., [и др.] – Москва : ДМК Пресс, 2020 – 556 с. – ISBN 978-5-97060-850-0.
17. В.А. АЛЕХИН. Project Panama: Interconnecting JVM and native code / В.А. АЛЕХИН – Москва : МИРЭА, 2017 – 396 с. – URL: <http://www.toe-mirea.ru/download/file47.pdf> (дата обращения: 13.04.2025). – Текст : электронный.
18. Коузен, К. Kotlin. Сборник рецептов. Предметный подход = Kotlin cookbook / К. Коузен – Москва : ДМК Пресс, 2021 – 218 с. – ISBN 978-5-97060-883-8.

## ПРИЛОЖЕНИЕ А

```
[
  {
    "name": "Counter",
    "numStateBytes": 8,
    "initialFnSym": "",
    "finalFnSym": "",
    "states": [
      {
        "name": "clk",
        "offset": 0,
        "numBits": 1,
        "type": "input"
      },
      {
        "name": "reset",
        "offset": 1,
        "numBits": 1,
        "type": "input"
      },
      {
        "name": "clk",
        "offset": 2,
        "numBits": 1,
        "type": "wire"
      },
      {
        "name": "reset",
        "offset": 3,
        "numBits": 1,
        "type": "wire"
      },
      {
        "name": "reg",
        "offset": 5,
        "numBits": 8,
```

```

        "type": "register"
    },
    {
        "name": "o",
        "offset": 6,
        "numBits": 8,
        "type": "wire"
    },
    {
        "name": "o",
        "offset": 7,
        "numBits": 8,
        "type": "output"
    }
]
}
]

```



## ПРИЛОЖЕНИЕ Б

```
interface ICounterDevice : Device {
    fun reset()
    fun click()
    val countValue: Int
}

class CounterDevice(context: Context, meta: Meta, arena: Arena) :
    DeviceBySpec<ICounterDevice>(Spec, context, meta),
    ICounterDevice {

    val model: CounterChiselModel =
        CounterChiselModel.instance(arena, "counterchisel")

    override fun click() {
        model.clock = 1
        model.eval()
        model.clock = 0
        model.eval()
    }

    override fun reset() {
        model.reset = 1
        model.click()
        model.reset = 0
    }

    override val countValue: Int get() = model.count.toInt()

    companion object Spec : DeviceSpec<ICounterDevice>(),
        FfmMetaFactory<CounterDevice> {
        val count by numberProperty { countValue }
        val reset by unitAction { reset() }
        val click by unitAction { click() }

        override fun factory(arena: Arena) = Factory
```

```

{ context, meta ->
    CounterDevice(context, meta, arena)
}
}

interface FfmMetaFactory<T> {
    fun factory(arena: Arena): Factory<T>
}

class CounterDeviceController(arena: Arena): ContextAware {
    override val context = Context(name = "Demo") {
        plugin(DeviceManager)
    }
    val deviceManager = context.request(DeviceManager)
    val counter by
deviceManager.installing(CounterDevice.factory(arena))
}

fun main() {
    Arena.ofShared().use { arena ->
        val controller = CounterDeviceController(arena)

        application {
            ControlPanel(controller)
        }
    }
}

@Composable
fun ControlPanel(controller: CounterDeviceController) {
    val controller = remember { controller }

    Window(/* ... */) {
        controller.Panel()
    }
}

```

```
@Composable
```

```
fun Screen(output: Byte) {  
    Text(output.toString())  
}
```

```
@Composable
```

```
fun CounterDeviceController.Panel() {  
    var count by remember { mutableStateOf(counter.countValue) }  
    var auto by remember { mutableStateOf(false) }
```

```
    LaunchedEffect(auto) {  
        while (auto) {  
            counter.click()  
            count = counter.countValue  
            delay(1000)  
        }  
    }  
}
```

```
Column {  
    Box { Screen(count.toByte()) }  
    Row {  
        Button(onClick = { auto = !auto }) {  
            Text(if (auto) "1s" else "off")  
        }  
        Button(onClick = {  
            counter.click()  
            count = counter.countValue  
        }) {  
            Text("click")  
        }  
        Button(onClick = {  
            counter.reset()  
            count = counter.countValue  
        }) {  
            Text("reset")  
        }  
    }  
}
```

}  
}  
}

## **ПРИЛОЖЕНИЕ В**

<https://github.com/elturin/circulator-kt>