Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Системы обработки информации и управления» (ИУ5)

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

«Разработка методов структурного анализа и адаптивного синтеза систем Backend-Driven UI на основе расширенной метаграфовой модели»

Студент группы ИУ5-32М _____ А. С. Ищенко

Руководитель _____ Ю. Е. Гапанюк

2025 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ5

_____ В.И. Терехов

« ____ » _____ 2025 г.

ЗАДАНИЕ
на выполнение научно-исследовательской работы

по теме: Разработка методов структурного анализа и адаптивного синтеза систем Backend-Driven UI на основе расширенной метаграфовой модели

Студент группы ИУ5-22М Ищенко Анастасия Сергеевна
(Фамилия имя отчество)

Направленность НИР (учебная, исследовательская, практическая, производственная, др.)

исследовательская

Источник тематики (кафедра, предприятие, НИР) учебная тематика

График выполнения НИР: 25% к 5 нед., 50% к 9 нед., 75% к 13 нед., 100% к 16 нед.

***Техническое задание*:** Исследование методологии проектирования систем Backend-Driven UI. Разработка расширенной метаграфовой модели для структурного анализа сложности, автоматической верификации и адаптивного синтеза динамических интерфейсов.

Оформление научно-исследовательской работы:
Расчетно-пояснительная записка, 28 листов формата А4.

Приложения: нет

Дата выдачи задания «15» сентября 2025 г.

Научный руководитель _____ Ю.Е. Гапанюк
(подпись, дата) (инициалы и фамилия)

Студент группы ИУ5-32М _____ А.С. Ищенко
(код группы) (подпись, дата) (инициалы и фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ

Актуальность темы исследования. В современной индустрии разработки мобильного программного обеспечения наблюдается устойчивая тенденция к переходу от монолитных клиентских архитектур к распределенным системам управления интерфейсом. Парадигма Backend-Driven UI (BDUI), при которой структура, логика и визуальное представление пользовательского интерфейса формируются динамически на сервере, стала стандартом де-факто для крупных экосистемных приложений (E-commerce, Banking, Superapps). Данный подход позволяет радикально сократить время вывода функциональности на рынок (Time-to-Market) и обеспечивает беспрецедентную гибкость персонализации.

Однако переход к генеративным интерфейсам породил класс новых, малоизученных инженерных проблем. Распределенная природа BDUI приводит к тому, что состояние клиентского приложения становится нетерминированным, зависящим от множества контекстных факторов и версий API. Отсутствие строгой типизации на этапе компиляции и экспоненциальный рост сложности серверной логики (эффект «комбинаторного взрыва» состояний) делают традиционные методы обеспечения качества неэффективными. Существующие средства моделирования (UML, классические графы, конечные автоматы) не обладают достаточной выразительной мощностью для описания иерархических, эмерджентных структур, характерных для современных UI-фреймворков (таких как DivKit, Jetpack Compose, SwiftUI).

В связи с этим, развитие нового математического аппарата, способного формализовать структуру и поведение динамических интерфейсов, а также создание на его основе методов количественной оценки и верификации, является актуальной научной задачей.

Объектом исследования являются процессы проектирования и функционирования динамических пользовательских интерфейсов, построенных на архитектуре Backend-Driven UI.

Предметом исследования выступают модели и методы структурного анализа, верификации и синтеза интерфейсов на основе теории сложных сетей и метаграфов.

Цель работы заключается в разработке формальной методологии анализа и синтеза динамических интерфейсов на базе расширенной метаграфовой модели для повышения их архитектурной надежности, предсказуемости и снижения структурной сложности.

Для достижения цели в работе решены следующие **задачи**:

1. **Системный анализ предметной области:** Провести классификацию архитектурных паттернов мобильной разработки, выявить ключевые проблемы надежности и сложности подхода BDUI, а также обосновать недостаточность существующих методов моделирования.
2. **Разработка математической модели:** Разработать и формализовать расширенную модель активного аннотируемого метаграфа, позволяющую описывать иерархическую структуру, динамику изменений и агентное поведение компонентов пользовательского интерфейса.
3. **Создание методологии анализа и синтеза:** Разработать систему метрик для количественной оценки структурной и когнитивной сложности интерфейсов, а также предложить алгоритмы автоматической верификации сценариев и адаптивного синтеза UI.

Научная новизна исследования:

1. Впервые предложена интерпретация JSON-контрактов динамических интерфейсов в терминах теории активных метаграфов, объединяющая структурное (дерево) и поведенческое (сеть) представления.
2. Разработан комплекс формальных метрик (Инвариант Когнитивной Сложности, Индекс Глубины Эмерджентности, Вес Экрана), позволяющих выполнять *априорную* оценку производительности интерфейса на стороне сервера.
3. Предложен метод верификации целостности пользовательских сценариев на графе метасостояний, позволяющий выявлять структурные аномалии (тупики, разрывы связей) без запуска клиентского кода.

Практическая значимость работы заключается в создании теоретического базиса для разработки инструментов автоматизированного анализа (линтеров) и систем проектирования UI. Применение разработанной методологии позволяет снизить риски возникновения ошибок в продакшн-среде («битые» экраны, падения приложений), оптимизировать сетевой трафик за счет контроля веса экранов и обеспечить корректную работу приложений на устаревших версиях клиентов.

2 Анализ проблематики проектирования динамических интерфейсов

2.1 Эволюция архитектурных паттернов: смещение центра тяжести

Чтобы понять природу сложности современных UI-систем, необходимо проследить эволюцию распределения ответственности между клиентом и сервером.

2.2.1. Классические нативные паттерны (MVC, MVP, MVVM, VIPER) В традиционных архитектурах сервер выступает исключительно как поставщик данных (Data Provider). REST API или GraphQL возвращают «сырые» сущности (Product, User, Cart), а клиентское приложение берет на себя все функции по их интерпретации и визуализации.

- *Преимущества:* Высокая производительность, строгая проверка типов на этапе компиляции, работа офлайн.
- *Недостатки:* Нулевая гибкость UI без обновления приложения. Логика отображения дублируется на iOS и Android, что ведет к рассинхронизации платформ.

2.2.2. Гибридные подходы и WebViews Попытка решения проблемы гибкости через встраивание веб-страниц (WebView) внутрь нативного приложения.

- *Преимущества:* Мгновенное обновление.
- *Недостатки:* Низкая производительность, эффект «зловещей долины» в UX (интерфейс не ощущается нативным), ограниченный доступ к аппаратным функциям устройства.

2.2.3. Парадигма Server-Driven UI (SDUI) Качественный скачок, при котором сервер передает не *данные*, а *инструкции по рендерингу*. Клиентское приложение вырождается в универсальный «браузер виджетов» (Rendering Engine). Оно не знает заранее, как будет выглядеть экран главной страницы; оно лишь умеет отрисовывать примитивы (текст, картинка, контейнер) по полученному JSON-контракту.

- *Особенность:* Центр принятия решений о том, *что* и *как* показывать, смещается на бэкенд. Это превращает задачу проектирования UI из

задачи верстки в задачу проектирования распределенных систем обмена сообщениями.

2.2 Архитектурно-структурный анализ систем Backend-Driven UI

На основе анализа технических решений ведущих технологических компаний (в частности, фреймворков DivKit от Яндекс и решений Ozon Tech), можно выделить обобщенную архитектуру SDUI-системы. Она представляет собой сложную гетерогенную структуру, состоящую из следующих компонентов:

1. **Layout Management Tool (LMT):** Админка-конструктор, где менеджеры или дизайнеры собирают структуру экранов из доступных блоков. Это точка входа изменений.
2. **BFF (Backend for Frontend) / API Gateway:** Слой агрегации, который принимает запрос от мобильного устройства, определяет контекст пользователя (ID, версия приложения, тема оформления, геолокация) и запрашивает данные у множества микросервисов.
3. **Сервис генерации Layout:** Ключевой компонент, который «сплавляет» данные и структуру. На выходе формируется иерархическое дерево (обычно JSON), описывающее UI.
4. **Тонкий Клиент (Renderer):** Библиотека на мобильном устройстве, которая рекурсивно обходит полученное дерево и инстанцирует нативные компоненты (например, RecyclerView в Android или UICollectionView в iOS).

Проблема нетерминированности состояния. В отличие от классического приложения, где граф переходов между экранами (Navigation Graph) статичен и детерминирован, в SDUI граф переходов формируется динамически. Клиент может получить инструкцию «Перейти на экран X», но структура экрана X будет известна только в момент перехода. Это переводит систему в класс систем с неполной информацией на стороне клиента, что делает невозможным применение статических анализаторов кода.

2.3 Типология проблем и «камней преткновения» в реализации BDUI

Внедрение SDUI, несмотря на маркетинговую привлекательность, сопровождается рядом серьезных технических вызовов, которые в литературе часто называют «Stumbling Blocks» (камни преткновения). Их анализ критически важен для постановки задачи исследования.

2.3.1. Проблема контрактного версионирования (Versioning Hell) Сервер всегда имеет тенденцию к развитию (появление новых виджетов), а парк клиентских устройств обновляется неравномерно. *Сценарий отказа:* Сервер отправляет JSON с новым типом виджета "type": "SuperStory". Старый клиент (версия N-1), не имея реализации для этого типа, либо падает (Crash), либо отображает пустое место, либо ломает всю верстку экрана. *Архитектурный вывод:* Необходима система согласования возможностей (capabilities negotiation) между клиентом и сервером, что усложняет протокол взаимодействия.

2.3.2. Эмерджентные ошибки верстки В BDUI структура экрана формируется динамически. Сочетание виджетов, которое никогда не тестировалось вместе, может привести к визуальным конфликтам. Например, виджет А требует горизонтальной прокрутки, а виджет Б, в который вложен А, перехватывает жесты прокрутки. В статическом приложении это выявляется на этапе QA. В динамическом — такая комбинация может возникнуть у 1% пользователей при специфических условиях, делая ошибку трудновоспроизводимой.

2.3.3. Разрыв семантической связности (Semantic Gap) В статической верстке связь между кнопкой «Купить» и полем ввода «Количество» очевидна и прописана в коде контроллера. В SDUI эти элементы приходят как независимые узлы JSON-дерева. Чтобы связать их логикой (валидация поля перед нажатием), требуются сложные механизмы «Actions» и «Variables» (как в DivKit), которые по сути являются тьюринг-полным языком программирования внутри JSON. Это резко повышает когнитивную сложность для разработчика: программирование переносится в конфигурационные файлы.

2.4 Проблема когнитивной нагрузки и «веса» экрана (Screen Weight)

Отдельным аспектом, требующим научного анализа, является производительность и потребление ресурсов. В нативном приложении верстка оптимизирована и скомпилирована. В SDUI происходит процесс:

1. Сериализация огромного JSON на сервере (нагрузка на CPU сервера).
2. Передача по сети (Traffic overhead — JSON содержит много метаданных о стилях).
3. Парсинг и рекурсивное построение View-дерева на клиенте (нагрузка на Main Thread мобильного устройства).

Существует прямая корреляция между глубиной вложенности JSON-структуры и падением FPS (кадров в секунду) при скролле. В индустрии эмпирически выявлено, что глубина вложенности более 8-10 уровней вызывает заметные лаги. Однако на текущий момент отсутствуют формальные метрики, позволяющие на этапе проектирования (на сервере) оценить «тяжесть» (Screen Weight) сгенерированного экрана для клиента. Введение таких метрик требует математической модели, учитывающей структуру вложенности.

2.5 Критический анализ существующих формальных языков описания систем

Для решения озвученных выше проблем необходим инструмент моделирования. Рассмотрим применимость классических подходов.

2.5.1 UML (Unified Modeling Language)

- *Диаграммы классов:* Описывают статическую структуру кода (классы Widget, Container). Бесполезны для описания *динамической структуры конкретного экрана*, приходящего с сервера.
- *Диаграммы последовательности:* Показывают обмен сообщениями (Client <-> Server), но не раскрывают *содержание* сообщений (структуру UI).
- *Вывод:* UML ориентирован на структуру кода, а не на структуру данных, которая в SDUI является определяющей.

2.5.2. Конечные автоматы (FSM) и Statecharts Подходят для моделирования жизненного цикла одного виджета (Loading -> Error -> Data).

Однако попытка описать весь экран как FSM приводит к «взрыву состояний» (state explosion problem). Если экран состоит из 20 независимых виджетов, количество состояний экрана стремится к произведению состояний всех виджетов ($S_{total} = \prod S_i$), что делает модель необозримой.

2.5.3. Сети Петри Позволяют моделировать параллелизм и синхронизацию. Могут быть полезны для описания асинхронной загрузки данных разными виджетами. Однако классические сети Петри являются «плоскими» структурами и плохо подходят для описания глубокой иерархической вложенности (View Hierarchy), характерной для UI.

2.5.4. Простые и ориентированные графы Традиционная графовая модель $G = (V, E)$способна описать структуру связей. Вершины — виджеты, ребра — отношения «родитель-потомок». Однако:

1. Обычный граф не имеет встроенного понятия «контейнеризации» или группировки подграфа в единую сущность (метавершину), что критично для компонентной архитектуры.
2. Обычный граф описывает только бинарные отношения. В UI часто встречаются отношения «один ко многим» (один Action обновляет состояние трех виджетов), что требует использования гиперграфов или более сложных структур.

2.6 Обоснование выбора метаграфового подхода

На основании проведенного анализа можно сформулировать требования к идеальной модели для анализа Backend-Driven UI:

1. **Иерархичность:** Модель должна естественно описывать вложенность (виджет внутри контейнера, контейнер внутри экрана).
2. **Эмерджентность:** Модель должна учитывать, что группа виджетов (компонент) обладает свойствами, отсутствующими у отдельных частей (например, padding контейнера влияет на всех детей).
3. **Гетерогенность связей:** Необходимость различать связи типа «Вложенность» (структура) и связи типа «Действие» (динамика).
4. **Агентность:** Возможность моделировать компоненты, обладающие собственным поведением (таймеры, анимации).

Метаграфовая модель, предложенная в работах А. Базу, Р. Блэннинга и развитая в трудах Ю.Е. Гапанюка, удовлетворяет всем перечисленным требованиям.

- **Метавершина** естественным образом моделирует UI-контейнер и экран целиком.
- **Метаребро** (связывающее множество вершин с множеством вершин) идеально описывает сложные Actions (например, «нажатие кнопки» -> «валидация полей А и Б» + «отправка запроса» + «показ лоадера»).
- **Атрибутивность** метаграфа позволяет хранить JSON-контракты и стили непосредственно в узлах модели.

Таким образом, переход от качественного описания проблем BDUI к их количественному анализу и автоматизированной верификации возможен через проекцию предметной области на формализм **аннотируемых активных метаграфов**.

2.7 Постановка задачи исследования

Исходя из выявленной проблематики, основной целью исследования становится адаптация общего метаграфового подхода для специфической задачи анализа динамических интерфейсов. Это требует решения следующих подзадач:

1. Formal mapping (отображение) понятий BDUI (Server Response, Widget, Action, DivKit Variables) на элементы метаграфа.
2. Разработка алгоритмов поиска структурных аномалий (тупиков, циклов, разрывов связей) на графе.
3. Введение метрик (Инвариант сложности), позволяющих численно оценить качество архитектуры до написания кода.

Решение этих задач позволит создать инструментарий для *a priori* анализа надежности мобильных приложений, синтезируемых на сервере, что является актуальной задачей для современной программной инженерии.

3 Разработка методов формализации и структурного описания Backend-Driven Ui на основе расширенной метаграфовой модели

3.1 Базовые положения и теоретико-множественное описание метаграфа интерфейса

В качестве фундаментальной основы используется определение метаграфа, данное в работах А. Базу и Р. Блэннинга, с модификациями Ю.Е. Гапанюка, адаптированными под предметную область UI-проектирования.

Определим **Метаграф пользовательского интерфейса** $\mathcal{M}_{UI}$ как упорядоченный кортеж: $\mathcal{M}_{UI} = \langle V, MV, E, ME, \mathcal{A}, \Omega \rangle$

Рассмотрим компоненты кортежа детально:

1. Множество вершин (Vertices) V: $V = \{v_1, v_2, \ldots, v_n\}$ — конечное множество атомарных вершин. В контексте BDUI вершина v_i соответствует примитивному UI-компоненту (виджету), который не является контейнером для других элементов с точки зрения логики системы, либо его внутренней структурой на данном уровне абстракции можно пренебречь. *Примеры* v_i*:* Текстовая метка (Text), Иконка (Image), Атомарный разделитель (Divider).

2. Множество метавершин (Metavertices) MV: $MV = \{mv_1, mv_2, \ldots, mv_m\}$ — конечное множество метавершин. Метавершина mv_j является ключевым элементом, обеспечивающим свойство иерархичности. Она представляет собой композитный компонент (Контейнер), который включает в себя подмножество вершин и других метавершин. Формально метавершина определяется как: $mv_j = \langle V_j, MV_j, E_j, \mathcal{A}_j \rangle$ где $V_j \subset V$, $MV_j \subset MV$ — содержимое контейнера (children). *Примеры* mv_j*:* Вертикальный стек (VStack), Скроллируемый список (RecyclerView), Карточка товара (ProductCard), Экран целиком (Screen).

Свойство эмерджентности: Важно отметить, что $mv_j \neq V_j \cup MV_j$. Метавершина обладает собственными атрибутами $\mathcal{A}_j$ (например, padding, background_color, layout_strategy), которые определяют правила расположения вложенных элементов. Удаление метавершины влечет за собой удаление или переструктурирование всего содержимого, что соответствует холонической природе UI.

3. Множество ребер (Edges) E**:** $E = \{e_k\}$ — множество классических ребер графа. В модели BDUI они используются для задания *пространственных* и *логических* отношений внутри одной метавершины, не являющихся действиями. Например, направленное ребро $e = (v_a, v_b)$ внутри метавершины типа HorizontalStack задает отношение порядка: «Элемент v_a рендерится перед элементом v_b».

4. Множество метаребер (Meta-edges) ME**:** $ME = \{me_p\}$ — множество метаребер. Это наиболее мощный инструмент модели. Метаребро связывает произвольные подмножества вершин и метавершин. $me_p = \langle X_p, Y_p, \mathrm{dir}_p \rangle$ где $X_p \subseteq (V \cup MV)$ — входящее множество (источники), $Y_p \subseteq (V \cup MV)$ — исходящее множество (приемники), dir_p — направление. В контексте UI метаребра моделируют **Действия (Actions)** и сложные зависимости. *Пример:* Нажатие на кнопку «Очистить» ($v_{btn} \in X_p$) приводит к изменению текста в трех полях ввода ($v_{inp1}, v_{inp2}, v_{inp3} \in Y_p$). Классический граф потребовал бы трех отдельных ребер, тогда как метаребро описывает это как единую транзакцию.

3.2 Аннотирование модели: Проекция JSON-контрактов на атрибуты

Backend-Driven UI базируется на строгих контрактах данных (обычно в формате JSON или Protobuf). Для отражения этого факта модель должна быть **аннотируемой**.

Введем множество атрибутов $\mathcal{A}$ и функции аннотирования. Для каждого элемента графа $x \in (V \cup MV \cup E \cup ME)$ определен вектор атрибутов: $\mathrm{attr}(x) = \{(k_1, val_1), (k_2, val_2), \dots\}$ где k_i — ключ атрибута (соответствует полю в JSON), val_i — значение.

Классификация атрибутов в модели:

1. **Визуальные атрибуты (**$\mathcal{A}_{vis}$**):** Цвет, шрифт, размер. Они статичны для конкретного состояния.
2. **Данные (**$\mathcal{A}_{data}$**):** Текст заголовка, URL картинки, цена товара. Эти атрибуты подставляются сервером динамически.
3. **Поведенческие атрибуты (**$\mathcal{A}_{behav}$**):** Ссылки на обработчики событий (DeepLinks), идентификаторы действий.

4. **Семантические атрибуты ($\mathcal{A}_{sem}$):** Метаданные для аналитики и доступности (Accessibility hints).

Такое разделение позволяет при анализе сложности (см. Глава 3) назначать разные веса разным типам атрибутов. Изменение визуального атрибута «дешево» для системы рендеринга, тогда как изменение структурных атрибутов требует перерасчета Layout.

3.3 Моделирование гибридной структуры: Дерево и Сеть

Особенностью предлагаемого подхода является дуализм представления. Модель $\mathcal{M}_{UI}$ можно рассматривать в двух проекциях:

Проекция 1: Дерево композиции (Layout Tree) Если рассматривать только отношение вложенности (принадлежность вершин метавершинам), модель вырождается в ориетированный лес (или дерево, если мы рассматриваем один экран). $v_a \in mv_b \implies v_a$ is child of mv_b Это позволяет применять к модели эффективные алгоритмы обхода деревьев (DFS/BFS) для расчета метрик глубины вложенности и общего веса экрана.

Проекция 2: Сеть взаимодействий (Interaction Network) Если рассматривать метаребра ME, модель превращается в сложную сеть, наложенную поверх дерева композиции. Метаребра могут соединять вершины из разных ветвей дерева иерархии. *Пример:* Кнопка «Купить» находится в «Подвале» экрана (mv_{Footer}), а счетчик корзины — в «Шапке» (mv_{Header}). Метаребро связывает их напрямую, игнорируя иерархические границы. Именно наличие этой проекции позволяет обнаруживать скрытые циклические зависимости и гонки состояний, которые невидны при анализе чистого JSON-дерева.

3.4 Концепция Активного Метаграфа и Агентное моделирование

Современные SDUI-фреймворки (например, Yandex DivKit, Google Jetpack Compose) поддерживают не только статический рендеринг, но и клиентскую логику: таймеры, валидаторы, анимации, вычисляемые выражения (variables). Статическая модель не способна описать поведение вида: «Спрятать кнопку, если переменная timer станет равна 0».

Для решения этой задачи вводится расширение — **Активный Метаграф**: $\mathcal{M}_{Active} = \langle \mathcal{M}_{UI}, \mathcal{AG}, \Phi \rangle$

1. Множество Агентов ($\mathcal{AG}$): $\mathcal{AG} = \{ag_1, ag_2, \dots\}$ — множество программных агентов, ассоциированных с узлами графа. Агент ag_i — это сущность, обладающая:

- Областью видимости (Scope): подграф, к которому агент имеет доступ (обычно mv_{parent} и его дети).
- Внутренним состоянием (Local State).
- Набором триггеров (Triggers).

2. Функция привязки (Φ): $\Phi: V \cup MV \to \mathcal{AG}$ — отображение, ставящее в соответствие элементам интерфейса управляющих агентов. Например, метавершина $mv_{TimerWidget}$ связана с агентом ag_{Timer}, который каждую секунду инициирует транзакцию обновления атрибута text.

Введение агентов позволяет моделировать интерфейс как **самоорганизующуюся систему**. Сервер задает начальные условия и правила (скрипты агентов), а конечное состояние системы в момент времени t является результатом работы этих агентов. Это позволяет верифицировать сценарии, которые зависят не от действий пользователя, а от времени или внешних событий (push-уведомлений).

3.5 Формализация динамики: Граф переходов метасостояний

Для анализа корректности пользовательских сценариев необходимо перейти от описания структуры к описанию поведения. Введем понятие **Состояния интерфейса** S_t. S_t — это полная конфигурация метаграфа $\mathcal{M}_{UI}$ в момент времени t, включающая текущие значения всех атрибутов и структуру графа.

Динамика системы описывается как последовательность трансформаций: $S_0 \xrightarrow{\text{Action}_1} S_1 \xrightarrow{\text{Action}_2} \dots \xrightarrow{\text{Action}_n} S_n$

Действие (Action) в терминах метаграфов формализуется как оператор трансформации T: $T: \mathcal{M}_{UI} \to \mathcal{M}'_{UI}$ Оператор T может выполнять следующие атомарные операции:

1. **UPDATE(v, attr, val):** Изменение значения атрибута (например, подсветка ошибки в поле ввода). Топология графа не меняется.
2. **INSERT(mv_parent, mv_new):** Динамическая подгрузка блока (например, при скролле ленты товаров). Топология меняется.
3. **DELETE(mv_target):** Удаление ветви графа.
4. **NAVIGATE(screen_id):** Полная замена текущего метаграфа на новый, загруженный с сервера.

Совокупность всех возможных состояний $\mathbb{S}$ и переходов между ними образует **Граф Метасостояний (State Metagraph)**. Анализ связности этого графа позволяет отвечать на вопросы верификации:

- *Существует ли путь из состояния* $Start$ *в состояние* $OrderSuccess$*?* (Задача достижимости).
- *Существует ли состояние, из которого нет выхода (кроме терминальных)?* (Задача поиска тупиков).

3.6 Методология синтеза: Контекстно-зависимая генерация

Одной из задач работы является не только анализ, но и синтез интерфейса. В парадигме BDUI синтез происходит на сервере в момент запроса. Формально процесс синтеза F_{synth} можно описать как функцию, зависящую от контекста пользователя $\mathbb{C}$: $\mathcal{M}_{UI} = F_{synth}(\mathbb{D}, \mathbb{C}, \mathcal{R})$ где:

- $\mathbb{D}$ — Доменные данные (товары, статьи).
- $\mathbb{C}$ — Контекст запроса (Платформа: iOS/Android, Версия App: 10.2, Тема: Dark, Роль: PremiumUser).
- $\mathcal{R}$ — Набор правил (Layout Rules) и шаблонов (Templates).

Метаграфовая модель позволяет формализовать этот процесс как **суперпозицию метаграфовых шаблонов**. Хранилище шаблонов содержит параметризованные под-метаграфы (Blueprints). Процесс синтеза заключается в инстанцировании этих шаблонов и связывании их метаребрами в единую структуру.

Механизм адаптации (Degradation Logic): Важнейшим аспектом синтеза является адаптивность к версии клиента. Пусть v_{new} — вершина нового

типа. В метаграфе определяется атрибут min_client_version. Функция синтеза проверяет условие: $\text{if } \mathbb{C}.version < v_{new}.min\_version \implies \text{REPLACE}(v_{new}, v_{fallback})$ Где $v_{fallback}$ — альтернативная, более простая вершина (метавершина), определенная в модели как безопасный вариант (например, «Веб-вью» вместо нативного виджета «AR-примерка»). Метаграф позволяет явно моделировать эти связи fallback-типа через специальные типы ребер, гарантируя целостность интерфейса для старых клиентов.

3.7 Темпоральные зависимости и жизненный цикл

В BDUI существенную роль играют временные задержки (Network Latency). Модель должна учитывать, что переход из состояния S_i в S_{i+1} не мгновенен. Введем понятие **Промежуточного состояния (Loading State)**. Метаребро действия me_{load} расщепляется на два этапа:

1. $S_{idle} \to S_{loading}$ (отображение скелетонов/лоадеров).
2. $S_{loading} \to S_{data}$ (отображение данных) ИЛИ $S_{loading} \to S_{error}$ (отображение ошибки).

В метаграфовой модели это реализуется через введение **вероятностных метаребер** для этапа 2. Это позволяет применять методы анализа надежности и оценивать вероятность успешного завершения пользовательского сценария в условиях нестабильной сети.

4 Методология анализа сложности, верификации и адаптивного синтеза BDUI-систем

4.1 Система метрик структурной оценки интерфейса

Для количественной оценки качества архитектуры UI вводится вектор метрик $\vec{Q} = (NDI, SW, GDI)$, вычисляемый на статическом срезе метаграфа.

4.1.1. Индекс Глубины Эмерджентности (Nesting Depth Index — NDI) В нативной разработке (особенно в Android) критическим фактором производительности является глубина иерархии View. Каждый уровень вложенности требует прохода алгоритмов измерения (measure) и размещения (layout). В BDUI, где структура формируется рекурсивно, риск создания избыточно глубоких деревьев возрастает.

Определим функцию глубины для вершины v рекурсивно:

$Depth(v) = 0$,если $v \in V$ (листовая вершина)$Depth(mv) = 1 + \max(\{Depth(x) \mid x \in Content(mv)\})$,где $mv \in MV$

Индекс NDI для всего экрана (корневой метавершины mv_{root}) определяется как высота дерева композиции: $NDI(\mathcal{M}_{UI}) = Depth(mv_{root})$

Интерпретация: Эмпирические исследования показывают, что при $NDI > 8$ время отрисовки кадра на устройствах среднего сегмента начинает превышать 16.6 мс (граница 60 FPS). Метрика позволяет автоматически помечать такие экраны как требующие рефакторинга (сплющивания иерархии) еще на этапе генерации JSON.

4.1.2. Взвешенный вес экрана (Screen Weight — SW) Простой подсчет количества элементов (nodes count) недостаточен, так как «стоимость» рендеринга текстовой метки и видеоплеера несопоставима. Введем весовую функцию $\omega: (V \cup MV) \to \mathbb{R}^+$, определяемую на основе типа компонента:

- $\omega_{light} = 1$ (Text, Spacer, Image без URL).
- $\omega_{medium} = 5$ (Container, Button, Image с URL).
- $\omega_{heavy} = 20$ (Video, Map, Webview, Pager).

Метрика Screen Weight: $SW(\mathcal{M}_{UI}) = \sum_{x \in (V \cup MV)} \omega(Type(x)) + \gamma \cdot |Links(x)|$ где γ — коэффициент стоимости

обработки связей (атрибутов binding). Эта метрика коррелирует с объемом оперативной памяти, расходуемой на клиенте, и временем парсинга JSON-ответа.

4.1.3. Индекс Графической Плотности (Graph Density Index — GDI) Отражает насыщенность интерфейса элементами. Определяется как отношение суммарного веса экрана к полезной площади (или количеству смысловых блоков). Высокий GDI при малом NDI свидетельствует о «плоском», но перегруженном интерфейсе (dashboard), что может быть негативно воспринято пользователем.

4.2 Оценка когнитивной и логической сложности (ICCI)

Если структурные метрики оценивают нагрузку на *устройство*, то метрики сложности оценивают нагрузку на *разработчика*, поддерживающего систему. Предлагается адаптировать понятие цикломатической сложности МакКейба для метаграфов.

Инвариант Когнитивной Сложности Интерфейса (ICCI): В BDUI сложность порождается не столько количеством элементов, сколько количеством динамических зависимостей (visibility conditions, actions, triggers). Формула расчета ICCI: $I_{CCI} = |\ ME\ | + \sum_{me \in ME}(\theta \cdot |\ Inputs(me)\ |) + \sum_{ag \in \mathcal{AG}} \beta \cdot Complexity(ag)$ где:

- $|\ ME\ |$ — количество метаребер (действий).
- $|\ Inputs(me)\ |$ — мощность входящего множества метаребра (сколько условий должно совпасть, чтобы действие сработало).
- $\mathcal{AG}$ — множество активных агентов (таймеры, валидаторы).
- θ, β — весовые коэффициенты (эмпирически $\theta = 1.5, \beta = 2.0$).

Интерпретация: Сценарий, где кнопка "Купить" активна всегда ($I \approx 1$), когнитивно проще, чем сценарий, где она активна, если "введен телефон" И "чекбокс нажат" И "сумма > 0" ($I \approx 5$). Высокий ICCI сигнализирует о риске возникновения логических ошибок ("багов") в бизнес-логике экрана.

4.3 Анализ связности и зацепления (Coupling Analysis)

В программной инженерии стремятся к низкой связности (Low Coupling). В BDUI это особенно важно: если изменение в одном виджете ломает другой, находящийся в другом конце экрана, отладка становится нетривиальной.

Введем метрику **Индекса Связности Действий (Action Coupling Index — ACI)**: $ACI = \frac{N_{cross}}{N_{total}}$ где:

- N_{total} — общее количество метаребер действий.
- N_{cross} — количество метаребер, у которых источник (X_p) и приемник (Y_p) принадлежат *разным* родительским метавершинам первого уровня.

Если $ACI \to 1$, интерфейс представляет собой "спагетти-структуру", где локальные действия имеют глобальные побочные эффекты. Это архитектурный антипаттерн, который метаграфовая модель позволяет выявить автоматически.

4.4 Алгоритмы автоматической верификации целостности

Одной из главных задач исследования является переход от пассивного наблюдения к активной верификации. Предлагаются следующие алгоритмы, работающие на Графе Метасостояний (State Metagraph).

4.4.1. Алгоритм обнаружения тупиковых состояний (Deadlock Detection) Тупиком в UI считается состояние, из которого пользователь не может перейти ни в какое другое полезное состояние или вернуться назад (кроме закрытия приложения).

Входные данные: Начальное состояние S_0, граф переходов G_{states}. *Алгоритм:*

1. Строим полное дерево достижимости (Reachability Tree) из S_0, моделируя все возможные Actions.
2. Помечаем состояния S_{term} как терминальные (успешное завершение сценария).
3. Для каждого листа дерева S_{leaf} проверяем: $S_{leaf} \notin S_{term} \wedge OutDegree(S_{leaf}) = 0 \implies \text{DEADLOCK}$
4. Дополнительно проверяем на наличие "ловушек" — сильных компонент связности, из которых нет выхода в S_{term} (пользователь ходит по кругу).

В терминах метаграфов это сводится к анализу топологии метаребер переходов.

4.4.2. Верификация контрактной совместимости (Version Gap Analysis) Алгоритм предотвращения отправки нового контента на старые клиенты. Пусть C_{ver} — версия клиентского приложения из заголовка запроса. Каждая вершина метаграфа v имеет атрибут $v.min\_ver$ (минимальная поддерживаемая версия).

Алгоритм проверки целостности ответа:

1. Обходим сгенерированный метаграф $\mathcal{M}_{UI}$.
2. Для каждой $v \in V \cup MV$: Если $v.min\_ver > C_{ver} \implies$ Anomaly Detected
3. При обнаружении аномалии запускается процедура поиска $fallback$ (см. раздел 3.6). Если $fallback$ не определен, метаграф считается невалидным, и отправка ответа блокируется.

4.4.3. Детекция "Висячих" ссылок (Dangling References) В BDUI часто используются идентификаторы для связывания (например, target_id: "input_email"). Частая ошибка: виджет удалили, а действие, ссылающееся на него, осталось. *Формально:* Проверка условия, что для любого метаребра $me = \langle X, Y \rangle$, все элементы множеств X и Y существуют в текущем срезе $V \cup MV$. $\forall me \in ME, \forall id \in (X_{Ids} \cup Y_{Ids}) \implies \exists v \in V : v.id = id$

4.5 Методология адаптивного синтеза (Adaptive UI Synthesis)

Метаграфовая модель позволяет перейти от простого *отображения* заранее заготовленных экранов к их *алгоритмическому синтезу* под конкретный контекст.

Постановка задачи синтеза: Дано:

- Множество доступных виджетов-кандидатов $\Omega = \{mv_1, mv_2, \ldots, mv_n\}$.
- Контекст пользователя $Context$ (например, медленный интернет 2G).
- Ограничение на вес экрана W_{max}.
- Функция полезности $Utility(mv_i, Context)$, определяющая релевантность виджета (например, "Stories" имеют высокую полезность для вовлечения, но низкую для функциональности).

Задача: Сформировать подмножество $\Omega' \subset \Omega$, такое что: $\sum_{mv \in \Omega'} SW(mv) \leq W_{max} \sum_{mv \in \Omega'} Utility(mv, Context) \rightarrow \max$

Это классическая **Задача о рюкзаке (Knapsack Problem)**, применяемая к UI. *Пример работы:* Если у пользователя медленный интернет (W_{max} низок), алгоритм исключает из метаграфа тяжелые метавершины (видео-баннеры), оставляя только функциональное ядро (кнопки, текст), максимизируя полезность в жестких ограничениях. Метаграфовая модель позволяет реализовать это как автоматический фильтр на стороне BFF.

4.6 Стратегии деградации интерфейса (Graceful Degradation)

В рамках синтеза необходимо формализовать стратегии обработки ошибок несовместимости (выявленных в п. 3.5.2). Предлагается использование **паттерна "Метавершина-Заместитель" (Proxy Metavertex)**.

В метаграфе для каждой критической метавершины $mv_{primary}$ определяется альтернативное ребро $e_{fallback}$, ведущее к $mv_{fallback}$.

$$Struktur(mv) = \begin{cases} mv_{primary}, & \text{если } CapabilityCheck() = true \\ mv_{fallback}, & \text{иначе} \end{cases}$$

Такая структура превращает метаграф в вероятностную модель, где структура ответа зависит от условий среды, но при этом гарантированно остается связной (валидной).

4.7 Интеграция методологии в цикл разработки (CI/CD)

Предложенная методология не должна оставаться теоретической абстракцией. Описывается схема интеграции анализатора в процесс разработки:

1. **Backend Build Time:** Статический анализ схем ответов, расчет ICCI, проверка на циклические зависимости в графе навигации.
2. **Runtime (BFF):** Динамический расчет SW (веса экрана) для каждого запроса. Логирование метрик для мониторинга "здоровья" UI.
3. **A/B Testing:** Использование метрик NDI и ACI как дополнительных факторов при анализе результатов тестов (например, "Вариант Б дал лучшую конверсию, но увеличил время рендеринга на 20% из-за высокого NDI").

5 Заключение

В настоящей научно-исследовательской работе решена задача формализации и алгоритмизации процессов проектирования динамических пользовательских интерфейсов (BDUI). Переход от эвристических подходов к строгому математическому моделированию позволил сформировать целостную картину управления сложностью в распределенных мобильных системах.

В ходе исследования получены следующие основные результаты:

1. **Проведен системный анализ проблематики BDUI (Глава 1).** Установлено, что современные подходы к Server-Driven UI, решая бизнес-задачи гибкости, переносят сложность на уровень межсервисного взаимодействия и конфигурационных файлов. Выявлены ключевые архитектурные ограничения («камни преткновения»): проблема версионирования контрактов, неконтролируемый рост вложенности элементов и потеря семантической связности. Доказано, что традиционные модели (графы, UML) не способны адекватно описывать гибридную природу таких систем, сочетающую свойства жесткой иерархии и сетевого взаимодействия.
2. **Разработана расширенная метаграфовая модель (Глава 2).** Предложен и детально описан математический формализм *активного аннотируемого метаграфа*. Ключевым достижением является объединение в единой модели статической структуры (через метавершины-контейнеры), динамики переходов (через метаребра действий) и внутренней логики компонентов (через агентное моделирование). Модель обеспечивает взаимно-однозначное соответствие (биекцию) с реальными JSON-контрактами промышленных фреймворков, что делает её применимой для анализа реальных систем. Введенное понятие двойственности «Дерево композиции — Сеть взаимодействий» позволило применить различные классы математических методов к анализу одной и той же системы.

3. **Сформирована методология анализа и синтеза (Глава 3).** На базе разработанной модели создан прикладной инструментарий для архитекторов и разработчиков.
    - Введена система количественных метрик (NDI, SW, ICCI, ACI), которая переводит понятие «качества интерфейса» из области субъективных оценок в область измеримых инженерных показателей.
    - Разработаны алгоритмы верификации, позволяющие автоматически детектировать структурные ошибки (тупиковые состояния, разрывы навигации, несовместимость версий) на этапе серверной генерации.
    - Сформулирована задача адаптивного синтеза интерфейса как задача дискретной оптимизации, что открывает путь к созданию полностью самоадаптирующихся UI, подстраивающихся под контекст пользователя (скорость сети, возможности устройства).

Итоговый вывод. Результаты работы демонстрируют, что метаграфовый подход является мощным и адекватным инструментом для решения задач программной инженерии в области мобильной разработки. Предложенная методология позволяет трансформировать процесс создания динамических интерфейсов из ручного «рисования» экранов в инженерную дисциплину, основанную на строгих правилах верификации и метриках сложности. Это создает фундамент для следующего поколения инструментов разработки, способных гарантировать надежность и производительность мобильных приложений в условиях постоянных изменений бизнес-требований.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.
2. Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017.
3. Basili, V. R., Caldiera, G., Rombach, H. D. The Goal Question Metric Approach. In Encyclopedia of Software Engineering. John Wiley & Sons, 1994.
4. Abramson, A., and J. A. Storer. "Metagraphs," in Metagraphs and Their Applications, vol. 8, Metagraphs and Their Applications, 2016.
5. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
6. Freeman, E., Robson, E., Bates, B., Sierra, K. Head First Design Patterns. O'Reilly Media, 2004.
7. Server-Driven UI: The Professionals' Guide. [Электронный ресурс]. URL: https://www.braze.com/resources/articles/server-driven-ui (дата обращения: 20.11.2025).
8. Basu A., Blanning R.W. Metagraphs and Their Applications. Springer, 2007.
9. Гапанюк Ю.Е. Основные положения многомерно-метаграфовой модели данных и знаний // CEUR Workshop Proceedings, 2019.
10. How Backend Driven UI is Changing the Approach to Mobile Development? [Электронный ресурс]. URL: https://oril.co/blog/how-backend-driven-ui-is-changing-the-approach-to-mobile-development/ (дата обращения: 20.11.2025).
11. Basu A., Robert W. Blanning. Metagraphs and their applications. – New York: Springer, 2007.
12. Gapanyuk Y.E. Metagraph approach to the information-analytical systems development // CEUR Workshop Proceedings. – 2019. – Vol. 2514. – P. 428-439.
13. Как мы строили BDUI: опыт Яндекс Маркета. 2024. URL: https://dev.go.yandex/blog/how-we-built-bdui-2024-07-12 (дата обращения: 10.10.2025).

14. Попков В.К. Математические модели связности. Новосибирск: ИВ-МиМГ СО РАН, 2006.
15. Как работает Backend-Driven UI на мобильном клиенте. URL: https://habr.com/ru/companies/ozontech/articles/661941/ (дата обращения: 11.10.2025).
16. Stumbling Blocks of Backend-Driven UI on Web. URL: https://dev.to/kirillunlimited/stumbling-blocks-of-backend-driven-ui-on-the-web-12bo (дата обращения: 10.10.2025).
17. McCabe, T. J. "A Complexity Measure". IEEE Transactions on Software Engineering, SE-2(4) – 1976. – P. 308-320.
18. Fenton, N. E., & Bieman, J. M. Software Metrics: A Rigorous and Practical Approach, Third Edition. // CRC Press – 2015.
19. Cormen, T. H. Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. — 4-е изд. — Cambridge : MIT Press, 2022.
20. Model-Based User-Interface-Entwicklung: Werkzeuge und Mmethoden / под ред. J. Vanderdonckt, F. Paterno. — Wiesbaden : Springer Vieweg, 2013