УДК: 004.94:519.178
DOI: 10.66523/gici.2026.v1.6

НАУЧНАЯ СТАТЬЯ OPEN ACCESS

Граф зависимостей игровых механик как аналитический объект: обнаружение дедлоков и перспективы формального анализа

Кугуракова В.В.[1]

[1] Казанский (Приволжский) федеральный университет, Казань, Россия

Автор для переписки: Афанасьев В.А.| vlada.kugurakova@gmail.com

Поступила в редакцию: 08.04.2025 | **Принята к публикации:** 28.04.2026 | **Опубликована:** 30.04.2026

Ключевые слова: развлекательные системы на основе местоположения, виртуальная реальность, разработка игр, шлем виртуальной реальности, пространственная калибровка.

Финансирование: *исследование не имело спонсорской поддержки.*

***Конфликт интересов**: авторы заявляют об отсутствии конфликта интересов.*

Аннотация

В данной статье представлен практический подход к пространственной калибровке для проектов развлекательных систем на основе местоположения (LBE) в виртуальной реальности (VR) на основе автономной гарнитуры Meta Quest 3 и фреймворка Unreal Engine 5. Предложенный метод двухточечной настройки выравнивает местоположение игрока на виртуальном уровне с координатной системой физической арены LBE. Используются простейшие геометрические операции и встроенные методы. В качестве промежуточного этапа вводится специальный уровень калибровки, где оператор записывает положения опорных точек и желаемое смещение поворота камеры. Параметры калибровки хранятся локально на каждой гарнитуре и автоматически переключаются между картами. В статье обсуждаются специфические для LBE ограничения безопасности и показано, что рассматриваемый конвейер может быть интегрирован в различные игровые проекты с минимальной перегрузкой.

The dependency graph of game mechanics as an analytical object: deadlock detection and prospects for formal analysis

Kugurakova V.V. Corresponding author: Kugurakova V.V. | vlada.kugurakova@gmail.com

Modern video games are complex systems comprising dozens of interdependent mechanics, in which structural defects — deadlocks in particular — are notoriously resistant to manual detection. This paper proposes treating the dependency graph of game mechanics not as a visual aid, but as an analytical object admitting formal operations. We show that a deadlock is present in a system of mechanics if and only if the graph contains a strongly connected component of cardinality greater than one, which makes it possible to detect cyclic blockings in $O(|V|+|E|)$ prior to the implementation stage. The method has been validated on a drone-racing simulator prototype: a structural deadlock in a system of three interdependent abilities was identified and eliminated at the design stage. We outline a research programme extending the approach: verification of temporal properties, detection of dominant strategies, synthesis of mechanics from specification, and structural complexity metrics for game design.

Keywords: *game mechanics, dependency graph, deadlock, Tarjan's algorithm, strongly connected components, formal methods, game design.*

***Для цитирования**: Кугуракова В.В. Граф зависимостей игровых механик как аналитический объект: обнаружение дедлоков и перспективы формального анализа. Международное сетевое научное рецензируемое издание "Геймификация, игры, креативные индустрии" 2026. Т. 1, № 1. С. 46-54 https://doi.org/10.66523/gici.2026.v1.5 / For citation: Kugurakova V.V. The dependency graph of game mechanics as an analytical object: deadlock detection and prospects for formal analysisInternational Peer-Reviewed Online Scientific Journal "Gamification, Games, Creative Industries", 2026, vol. 1, no. 1, pp. 46-54. https://doi.org/10.66523/gici.2026.v1.5*

Введение

Традиционный цикл разработки видеоигры — итеративный процесс, описанный как в классических работах по геймдизайну [9, 23, 24], так и в современных исследованиях практик разработки [1]: геймдизайнер описывает правила → программист реализует → тестировщик запускает → находит ошибку → возвращается к началу. Можно привести следующую цитату из [1]: «fixing a bug found in testing can be 15× more costly than if it had been found while it was in design» — устранение ошибки на этапе тестирования обходится в 15 раз дороже, чем если бы она была выявлена на этапе проектирования. Это делает раннее обнаружение структурных дефектов принципиально важной задачей. Однако при росте числа параллельно действующих механик — ресурсных систем, способностей персонажей, триггеров событий — в нём возникает принципиальная слабость: тестирование не может охватить все возможные комбинации состояний системы. Дефекты, возникающие на стыке нескольких механик, обнаруживаются поздно и обходятся дорого [1, 2].

Одним из наиболее опасных классов таких дефектов являются дедлоки* — ситуации взаимной блокировки, при которых две или более механики ожидают друг друга и ни одна не может завершиться. В традиционном понимании дедлоки ассоциируются с многопоточным программированием. Однако в игровых системах они возникают на более высоком уровне абстракции: не между потоками операционной системы, а между игровыми механиками, связанными зависимостями по данным и условиям активации. Ключевое наблюдение, лежащее в основе настоящей статьи, состоит в следующем: если описание игровых механик и их зависимостей формализовано в виде графа, этот граф перестаёт быть просто иллюстрацией — он становится аналитическим объектом, над которым определены формальные операции. Обнаружение дедлоков - первая и наиболее практически значимая из таких операций. Но не последняя: тот же граф допускает верификацию темпоральных свойств, поиск доминирующих стратегий, синтез механик по спецификации и вычисление метрик сложности.

2. Существующие инструменты моделирования механик и их ограничения

Прежде чем предлагать новый подход, важно зафиксировать, что именно не позволяют делать существующие инструменты. Речь идёт не об их общей оценке — каждый из них решает свои задачи хорошо, — а о конкретном вопросе: допускают ли они автоматическое обнаружение структурных дефектов, таких как дедлоки, на уровне проектирования?

Machinations [5] — наиболее развитый из инструментов визуального моделирования игровой экономики — предоставляет нотацию токен-потоков и поддерживает симуляцию ресурсных систем [2]. Machinations позволяет наблюдать поведение системы, но не анализировать её структуру: в инструменте нет формальной операционной семантики, определяющей, что является «зависимостью» между механиками. Как следствие Machinations не может обнаружить циклическую зависимость алгоритмически - только визуально, если разработчик её заметит.
Визуальные скриптовые системы — Unreal Engine Blueprint [21], Unity Visual Scripting[1] [31] — позволяют создавать игровую логику без написания кода через графовый интерфейс узлов и связей. Однако эти графы являются графами исполнения кода, а не графами зависимостей между механиками. Анализ таких графов требует запуска — то есть реализации, — тогда как мы предлагаем сфокусироваться на анализе до этапа реализации.

Событийно-ориентированные движки (GDevelop [22], GameMaker [32]) предоставляют удобные средства описания реакций на события, но не имеют понятия «зависимость между механиками» как явной структуры данных, доступной для алгоритмического анализа. Ручное тестирование и экспертная проверка остаются основными методами обнаружения структурных дефектов на практике [1, 2]. Их принципиальное ограничение - не масштабируемость: число возможных комбинаций состояний растёт экспоненциально с числом механик, и исчерпывающая проверка вручную практически невозможна.

Таким образом, общее ограничение перечисленных инструментов: они оперируют с механиками как с визуальными или исполняемыми объектами, но не как с формальными структурами, допускающими алгоритмический анализ до реализации.

3. Смежные исследования

В последние годы появились работы, в которых предпринимаются попытки применить формальные методы к игровым системам — однако каждая из них затрагивает лишь часть проблемы.

Резин и Шлинглофф [27] предложили строить NFA-модель[1] игры непосредственно из её описания и применять к ней model checking[2]. Их инструмент

* Deadlock (англ. «взаимная блокировка») — ситуация циклического ожидания, при которой ни один из участников не может продолжить выполнение [25]. В данной работе термин применяется к игровым механикам: дедлок возникает, когда две или более механики взаимно зависят друг от друга и не могут активироваться. См. также [30].

SafeGame верифицирует свойства мультиплеерных игр через NuSMV. Авторы прямо констатируют: «The gap between research in formal verification and its adoption in actual game development is still significant» — разрыв между исследованиями формальной верификации и её применением в разработке игр по-прежнему значителен. Ограничение подхода: модель строится по готовому описанию всей игры как единому автомату, а не по структуре зависимостей между отдельными механиками. Структурные дедлоки в таком подходе не выделяются как самостоятельный класс дефектов.

*NFA (Nondeterministic Finite Automaton, недетерминированный конечный автомат) — абстрактная машина, допускающая наличие нескольких возможных следующих состояний для одного входного символа; в работе [27] NFA используется как упрощённое математическое представление игровой динамики, которое затем подвергается процедуре model checking.
Model checking (проверка моделей) — метод автоматической формальной верификации, при котором алгоритмически проверяется, удовлетворяет ли формальная модель системы заданным спецификациям (обычно формулам темпоральной логики).

Хасегава и Ёкогава [28] проанализировали базу дефектов, накопленную в ходе разработки Final Fantasy XV[1], и обнаружили, что значительная часть ошибок была вызвана простыми несоответствиями в визуальных скриптах и могла быть выявлена механически. Авторы предложили транслировать Blueprint-подобные скрипты в модели NuSMV и автоматически верифицировать их. Этот подход концептуально близок к предлагаемому — те же визуальные правила, тот же верификатор. Принципиальное отличие: Хасегава и Ёкогава работают с уже написанными скриптами (уровень реализации), тогда как здесь предлагается анализ графа зависимостей до написания кода (уровень проектирования).

Стивенс и Экстон [29], опираясь на формализм Machinations, предложили оценивать экономики мультиплеерных игр без масштабного плейтестирования — выявлять потенциальные эксплойты через симуляцию ресурсных потоков.

*NFA (Nondeterministic Finite Automaton, недетерминированный конечный автомат) — абстрактная машина, допускающая наличие нескольких возможных следующих состояний для одного входного символа; в работе [27] NFA используется как упрощённое математическое представление игровой динамики, которое затем подвергается процедуре model checking.

Это ближайший по духу подход: анализ до полноценного тестирования. Однако симуляция в Machinations не даёт формальных гарантий — она обнаруживает дефекты статистически, а не исчерпывающим образом. Таким образом, предлагаемый в настоящей статье подход занимает нишу, не заполненную ни одной из существующих работ: структурный анализ графа зависимостей механик на уровне проектирования, дающий формальные гарантии отсутствия дедлоков за до написания кода.

4. Граф зависимостей игровых механик

Понятие «игровая механика» не имеет единого общепринятого определения в литературе, и это само по себе примечательно: разные исследователи акцентируют разные аспекты. Кроуфорд [9] понимает механики как правила, управляющие взаимодействием игрока с системой. Саллен и Циммерман [10] определяют механики как конкретные взаимодействия, которые игрок может выполнять в контексте игры. Юнике, ЛеБлан и Зубек в рамках MDA-концепции [11] трактуют механики как данные и алгоритмы — компонент системы, порождающий поведение. Сикарт даёт, пожалуй, наиболее прикладное определение: «методы, вызываемые агентами, предназначенные для взаимодействия с игровым состоянием» [12].
При всём разнообразии формулировок во всех этих определениях прослеживается общая структура: механика — это правило активации (что должно быть выполнено), правило эффекта (что изменяется в состоянии) и, опционально, стоимость (что расходуется). Именно эта структура лежит в основе формального определения, используемого в настоящей статье. В работе [3] была предложена формальная модель видеоигры (где S - множество игровых состояний, A - множество действий агентов, δ - функция переходов, R - декларативные правила, Π - множество агентов, ω - функция наблюдаемости, W - множество терминальных состояний)

$$G = \langle S, A, \delta, R, \Pi, \omega, W \rangle$$

и введён формализм FAST-GM, в рамках которого механика задаётся тройкой

$$m \langle pre_m, eff_m, cost_m \rangle$$

где pre_m — предусловие применимости, eff_m — функция эффекта, $cost_m$ — функция стоимости. Полная игровая система описывается набором механик,

$$M = \{m_1, m_2, \dots, m\}$$

связанных зависимостями: предусловие одной механики может зависеть от результата применения другой. Граф зависимостей механик

$$G_{dep} = (V_m, E_{dep})$$

строится следующим образом. Каждая вершина $v \in V_m$ соответствует одной механике.

Направленное ребро $(m_i, m_j) \in E_{dep}$ проводится тогда, когда предусловие pre_{m_i} содержит ссылку на результат применения механики m_j — то есть m_i не может активироваться до завершения m_j. Такой граф является корректным формальным представлением: он однозначно описывает структуру зависимостей вне зависимости от конкретного способа реализации механик.

Принципиально важно, что граф зависимостей строится на уровне проектирования — до написания кода. Это позволяет выявлять структурные дефекты на самом раннем этапе, когда их исправление требует минимальных затрат [1]. В платформе Tula [4] граф визуально строится в редакторе GD-FST, после чего автоматически подвергается формальному анализу. Ключевое свойство графа зависимостей, делающее его аналитически полезным: наличие цикла в G_{dep} свидетельствует о взаимной зависимости механик, потенциально приводящей к дедлоку. Если m_i ждёт m_j ,а m_j ждёт m_i — ни одна из них никогда не активируется. Детектировать такие циклы — значит обнаружить дедлоки на уровне модели.

5. Обнаружение дедлоков: сильносвязные компоненты и алгоритм Тарьяна

Дедлок в графе зависимостей механик присутствует тогда и только тогда, когда граф G_{dep} содержит сильносвязную компоненту (Strongly Connected Component, SCC) мощностью более одной вершины. Сильносвязная компонента — это максимальный подграф, в котором из каждой вершины достижима каждая другая. Механики, образующие такой подграф, взаимно зависят друг от друга и не могут выполниться ни одна из них первой.

Лемма 1 (критерий структурного дедлока).

Пусть $G_{dep} = (V_m, E_{dep})$ — граф зависимостей игровых механик. Система механик M содержит структурный дедлок тогда и только тогда, когда в G_{dep} существует SCC мощностью $|SCC| > 1$. При этом каждая SCC мощностью $k > 1$ соответствует независимому дедлоку, вовлекающему ровно k механик.

Доказательство.

(⇒) Пусть система содержит структурный дедлок.

По определению, существует подмножество механик $M' \subseteq M$, $|M'| \geq 2$, такое что для каждой $m \in M'$ множество её прямых предшественников в G_{dep} непусто и целиком содержится в M'.

В конечном ориентированном графе подграф, в котором каждая вершина имеет полустепень захода* ≥ 1, обязательно содержит хотя бы один направленный цикл. Наличие цикла означает, что все вершины этого цикла принадлежат одной сильносвязной компоненте (SCC) мощности ≥ 2.

(⇐) Пусть в G_{dep} существует SCC мощности $k > 1$.

По определению сильносвязности, для любой пары $u, v \in SCC$ существует направленный путь $u \to \cdots \to v$. В частности, каждая механика в SCC транзитивно зависит от всех остальных механик той же компоненты. Поскольку активация механики требует завершения всех её прямых зависимостей, а в цикле нет вершины с нулевой полустепенью захода внутри SCC, ни одна механика из данной компоненты не может получить право на выполнение первой. Следовательно, система переходит в состояние структурного дедлока.

Замечание 1. Следует отметить, что критерий обнаружения взаимоблокировок через сильносвязные компоненты ориентированного графа известен в информатике - в частности, применительно к задачам планирования ресурсов операционных систем [25] и анализа транзакций в базах данных [26]. Новизна Леммы 1 в контексте настоящей работы состоит не в самом критерии, а в объекте его применения: граф зависимостей игровых механик G_{dep} как формальный объект вводится здесь впервые, и именно его определение позволяет перенести классический результат на предметную область видеоигр.

Замечание 2. Критерий Леммы 1 является строгим для структурных (статических) дедлоков, определяемых исключительно топологией G_{dep} независимо от текущих значений игровых параметров. Для учёта условных зависимостей, активируемых только при определённых значениях игровых параметров, рёбра графа могут быть снабжены логическими аннотациями $\varphi_{ij} \subseteq \Phi$, где Φ - пространство состояний игры. В этом случае наличие SCC в G_{dep} остаётся необходимым условием дедлока, но достаточность проверяется дополнительно через анализ выполнимости конъюнкции аннотаций вдоль цикла:

$$\bigwedge_{(m_i\, m_j) \in C} \varphi_{ij} \neq \bot.$$

*Полустепень захода (англ. indegree) - число рёбер, входящих в вершину в ориентированном графе. В контексте графа механик полустепень захода означает, что у данной механики есть как минимум одна зависимость (предусловие), которая должна быть выполнена до её активации.

Если конъюнкция невыполнима, цикл C является «виртуальным» и не приводит к блокировке в рантайме. Для поиска всех SCC применяется алгоритм Тарьяна [6]. Алгоритм выполняет обход в глубину (Depth-First Search, DFS), поддерживая для каждой вершины два значения: порядковый номер обнаружения (disc – порядок первого посещения вершины) и значение lowlink (наименьший порядковый номер среди вершин, достижимых из данной по рёбрам графа, включая обратные рёбра). Вершина является корнем SCC тогда, когда disc=lowlink — в этот момент из стека извлекаются все вершины, образующие данную компоненту.

Выбор алгоритма Тарьяна [6] среди альтернатив обусловлен тремя соображениями. Алгоритм Флойда-Уоршелла [13] находит кратчайшие пути между всеми парами вершин и может обнаруживать циклы через транзитивное замыкание, однако имеет сложность $O(|V|^3)$ — неприемлемую для интерактивного анализа при сотнях механик. Алгоритм Косарайю-Шарира [14] также работает за $O(|V| + |E|)$, однако требует двух проходов DFS и построения транспонированного графа, что удваивает константу. Алгоритм Тарьяна обнаруживает все SCC за один проход DFS при пространственной сложности $O(|V|)$ — оптимальный выбор для встроенного анализатора реального времени. При обнаружении SCC мощностью более одной вершины система возвращает диагностическое сообщение: список механик, образующих цикл, с указанием конкретных зависимостей, приводящих к блокировке. Это позволяет геймдизайнеру немедленно локализовать проблему и принять решение: разорвать зависимость, ввести дополнительное условие активации или перепроектировать механику.

6. Практический кейс: симулятор гонки дронов

Рассмотрим применение метода к игровому прототипу мобильного симулятора гонки дронов с тремя параллельными способностями: boost (ускорение), shield (защитный щит) и stunt (акробатический манёвр). Каждая способность имеет предусловия активации и побочные эффекты, влияющие на другие способности. При проектировании системы между ними была введена следующая логика синхронизации:

- способность boost не может активироваться, пока не завершится stunt (предусловие: stunt.completed);
- способность shield требует завершения boost (предусловие: boost.completed);
- способность stunt требует завершения shield (предусловие: shield.completed).

Граф зависимостей для этой системы содержит три вершины {boost, shield, stunt} и три ребра: boost → stunt, shield → boost, stunt → shield. Визуально это выглядит как треугольник – замкнутый цикл из трёх вершин. Граф до исправления показан на рис. 1, граф после — на рис. 2.

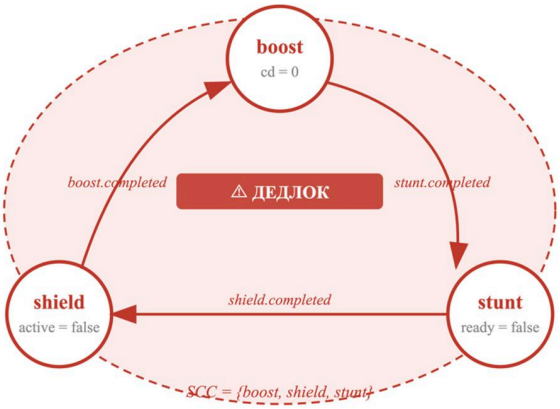

Рисунок 1 - Граф зависимостей способностей дрона: дедлок (цикл выделен)

Применение алгоритма Тарьяна немедленно выявляет единственную SCC={boost, shield, stunt} мощностью 3 — дедлок подтверждён.

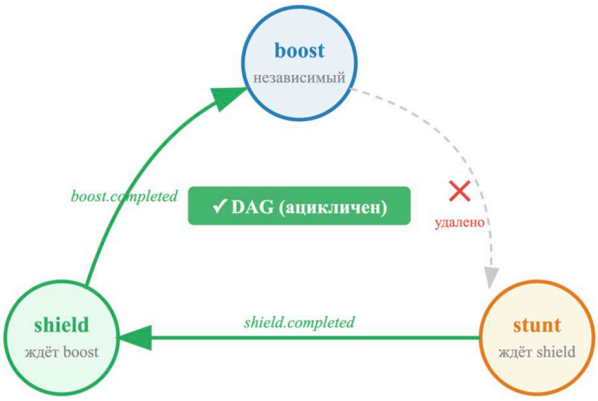

Рисунок 2 - Граф зависимостей после устранения дедлока (ациклический граф)

Аналитическая диагностика: ни одна из трёх способностей не может быть первой активирована, поскольку каждая из них ждёт завершения другой. В терминах цикла разработки — это дефект, который при традиционном тестировании обнаруживается только тогда, когда тестировщик пытается использовать все три способности в определённой последовательности и обнаруживает, что игра «зависла». Формальный анализ выявил его на этапе проектирования.

Исправление: введена приоритетная иерархия активации – boost может активироваться независимо, shield требует только завершения boost, stunt требует завершения shield. Зависимость boost → stunt убрана. Повторный анализ: SCC мощностью более 1 не обнаружены – граф зависимостей ациклический (Directed Acyclic Graph, DAG), дедлок устранён.

Важно корректно интерпретировать это сравнение: речь идёт не о конкурирующих методах для одной задачи, а о двух принципиально разных уровнях абстракции. Анализ графа зависимостей работает на уровне проектирования и обнаруживает структурные дедлоки — те, что определяются топологией зависимостей, — за $O(|V|+|E|$, до написания кода. Верификатор SPIN [33] работает на уровне реализации: в системе с 470 000 состояний он обнаружил дедлок с конкретной трассой длиной 23 шага, что заняло 45 минут [3]. Разница во времени объясняется не превосходством одного метода над другим, а тем, что SPIN решал более трудную задачу — анализировал полное пространство состояний реализованной системы. Метод на основе G_{dep} работает быстрее ценой меньшей выразительности: он не обнаруживает динамические дедлоки, зависящие от значений параметров в рантайме (раздел 7.6). Оба инструмента взаимодополняют друг друга и реализованы в платформе Tula [4].

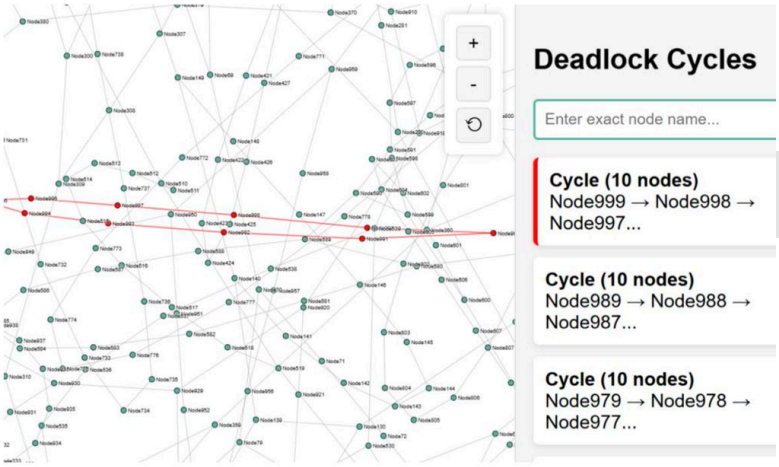

Рисунок 3 — Визуализация обнаруженных структурных дедлоков (красным цветом выделены рёбра и вершины, образующие SCC ; в правой панели перечислены обнаруженные циклы с указанием задействованных узлов и длины цикла)

Метод интегрирован в платформу Tula [4]: при обнаружении SCC алгоритм автоматически подсвечивает циклические зависимости в редакторе графа (рис. 3) и выводит список затронутых механик в диагностической панели. Техническая реализация алгоритма Тарьяна описана в [37].

7. Граф зависимостей как платформа для формального анализа: перспективные направления

Обнаружение дедлоков демонстрирует общий принцип: формализованное представление игровых механик в виде графа открывает возможность применения к нему разнообразных аналитических операций. Те же структурные свойства, которые позволяют обнаруживать дедлоки через поиск, могут служить основой для более широкого класса формальных операций — каждая из которых добавляет новый уровень аналитической глубины к процессу геймдизайна. Ниже описаны наиболее перспективные из них.

7.1. Верификация темпоральных свойств

Граф GD-FST непосредственно подаётся на вход верификаторам SPIN [33], NuSMV [34], UPPAAL [35] и PRISM [36], позволяя проверять свойства вида «механика атаки недоступна в течение кулдауна», «квест всегда достижим при выполнении предусловий», «вероятность победы не превышает 0.6» [3]. Эта интеграция реализована в платформе Tula [4] и является наиболее разработанным из перспективных направлений.

Связь с темпоральной верификацией. Структурный критерий дедлока естественно встраивается в рамки вычислительной темпоральной логики (Computation Tree Logic, CTL). Условие отсутствия структурных дедлоков эквивалентно выполнению CTL-формулы*

$$\boldsymbol{AG\ EF}\ (\exists m \in M: nabled(m))^1$$

на абстрактной модели переходов, порождённой G_{dep}. На практике обнаружение SCC мощностью > 1 даёт быстрый статический контрпример к свойству $\boldsymbol{AG}\ \neg deadlock$, что позволяет отсеять целый класс некорректных спецификаций до построения полной модели состояний для model checking (SPIN, NuSMV). Таким образом, анализ SCC выступает эффективным pre-check фильтром, сокращающим пространство поиска для последующей темпоральной верификации.

7.2. Синтез механик по спецификации.

Задача, обратная верификации: вместо проверки того, удовлетворяет ли механика свойству φ, автоматически строится механика, которая по конструкции удовлетворяет φ. Это задача реактивного синтеза, активно исследуемая в формальных методах [7], но почти не применявшаяся к игровым системам.

*Операторы CTL: (All paths) — «для всех путей», (Exists path) — «существует путь», (Globally) — «глобально/всегда», (Finally) — «в будущем/рано или поздно». Формула читается как «для всех [1] Операторы CTL: (All paths) — «для всех путей», (Exists path) — «существует путь», (Globally) — «глобально/всегда», (Finally) — «в будущем/рано или поздно». Формула читается как «для всех путей глобально верно, что существует путь, на котором рано или поздно выполнится» — это стандартная формулировка свойства отсутствия тупиковых состояний (deadlock-freedom). Предикат означает, что механика может быть активирована в текущем состоянии (её предусловие выполнено).

7.3. Выявление доминирующих стратегий

Операция над графом, позволяющая формально доказать отсутствие стратегий, доминирующих при всех начальных условиях независимо от значений параметров. В отличие от балансировки (поиска оптимального θ^*), это качественная гарантия: «при данной архитектуре механик доминирующей стратегии не существует в принципе» [8]. В теории игр стратегия называется строго доминирующей, если она обеспечивает агенту лучший результат по сравнению с любой альтернативной стратегией при любом поведении остальных агентов [12]. Наличие доминирующей стратегии — признак дисбаланса принципиально иного рода, чем неоптимальные значения параметров: здесь проблема заключена в самой архитектуре механик, и никакая настройка θ её не устранит. Полная верификация отсутствия доминирующих стратегий для стохастических систем выполняется средствами PRISM [36] через верификацию MDP-свойств: равенство

$$Pmax =?[F\ win] \ = \ Pmin =?[F\ win]$$

означает, что наилучшая* и наихудшая** стратегии дают одинаковую вероятность победы – признак отсутствия доминирования.

7.4. Нарративная связность

Граф переходов состояний допускает формальную проверку нарративных свойств: достижимость всех финалов, наличие значимых выборов на каждом пути, отсутствие «тупиковых» ветвей. Это прямое применение CTL-верификации к графу сценария, построенному парсером DSL [4].

• Достижимость всех финалов: для каждого терминального состояния должен существовать хотя бы один путь из начального состояния, т. е. . Нарушение этого свойства означает, что часть концовок игры недостижима при любом поведении игрока: сюжетная ветвь существует в коде, но игрок никогда до неё не доберётся.
• Наличие значимых выборов: на каждом пути не должно быть состояний, из которых все исходящие переходы ведут в одно и то же следующее состояние, т. е. . Формально это означает, что выбор игрока имеет наблюдаемые последствия: нет «иллюзорных развилок», где все ветви немедленно сходятся.
• Отсутствие тупиковых ветвей (softlock): из любого достижимого состояния должен существовать путь к хотя бы одному терминальному состоянию, т. е. . Нарушение означает ситуацию, когда игрок оказывается в состоянии, из которого невозможно завершить игру – известная категория дефектов в нелинейных играх.

Инструментальная основа для верификации этих свойств в платформе Tula уже создана: DSL-парсер сценариев строит граф достижимости, над которым применяется CTL-верификация через NuSMV [4]. Таким образом, нарративная верификация является естественным расширением существующего цикла «спецификация-верификация», требующим только формализации сценарных свойств в виде CTL-формул.

7.5. Метрики сложности игровой системы

Вопрос объективного измерения сложности геймдизайна исследуется в литературе, однако до сих пор не имеет общепринятого решения. Существующие подходы можно разделить на три класса.

Первый класс – метрики пространства состояний. Классическая мера – мощность пространства состояний игры (число достижимых позиций) [15] – хорошо определена для настольных игр (шахматы, го), но трудно применима к видеоиграм с непрерывными параметрами. Работа [16] предлагает два агентных показателя для процедурно генерируемых уровней: разнообразие (расстояние Левенштейна [38] между траекториями A*-агента [39]) и сложность (объём поискового дерева A* до нахождения решения). Эти метрики не зависят от конкретной игры и не требуют предметно-ориентированной разметки.

Второй класс — энтропийные метрики. В [17] вводится мера «избыточной стратегической энтропии» — информационно-теоретическую характеристику предсказуемости смешанной стратегии игрока. В [18] предлагают «глубину» игры как человекоориентированную меру, коррелирующую с разрывом в силе игры между уровнями мастерства и не совпадающую с вычислительной сложностью. Оба подхода отражают разные аспекты одного явления: насколько «богата» игровая система с точки зрения стратегического выбора.

Третий класс — структурные метрики по графу. Цикломатическая сложность Маккейба [19], разработанная для программного кода, применима к графу зависимостей механик как мера числа независимых путей. Работа по анализу пространства состояний игр-головоломок методом раскрашенных сетей Петри [20] показывает, что формальный анализ позволяет автоматически извлекать количественные характеристики сложности.

Общий вывод из обзора: существующие метрики либо ориентированы на конкретные жанры (уровни платформеров, настольные игры), либо требуют запуска симуляции. Вычисление метрик непосредственно по графу зависимостей механик — до симуляции и реализации — остаётся нерешённой задачей.

Именно здесь граф G_{dep}, введённый в настоящей статье, открывает возможность определить такие метрики структурно: цикломатическая сложность — по рёбрам графа; энтропия — по распределению степеней вершин; глубина планирования — по длине максимального ациклического пути.
Перечисленные направления образуют программу исследований, в которой граф зависимостей игровых механик выступает единым формальным основанием для разных классов аналитических задач. Настоящая статья представляет первый реализованный элемент этой программы.

7.6. Ограничения предложенного метода

Метод обнаружения дедлоков через анализ обнаруживает структурные дедлоки — те, что определяются топологией графа зависимостей независимо от текущих значений игровых параметров. Это необходимое, но не достаточное условие отсутствия всех возможных дедлоков.
Динамические дедлоки — возникающие только при определённых значениях параметров в рантайме — данным методом не обнаруживаются. Например, если механика *A* ожидает ресурса *R*, а механика *B* захватывает *R* только при условии $health > 50$, то зависимость видна лишь при конкретных значениях $health$
Для обнаружения таких дефектов необходимы верификация на уровне реализации (SPIN, NuSMV) или мониторинг графа ожидания (wait-for graph [26]) в рантайме — алгоритм предотвращения дедлоков Дейкстры, использующийся в операционных системах [25], либо динамический анализ цикличности в системах управления базами данных. Таким образом, анализ G_{dep} и верификация на уровне реализации являются взаимодополняющими инструментами одного цикла разработки, а не конкурирующими альтернативами.

Для обнаружения динамических дедлоков применяются методы другого уровня. Алгоритм Banker [26] предотвращает дедлоки в рантайме, динамически проверяя безопасность каждого запроса ресурса. Wait-for graph — граф ожидания, обновляемый в реальном времени, — используется в системах управления базами данных для обнаружения дедлоков в транзакциях [26]. Верификация через SPIN и NuSMV при полной модели с конкретными значениями параметров также позволяет обнаруживать параметрически зависимые дедлоки [3].

Таким образом, анализ G_{dep} и верификация на уровне реализации — не конкурирующие, а взаимодополняющие методы: первый работает быстро на этапе проектирования и устраняет целый класс дефектов до написания кода; второй обеспечивает более полные гарантии на уровне реализации ценой большей вычислительной сложности.

8. Заключение

В статье предложено рассматривать граф зависимостей игровых механик не как визуальный инструмент геймдизайна, а как аналитический объект, допускающий формальные операции. На примере задачи обнаружения дедлоков показано, что применение алгоритма Тарьяна к этому графу позволяет выявлять циклические блокировки на этапе проектирования — до реализации и тестирования. Практический кейс демонстрирует обнаружение и устранение дедлока в системе из трёх способностей дрона за время, измеряемое миллисекундами.
Предложенный подход меняет цикл разработки: между этапами «проектирование» и «реализация» появляется новый этап — формальный анализ модели. Этот этап не требует написания кода и не зависит от конкретной игровой платформы; он работает с абстрактным описанием механик и их зависимостей. Обнаружение дедлоков — первый шаг в программе формального анализа игровых систем. Последующие шаги — верификация темпоральных и вероятностных свойств, выявление доминирующих стратегий, синтез механик по спецификации, метрики сложности — используют тот же граф зависимостей как аналитический объект и открывают перспективу полноценной формальной инженерии игровых систем.

Список литературы

1. Cho J., Ali K. Exploring quality assurance practices and tools for indie games // 2023 IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS). – IEEE, 2023. – P. 16-24.
2. Сахибгареева Г.Ф., Кугуракова В.В., Большаков Э.С. Инструменты балансирования игр // Электронные библиотеки. – 2023. – Т. 26, № 2. – С. 225-251.
3. Кугуракова В.В. A Formal Approach to Spatio-Temporal Modeling of Game Systems // Учёные записки Казанского университета. Серия: физико-математические науки. – 2024. – Т. 166, № 4. – С. 532-554.
4. Рахманкулова В.Р., Кугуракова В.В. Онлайн-инструмент Tula для балансировки видеоигр // Электронные библиотеки. – 2024. – Т. 28, № 4. – С. 903-930.
5. Dormans J. Machinations: Elemental feedback systems for game design // Proceedings of the Analog Game Studies Conference. – 2008. – P. 33-40.
6. Tarjan R.E. Depth-first search and linear graph algorithms // SIAM Journal on Computing. – 1972. – Vol. 1, No. 2. – P. 146-160.
7. Pnueli A., Rosner R. On the synthesis of a reactive module // Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. – 1989. – P. 179-190.
8. Кугуракова В.В. Видеоигра как информационная система: формальные основания и классификация задач // Информационное общество. – 2026. (на рецензии).
9. Crawford C. The Art of Computer Game Design. – Berkeley: Osborne/McGraw-Hill, 1984. – 114 p.
10. Salen K., Zimmerman E. Rules of Play: Game Design Fundamentals. – Cambridge: MIT Press, 2004. – 672 p.
11. Hunicke R., LeBlanc M., Zubek R. MDA: A formal approach to game design and game research // Proceedings of the AAAI Workshop on Challenges in Game AI. – 2004. – P. 1-5.
12. Sicart M. Defining game mechanics // Game Studies. – 2008. – Vol. 8, No. 2. – P. 1-14.
13. Floyd R.W. Algorithm 97: Shortest path // Communications of the ACM. – 1962. – Vol. 5, No. 6. – P. 345.

14. Sharir M. A strong-connectivity algorithm and its applications in data flow analysis // Computers & Mathematics with Applications. – 1981. – Vol. 7, No. 1. – P. 67-72.
15. Hearn R.A., Demaine E.D. Games, Puzzles, and Computation. – Wellesley: A.K. Peters, 2009. – 248 p.
16. Beukman M., James S., Cleghorn C. Towards objective metrics for procedurally generated video game levels // arXiv:2201.10334. – 2022.
17. Falniowski F. Entropy-based measure of statistical complexity of a game strategy // Entropy. – 2020. – Vol. 22, No. 4. – P. 470.
18. Cauwet M.-L., Teytaud O., Cazenave T., Saffidine A. Depth, balancing, and limits of the Elo model // IEEE Conference on Computational Intelligence and Games (CIG). – IEEE, 2015. – P. 376-382.
19. McCabe T.J. A complexity measure // IEEE Transactions on Software Engineering. – 1976. – Vol. SE-2, No. 4. – P. 308-320.
20. Arabzadeh M. et al. State-space analysis and complexity assessment of puzzle games using colored Petri nets // International Journal of Web Research. – 2024. – Vol. 7, No. 2. – P. 13-27.
21. Unreal Engine Blueprint Visual Scripting System. – Epic Games, 2023. – URL: https://docs.unrealengine.com/en-US/blueprints-visual-scripting
22. GDevelop – Open-Source Game Development Software. – URL: https://gdevelop.io
23. Schell J. The Art of Game Design: A Book of Lenses. – Burlington: Morgan Kaufmann, 2008. – 489 p.
24. Zimmerman E. Play as research: The iterative design process // Design Research: Methods and Perspectives / Ed. B. Laurel. – Cambridge: MIT Press, 2003. – P. 176-184.
25. Dijkstra E.W. Co-operating Sequential Processes // Programming Languages / Ed. F. Genuys. – New York: Academic Press, 1968. – P. 43-112.
26. Silberschatz A., Galvin P.B., Gagne G. Operating System Concepts. 10th ed. – Hoboken: Wiley, 2018. – 938 p.
27. Rezin V., Schlingloff B.-H. Model checking in multiplayer games development // arXiv:1712.01207. – 2017.
28. Hasegawa I., Yokogawa T. Formal verification for node-based visual scripts using symbolic model checking // IEICE Transactions on Information and Systems. – 2022. – Vol. E105-D, No. 1. – P. 78-91. DOI: 10.1587/transinf.2021EDP7063.
29. Stephens C., Exton C. Towards a framework for the formal assessment of online multiplayer game economies // Proceedings of the 16th International Conference on the Foundations of Digital Games. – 2021.
30. Таненбаум Э., Бос Х. Операционные системы: разработка и реализация. – 5-е изд. – СПб.: Питер, 2015. – 816 с. – С. 412-418.
31. Unity Visual Scripting. – Unity Technologies, 2023. – URL: https://docs.unity3d.com/Packages/com.unity.visualscripting@latest
32. GameMaker Studio 2. – YoYo Games, 2023. – URL: https://docs.yoyogames.com
33. Holzmann G.J. The SPIN Model Checker: Primer and Reference Manual. – Boston: Addison-Wesley Professional, 2004. – 340 p. – URL: https://spinroot.com
34. NuSMV: a New Symbolic Model Checker. – FBK-irst, University of Trento, 2023. – URL: https://nusmv.fbk.eu
35. UPPAAL – Tool for Modeling, Simulation and Verification of Real-Time Systems. — Uppsala University & Aalborg University, 2023. – URL: https://uppaal.org
36. PRISM – Probabilistic Symbolic Model Checker. – University of Oxford, 2023. — URL: https://www.prismmodelchecker.org
37. Крмаков Э.В. Обнаружение дедлоков в игровых прототипах с помощью алгоритмов поиска циклов: выпуск.квалиф.раб. на с.з. Бакалавр, спец. 09.03.04 - Программная инженерия, науч.рук. Кугуракова В.В., Казанский федеральный университет, Институт информационных технологий и интеллектуальных систем, 2025. 44 с.
38. Levenshtein V.I. Binary codes capable of correcting deletions, insertions, and reversals // Soviet Physics Doklady. — 1966. — Vol. 10, No. 8. — P. 707-710.
39. Hart P.E., Nilsson N.J., Raphael B. A formal basis for the heuristic determination of minimum cost paths // IEEE Transactions on Systems Science and Cybernetics. — 1968. — Vol. 4, No. 2. — P. 100-107.

ЛУЧШАЯ НАУЧНАЯ
СТАТЬЯ НОМЕРА
ПО МНЕНИЮ РЕДАКЦИИ