

ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ПРОЦЕССЫ/INFORMATICS AND INFORMATION PROCESSES

DOI: <https://doi.org/10.60797/IRJ.2026.172.77> EDN: WWYNHO

МЕТОДЫ АНАЛИЗА И ВИЗУАЛИЗАЦИИ ЗАВИСИМОСТЕЙ ES-МОДУЛЕЙ В JAVASCRIPT-ПРИЛОЖЕНИЯХ С ПОМОЩЬЮ ГРАФОВЫХ БАЗ ДАННЫХ

Научная статья

Астахова И.Ф.^{1,*}, Крючков Н.С.², Борисенков Д.В.³³ORCID : 0009-0005-8770-1081;^{1,2,3} Воронежский государственный университет, Воронеж, Российская Федерация

* Корреспондирующий автор (astachova[at]list.ru)

Предложена: 05.07.2026; Принята: 24.09.2026; Опубликовано: 16.10.2026

Аннотация

Анализ зависимостей программных модулей в приложениях становится с каждым годом все более и более важным этапом разработки современных приложений. Создание общей математической модели, описывающей анализ зависимостей на фундаментальном уровне — одна из задач, которые могут позволить ускорить разработку новых утилит для анализа зависимостей для самых разных систем модулей и языков программирования применение графовой БД. В представленной статье была предпринята попытка систематизировать существующие подходы к анализу модульных зависимостей в JavaScript, провести обзор технических решений и обозначить методологические основы будущего исследования. Показано, что применение графового представления и визуализации открывает широкие возможности для получения целостного и наглядного представления архитектуры проекта. Научная новизна работы заключается в адаптации формальных методов анализа к специфике динамической природы JavaScript и в разработке подходов, позволяющих интегрировать визуальные и количественные методы в инструменты поддержки разработки, внедрение в исследование графовой БД [1], [2], [3], [4], [5], а также [6], [7].

Результаты планируемого исследования могут быть полезны как для научного сообщества, так и для практикующих разработчиков. Они позволят по-новому взглянуть на архитектуру приложений, выявлять потенциальные проблемы на ранних этапах, повышать прозрачность системы и, в конечном счёте, улучшать качество программного обеспечения.

Ключевые слова: лексический анализ, регулярные выражения, синтаксический анализ, ES-модули, JavaScript ECMAScript, граф, визуализация графа, граф зависимостей, построение графовой БД.

METHODS FOR ANALYSING AND VISUALISING ES MODULE DEPENDENCIES IN JAVASCRIPT APPLICATIONS USING GRAPH DATABASES

Research article

Astahova I.F.^{1,*}, Kryuchkov N.S.², Borisenkov D.V.³³ORCID : 0009-0005-8770-1081;^{1,2,3} Voronezh State University, Voronezh, Russian Federation

* Corresponding author (astachova[at]list.ru)

Suggested: 05.07.2026; Accepted: 24.09.2026; Published: 16.10.2026

Abstract

The analysis of dependencies between software modules in applications is becoming an increasingly important stage in the development of modern applications with each passing year. The creation of a general mathematical model describing dependency analysis at a fundamental level is one of the tasks that could help accelerate the development of new dependency analysis tools for a wide variety of module systems and programming languages using a graph database. The presented article attempts to systematise existing approaches to the analysis of module dependencies in JavaScript, review technical solutions and outline the methodological foundations for future research. It is demonstrated that the use of graph-based representation and visualisation opens up extensive possibilities for obtaining a comprehensive and clear view of the project's architecture. The scientific novelty of this work lies in the adaptation of formal analysis methods to the specific dynamic nature of JavaScript and in the development of approaches that make it possible to integrate visual and quantitative methods into development support tools, and the incorporation of graph databases into the research [1], [2], [3], [4], [5], as well as [6], [7].

The results of the planned research may be useful both to the academic community and to practicing developers. They will provide a fresh perspective on application architecture, enable potential problems to be identified at an early stage, increase system transparency and, ultimately, improve the quality of software.

Keywords: lexical analysis, regular expressions, syntactic analysis, ES modules, JavaScript ECMAScript, graph, graph visualisation, dependency graph, graph database construction.

Введение

Графовые базы данных — класс баз данных, в которых данные представлены в виде сущностей и отношений между ними. За счёт того, что структура хранения напрямую отражает структуру связей предметной области, такие

базы данных часто обеспечивают более высокую гибкость и производительность при обработке сильно связанных данных по сравнению с реляционными системами. Узлы, или вершины (nodes), — это экземпляры сущностей; рёбра, или связи (edges), — отношения между ними; свойства (properties) содержат описательную информацию, связанную с узлами и рёбрами.

К преимуществам графовых баз данных обычно относят:

- гибкую схему данных, не требующую заранее фиксированной структуры таблиц;
- наглядное представление связей между сущностями, естественное для задач анализа зависимостей;
- высокую скорость выполнения запросов, связанных с обходом связей, — в отличие от реляционных СУБД, где аналогичные операции требуют дорогостоящих операций соединения таблиц.

Среди ограничений можно отметить:

- снижение производительности при выполнении запросов, требующих обхода очень большого числа связей, на графах с миллионами узлов;
- сравнительно небольшое, по сравнению с реляционными СУБД, сообщество пользователей у ряда графовых платформ, что может затруднять получение технической поддержки при решении нетиповых задач [1], [2].

Наиболее распространённой в настоящее время графовой СУБД является Neo4j — платформа для нативного хранения и обработки графовых структур, написанная на Java; название Neo4j иногда расшифровывают как «Network Exploration and Optimization for Java» («исследование и оптимизация сетей для Java»). Neo4j поддерживает свойства ACID (атомарность, согласованность, изолированность и долговечность транзакций), что делает её пригодной в том числе для систем, предъявляющих требования к транзакционной целостности данных [1].

Cypher — декларативный язык запросов, оптимизированный для работы с графовыми базами данных. Он предоставляет удобный синтаксис для описания узлов, рёбер и их атрибутов, а также для формулирования запросов на поиск путей, подграфов и агрегированных характеристик графа [3].

Зачастую разработчики, особенно при работе с крупными JavaScript-проектами, сталкиваются с проблемой усложнения архитектуры приложения из-за разрастания количества модулей и усложнения связей между ними. Отсутствие наглядного представления о зависимостях может привести к дублированию кода, созданию циклических зависимостей и затруднённому сопровождению проекта.

В связи с этим возникла необходимость в разработке подходов и инструментов, позволяющих анализировать и визуализировать структуру модульных зависимостей в современных JavaScript-приложениях. Это особенно актуально для приложений, использующих стандарт ES-модулей, активно применяемый в экосистеме ECMAScript [7].

Основной целью данной статьи является обзор существующих решений, выявление их недостатков и формулировка методологических основ дальнейшего исследования по анализу и визуализации модульных зависимостей. В частности, рассматриваются технологии, применяемые при создании графов зависимостей, а также инструменты, основанные на статическом анализе кода, с фокусом на библиотеках и методах, применимых к ES-модулям, и на возможностях их представления средствами графовых баз данных.

1.1. Актуальность и постановка проблемы

С ростом масштабов клиентских и серверных JavaScript-приложений возросла потребность в средствах, позволяющих отслеживать архитектурную сложность и выявлять потенциальные проблемы. Одним из ключевых аспектов при разработке современных веб-приложений становится организация модульных зависимостей, от которой напрямую зависит поддерживаемость и масштабируемость системы. Анализ модульных связей помогает упрощать рефакторинг, снижать технический долг, минимизировать риски, связанные с безопасностью, и повышать эффективность процессов разработки. Однако существующие инструменты либо ограничены в функциональности, либо не обладают достаточной гибкостью для анализа сложных, динамически формируемых связей. Это делает задачу комплексного анализа зависимостей особенно актуальной в контексте быстроразвивающейся JavaScript-экосистемы, а специфика самой задачи — большое число слабо структурированных связей между модулями — естественным образом подводит к использованию графовой модели данных и графовых СУБД как средства хранения и анализа таких зависимостей.

Обзор существующих подходов

Современная экосистема JavaScript предоставляет разработчикам ряд инструментов для анализа и визуализации модульных зависимостей. Наиболее активно применяются утилиты, интегрируемые в сборщики, такие как Webpack и Rollup. К примеру, Webpack Bundle Analyzer [12] позволяет разработчику визуально оценить структуру собранного бандла, отображая дерево модулей и их размеры, что, впрочем, больше ориентировано на оптимизацию размера конечного продукта, чем на архитектурный анализ. Аналогичным функционалом обладает Rollup Plugin Visualizer [9], который предоставляет графическое представление модулей, их вложенности и взаимосвязей на этапе сборки проекта.

Инструмент Dependency Cruiser [8] предоставляет возможность статического анализа и построения графов импортов с учётом определённых правил и ограничений. Его особенность заключается в том, что он позволяет конфигурировать набор допустимых и недопустимых связей, что делает его полезным при соблюдении архитектурных соглашений в крупных проектах. Однако при этом он не всегда справляется с нетривиальными случаями, такими как алиасные импорты, использование нестандартных разрешений путей или динамические конструкции импорта, разбор которых требует применения специализированных синтаксических анализаторов [13], [14], [15], [16].

Отдельного внимания заслуживает Madge [10] — утилита командной строки, предназначенная для построения графа зависимостей и выявления циклов. Она может интегрироваться в CI/CD процессы и предоставлять сведения о том, где в проекте возникают нежелательные связи. Однако её визуализации носят скорее вспомогательный характер и не рассчитаны на глубокий архитектурный анализ.

Общей чертой большинства перечисленных решений является то, что они ориентированы преимущественно на технических специалистов и решают прикладные задачи — уменьшение размера сборки, выявление циклов или соблюдение архитектурных правил. При этом отсутствует целостное представление об архитектуре системы в терминах графовой теории или метрик качества кода. Кроме того, эти инструменты слабо адаптированы к научному анализу и не позволяют масштабировать визуализацию на большие проекты с сотнями и тысячами модулей без потери наглядности.

2.1. Научные подходы к графовому анализу программной архитектуры

Научное сообщество активно исследует архитектуру программного обеспечения, однако подавляющее большинство таких исследований сосредоточено на языках с более выраженной статической типизацией, таких как Java и C++. Это обусловлено тем, что такие языки предоставляют больше возможностей для глубокого анализа кода на этапе компиляции. Тем не менее полученные в этих работах подходы могут быть адаптированы и для динамических языков, включая JavaScript.

В частности, в литературе активно изучается применение графов для описания архитектур программ [5], [6]. Работы в этом направлении предлагают представлять программную систему в виде направленного графа, где узлы обозначают модули, классы или функции, а рёбра — зависимости, вызовы или передачи управления. Такой подход позволяет использовать известные из теории графов метрики: степень вершины, центральность, связность, плотность и другие. Это даёт возможность формализованно оценивать сложность архитектуры, выявлять перегруженные компоненты или потенциальные узкие места.

Некоторые исследования рассматривают возможности визуального анализа программной архитектуры как средства поддержки принятия решений при рефакторинге или при переходе к новым архитектурным стилям. Используются различные техники визуализации, в том числе force-directed layout, иерархические (Sugiyama-style) схемы, а также методы кластеризации и автоматической группировки модулей по архитектурным слоям.

Стоит отметить, что в контексте JavaScript и особенно ES-модулей научных публикаций пока немного: большинство работ по статическому анализу JavaScript сосредоточено на построении графов вызовов для решения задач безопасности и рефакторинга [4], тогда как задачи архитектурной визуализации модульных зависимостей остаются менее исследованными. Это объясняется относительной новизной стандарта и высокой степенью динамичности JavaScript как языка. Однако именно эта сфера открывает широкое поле для научных исследований, направленных на адаптацию классических методов архитектурного анализа к особенностям современных веб-технологий. Отсутствие типовой модели модульной архитектуры и нестабильность зависимостей в JavaScript требуют разработки специализированных подходов, способных учитывать динамическую природу кода и гибкую структуру проектов.

Методологические аспекты исследования

Методология, лежащая в основе предлагаемого исследования, ориентирована на построение формальной модели зависимостей между модулями в JavaScript-приложениях, использующих стандарт ES-модулей, с последующей визуализацией и анализом этой структуры. Важной особенностью выбранного подхода является сочетание технических решений — таких как парсинг исходного кода и генерация графов, — с элементами научного анализа, в частности, применением графовых метрик и принципов визуальной аналитики.

На первом этапе планируется сформировать корпус открытых JavaScript-проектов, размещённых на платформе GitHub, удовлетворяющих следующим критериям: активная разработка, использование ES-модулей в структуре проекта, достаточный масштаб кода (от 100 файлов и более). Для сбора и предварительной фильтрации данных предполагается использовать GitHub API, а также утилиты командной строки, позволяющие производить автоматическую загрузку и очистку репозитория.

Следующим ключевым шагом является парсинг структуры исходного кода. Для этого предполагается использовать инструментальные средства, способные обрабатывать синтаксис ECMAScript, например Babel и Acorn. В процессе анализа будут выявлены связи между файлами, выраженные через ключевые конструкции `import` и `export`. Особое внимание будет уделено корректной интерпретации относительных и абсолютных путей, разрешению алиасов и обработке потенциально динамических импортов. На этом этапе данные о зависимостях будут трансформированы в представление, пригодное для построения графов — в частности, в виде списков смежности или пар связанных узлов, что соответствует общепринятым способам представления графовых структур в задачах компиляции и статического анализа [17], [18], [19].

Третьим этапом станет построение ориентированных графов, где каждая вершина будет соответствовать отдельному модулю, а каждое направленное ребро — зависимости между ними. Полученные графы могут содержать сотни и тысячи узлов, поэтому важной задачей становится не только их формальное описание, но и поиск подходящих алгоритмов визуального представления. Предпочтение планируется отдать интерактивным визуализациям, реализуемым средствами JavaScript-библиотек D3.js, Sigma.js или Cytoscape.js. Эти библиотеки позволяют реализовать гибкую навигацию по графу, масштабирование, группировку узлов, а также отображение дополнительных свойств модулей, таких как размер, частота использования или количество входящих/исходящих связей.

Дополнительно предполагается использование количественного анализа графов, включающего в себя вычисление таких метрик, как средняя степень вершины, плотность графа, диаметр, коэффициент кластеризации и центральность. Эти метрики позволят выявлять узлы с высокой степенью зависимости, потенциальные архитектурные «бутылочные горлышки», а также зоны избыточной связанности, которые могут привести к усложнению сопровождения системы. Отдельный интерес представляет задача выявления циклических зависимостей, для которой в теории графов существуют эффективные алгоритмы поиска всех элементарных циклов ориентированного графа [20], [21]; их реализация средствами языка запросов Cypher рассматривается в разделе 5.

Завершающим этапом станет интерпретация полученных данных и сопоставление количественных характеристик архитектуры с качественными признаками кода. Это позволит выработать гипотезы о корреляции между структурой графа и архитектурной зрелостью проекта, а также наметить пути автоматизированной диагностики и возможного рефакторинга на основе выявленных паттернов.

Таким образом, методология исследования объединяет технические, визуальные и аналитические подходы, создавая основу для комплексного анализа модульных зависимостей в современных JavaScript-приложениях.

Ожидаемые результаты и научная новизна

В рамках исследования планируется сформировать набор практических рекомендаций и инструментальных решений для анализа модульной структуры JavaScript-приложений. Ожидается, что результаты позволят более эффективно выявлять узлы архитектурной сложности, предложат новые способы визуального представления и будут способствовать развитию подходов к сопровождению и рефакторингу кода.

В отличие от Dependency Cruiser [8] и Madge [10], которые хранят граф зависимостей во внутреннем представлении конкретного инструмента и предоставляют ограниченный, заранее определённый набор проверок, предлагаемый подход рассматривает граф зависимостей как самостоятельный объект хранения и анализа в графовой базе данных. Это даёт возможность формулировать произвольные аналитические запросы на языке Cypher — в том числе такие, которые не предусмотрены авторами исходных утилит, — сопоставлять графы разных версий проекта во времени, а также применять к ним стандартные алгоритмы теории графов и накопленный инструментарий графовой аналитики, не реализуя их заново под конкретную задачу. Особый интерес представляет возможность обобщения методики на другие языки, поддерживающие модульность, и интеграция полученных решений в процесс автоматизированного анализа качества программного обеспечения.

Представление графа зависимостей в графовой базе данных: иллюстративный пример

Чтобы показать, как описанная выше методология может быть реализована на практике, рассмотрим иллюстративный пример небольшого гипотетического JavaScript-приложения, состоящего из девяти ES-модулей: точки входа `app.js`, модуля маршрутизации `router.js`, модуля состояния `store.js`, двух модулей представления `views/home.js` и `views/profile.js`, модуля обращения к API `api/users.js`, модуля низкоуровневых HTTP-запросов `api/http.js`, вспомогательного модуля `utils/format.js` и файла конфигурации `config.js`. Связи импорта между этими модулями образуют граф зависимостей, показанный на рисунке 1.

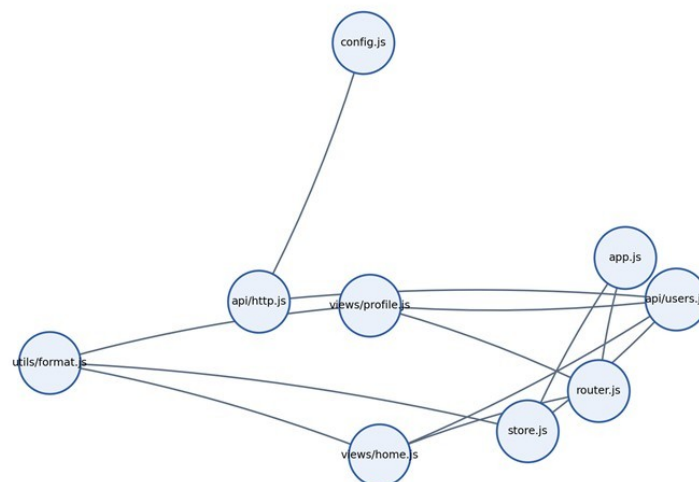


Рисунок 1 - Граф зависимостей ES-модулей гипотетического приложения (иллюстративный пример)

DOI: <https://doi.org/10.60797/IRJ.2026.172.77.1>

В модели графовой базы данных каждый модуль естественно представляется узлом с меткой `Module` и свойством `path`, а каждая связь `import` — направленным ребром `DEPENDS_ON`. Загрузка такого графа в Neo4j может быть выполнена, например, следующим набором инструкций на языке Cypher:

```
CREATE (app:Module {path: "app.js"})
CREATE (router:Module {path: "router.js"})
CREATE (store:Module {path: "store.js"})
CREATE (home:Module {path: "views/home.js"})
CREATE (profile:Module {path: "views/profile.js"})
```

```
CREATE (users:Module {path: "api/users.js"})
CREATE (http:Module {path: "api/http.js"})
CREATE (fmt:Module {path: "utils/format.js"})
CREATE (cfg:Module {path: "config.js"})
CREATE (app)-[:DEPENDS_ON]->(router)
CREATE (app)-[:DEPENDS_ON]->(store)
CREATE (router)-[:DEPENDS_ON]->(home)
CREATE (router)-[:DEPENDS_ON]->(profile)
CREATE (home)-[:DEPENDS_ON]->(users)
CREATE (profile)-[:DEPENDS_ON]->(users)
CREATE (users)-[:DEPENDS_ON]->(http)
CREATE (http)-[:DEPENDS_ON]->(cfg)
CREATE (store)-[:DEPENDS_ON]->(users)
CREATE (store)-[:DEPENDS_ON]->(fmt)
CREATE (home)-[:DEPENDS_ON]->(fmt)
CREATE (profile)-[:DEPENDS_ON]->(fmt)
```

После загрузки графа становятся доступны аналитические запросы, которые в существующих инструментах статического анализа (Dependency Cruiser, Madge) либо отсутствуют, либо требуют написания дополнительного кода. Так, поиск модулей с наибольшим количеством входящих зависимостей — потенциальных «узких мест» архитектуры — выполняется запросом:

```
MATCH (m:Module)-[:DEPENDS_ON]-(dependent)
RETURN m.path AS module, count(dependent) AS incomingDependencies
ORDER BY incomingDependencies DESC
LIMIT 5
```

Поиск циклических зависимостей — одна из типовых задач архитектурного анализа — формулируется как поиск замкнутого пути в графе:

```
MATCH path = (m:Module)-[:DEPENDS_ON*2..6]->(m)
RETURN [n IN nodes(path) | n.path] AS cycle
```

а вычисление средней степени вершины графа, характеризующей его общую связность, — агрегирующим запросом:

```
MATCH (m:Module)
OPTIONAL MATCH (m)-[:DEPENDS_ON]->()
WITH m, count(r) AS outDegree
RETURN avg(outDegree) AS averageOutDegree
```

Приведённый пример носит иллюстративный характер и не заменяет полноценной апробации методики на корпусе реальных проектов, которая является следующим этапом исследования (раздел 5). Тем не менее он показывает, что граф зависимостей ES-модулей может быть напрямую отображён в модель свойств графовой базы данных без потери информации и что типовые задачи архитектурного анализа — поиск циклов, оценка связности и выявление наиболее зависимых модулей — можно сформулировать компактными запросами на языке Cypher, не прибегая к отдельной реализации соответствующих алгоритмов.

Заключение

В условиях постоянного роста сложности JavaScript-приложений задача анализа и визуализации модульных зависимостей приобретает особую актуальность. ES-модули, являясь основой современной модульной системы в экосистеме ECMAScript, создают структурные связи, которые требуют осмысленного подхода к организации и контролю. Существующие инструменты статического анализа — Webpack Bundle Analyzer, Rollup Plugin Visualizer, Dependency Cruiser, Madge — решают отдельные прикладные задачи, но не предоставляют единого представления архитектуры в терминах графовой теории и не позволяют формулировать произвольные аналитические запросы к графу зависимостей.

В настоящей статье предложен подход, устраняющий этот пробел за счёт представления графа зависимостей ES-модулей в графовой базе данных Neo4j. Систематизированы существующие инструменты и научные работы по графовому анализу программной архитектуры, сформулированы методологические основы дальнейшего исследования — от сбора корпуса открытых проектов и парсинга исходного кода до вычисления графовых метрик, — а на

иллюстративном примере показано, как граф зависимостей загружается в графовую базу данных и как типовые задачи архитектурного анализа (поиск циклов, оценка связности, выявление наиболее зависимых модулей) решаются запросами на языке Cypher.

Дальнейшие шаги исследования будут направлены на реализацию программного прототипа, сбор эмпирических данных на основе анализа открытых репозиторий с GitHub и оценку предложенного подхода на корпусе реальных проектов в сопоставлении с существующими инструментами. Полученные результаты, как ожидается, станут вкладом в развитие области анализа программной архитектуры на языке JavaScript.

Благодарности

Авторы выражают благодарность рецензенту за проделанную работу.

Конфликт интересов

Не указан.

Рецензия

Все статьи проходят рецензирование. Но рецензент или автор статьи предпочли не публиковать рецензию к этой статье в открытом доступе. Рецензия может быть предоставлена компетентным органам по запросу.

Acknowledgement

The authors express their gratitude to the reviewer for their work.

Conflict of Interest

None declared.

Review

All articles are peer-reviewed. But the reviewer or the author of the article chose not to publish a review of this article in the public domain. The review can be provided to the competent authorities upon request.

Список литературы / References

1. Ковалев М.С. Оптимизация хранения и обработки графовых данных в гибридных системах: комбинация NoSQL и графовых СУБД для масштабируемых решений / М.С. Ковалев // Молодой учёный. — 2025. — № 20(571). — С. 30–32.
2. Angles R. Survey of graph database models / R. Angles, C. Gutierrez // ACM Computing Surveys (CSUR). — 2008. — Vol. 40, № 1. — Art. 1.
3. Arora R. Using dependency graphs to support collaboration over GitHub: the Neo4j graph database approach / R. Arora, S. Goel, R.K. Mittal // 2016 Ninth International Conference on Contemporary Computing (IC3). — IEEE, 2016. — P. 1–7.
4. Antal G. Static JavaScript call graphs: a comparative study / G. Antal, P. Hegedűs, Z. Tóth [et al.] // 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). — 2018. — P. 177–186.
5. Lanza M. Understanding software evolution using a combination of software visualization and software metrics / M. Lanza, S. Ducasse // Proc. of LMO'02. — Paris : Lavoisier, 2002. — P. 135–149.
6. Zimmermann T. Mining version histories to guide software changes / T. Zimmermann [и др.] // IEEE Transactions on Software Engineering. — 2005. — Vol. 31, № 6. — P. 429–445.
7. ECMAScript Language Specification. — URL: <https://262.ecma-international.org> (accessed: 22.08.2026).
8. Dependency Cruiser Documentation. — URL: <https://github.com/sverweij/dependency-cruiser> (accessed: 22.08.2026).
9. Rollup Plugin Visualizer. — URL: <https://www.npmjs.com/package/rollup-plugin-visualizer> (accessed: 22.08.2026).
10. Madge — Create graphs from module dependencies. — URL: <https://github.com/pahen/madge> (accessed: 22.08.2026).
11. Флэнаган Д. JavaScript : подробное руководство / Д. Флэнаган. — 6-е изд. — СПб. : Символ-Плюс, 2012. — 1080 с.
12. Webpack. — URL: <https://webpack.js.org> (accessed: 22.08.2026).
13. Detect circular dependencies in modules compiled with Webpack. — URL: <https://github.com/aackerman/circular-dependency-plugin> (accessed: 22.08.2026).
14. ECMAScript parsing infrastructure for multipurpose analysis. — URL: <https://github.com/jquery/esprima> (accessed: 22.08.2026).
15. Esprima-compatible JavaScript parser. — URL: <https://github.com/eslint/esprex> (accessed: 22.08.2026).
16. Escope, an ECMAScript scope analyzer. — URL: <https://github.com/eslint/esprex> (accessed: 22.08.2026).
17. Ахо А.В. Компиляторы: принципы, технологии и инструментарий / А.В. Ахо [и др.]. — 2-е изд. — Москва : Вильямс, 2008. — 1184 с.
18. AST explorer. — URL: <https://astexplorer.net> (accessed: 22.08.2026).
19. The ESTree Spec. — URL: <https://github.com/estree/estree> (accessed: 22.08.2026).
20. Johnson D.B. Finding all the elementary circuits of a directed graph / D.B. Johnson // SIAM Journal on Computing. — 1975. — Vol. 4, № 1. — P. 77–84.
21. Meyer F. Search all elementary cycles in a given directed graph. — URL: <http://normalisiert.de/code/java/elementaryCycles.zip> (accessed: 22.08.2026).

Список литературы на английском языке / References in English

1. Kovalev M.S. Optimizatsiya khraneniya i obrabotki grafovyykh dannyykh v gibridnykh sistemakh: kombinatsiya NoSQL i grafovyykh SUBD dlya masshtabiruemykh resheniy [Optimization of storage and processing of graph data in hybrid systems: a combination of NoSQL and graph DBMS for scalable solutions] / M.S. Kovalev // Molodoy uchyonyy [Young Scientist]. — 2025. — № 20(571). — P. 30–32. [in Russian]



2. Angles R. Survey of graph database models / R. Angles, C. Gutierrez // ACM Computing Surveys (CSUR). — 2008. — Vol. 40, № 1. — Art. 1.
3. Arora R. Using dependency graphs to support collaboration over GitHub: the Neo4j graph database approach / R. Arora, S. Goel, R.K. Mittal // 2016 Ninth International Conference on Contemporary Computing (IC3). — IEEE, 2016. — P. 1–7.
4. Antal G. Static JavaScript call graphs: a comparative study / G. Antal, P. Hegedűs, Z. Tóth [et al.] // 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). — 2018. — P. 177–186.
5. Lanza M. Understanding software evolution using a combination of software visualization and software metrics / M. Lanza, S. Ducasse // Proc. of LMO'02. — Paris : Lavoisier, 2002. — P. 135–149.
6. Zimmermann T. Mining version histories to guide software changes / T. Zimmermann [и др.] // IEEE Transactions on Software Engineering. — 2005. — Vol. 31, № 6. — P. 429–445.
7. ECMAScript Language Specification. — URL: <https://262.ecma-international.org> (accessed: 22.08.2026).
8. Dependency Cruiser Documentation. — URL: <https://github.com/sverweij/dependency-cruiser> (accessed: 22.08.2026).
9. Rollup Plugin Visualizer. — URL: <https://www.npmjs.com/package/rollup-plugin-visualizer> (accessed: 22.08.2026).
10. Madge — Create graphs from module dependencies. — URL: <https://github.com/pahen/madge> (accessed: 22.08.2026).
11. Flanagan D. JavaScript: podrobnoye rukovodstvo [JavaScript: The Definitive Guide] / D. Flanagan. — 6th ed. — St. Petersburg : Simvol-Plyus, 2012. — 1080 p. [in Russian]
12. Webpack. — URL: <https://webpack.js.org> (accessed: 22.08.2026).
13. Detect circular dependencies in modules compiled with Webpack. — URL: <https://github.com/aackerman/circular-dependency-plugin> (accessed: 22.08.2026).
14. ECMAScript parsing infrastructure for multipurpose analysis. — URL: <https://github.com/jquery/esprima> (accessed: 22.08.2026).
15. Esprima-compatible JavaScript parser. — URL: <https://github.com/eslint/esprez> (accessed: 22.08.2026).
16. Escope, an ECMAScript scope analyzer. — URL: <https://github.com/estools/escape> (accessed: 22.08.2026).
17. Aho A.V. Kompilyatory: printsipy, tekhnologii i instrumentariy [Compilers: Principles, Techniques, and Tools] / A.V. Aho [et al.]. — 2nd ed. — Moscow : Williams, 2008. — 1184 p. [in Russian]
18. AST explorer. — URL: <https://astexplorer.net> (accessed: 22.08.2026).
19. The ESTree Spec. — URL: <https://github.com/estree/estree> (accessed: 22.08.2026).
20. Johnson D.B. Finding all the elementary circuits of a directed graph / D.B. Johnson // SIAM Journal on Computing. — 1975. — Vol. 4, № 1. — P. 77–84.
21. Meyer F. Search all elementary cycles in a given directed graph. — URL: <http://normalisiert.de/code/java/elementaryCycles.zip> (accessed: 22.08.2026).