УДК 004

Visual Graph Library: архитектура и возможности Java-библиотеки для визуализации графов

Золотухин Т.А. (Институт систем информатики им. А.П. Ершова СО РАН)

В статье описывается архитектура и возможности Visual Graph Library — библиотеки для хранения, укладки и визуализации иерархических атрибутированных графов с портами. Рассматриваются ключевые модули системы и интеграция с платформой Java. Приводится анализ существующих библиотек и приложений для работы с графами.

Ключевые слова: *иерархические атрибутированные графы с портами, графовые представления потоковых программ, визуализация графов, системы визуализации графов.*

1. Введение

Визуализация графов является фундаментальным инструментом анализа сложных программных и технических систем. Представление синтаксических деревьев, управляющих графов и структур вызовов в наглядной форме критически важно для верификации, оптимизации и проектирования современного ПО [1, 3, 7].

Однако инструменты общего назначения имеют архитектурные ограничения при работе со специфическими структурами — **иерархическими атрибутированными графами, содержащими порты**. Базовый функционал таких библиотек редко позволяет задать жесткую привязку дуг к конкретным точкам на границе вершин, а также корректно обработать их многоуровневую вложенность. Адаптация подобных решений под требования инженерного ПО неизбежно ведет к усложнению архитектуры приложения и реализации избыточных программных надстроек. Кроме того, большинство существующих инструментов не предоставляет гибких средств программного управления сложными графовыми структурами напрямую в рамках Java-экосистемы.

Библиотека **Visual Graph Library (VGL)** создана на основе архитектурного ядра системы «Visual Graph», разработанной в рамках исследования методов укладки и навигации в иерархических атрибутированных графах [4, 26]. В ходе развития проекта архитектурные решения были переосмыслены и унифицированы, а графическое ядро — изолировано от прикладной логики. Это позволило трансформировать специализированный инструментарий

в универсальную библиотеку для Java-экосистемы, предлагающую открытую модульную архитектуру для свободного использования.

Цель настоящей статьи — представить архитектуру VGL, описать модель хранения графовых структур, реализованные алгоритмы укладки и средства визуализации. Также в работе проводится обзор существующих аналогов и обосновывается необходимость разработки библиотеки VGL.

2. Архитектура библиотеки

Архитектура библиотеки построена на принципах модульности и разделения ответственности. В основу системы положен паттерн MVC (Model-View-Controller) [17], позволяя изолировать данные от механизмов их обработки и отображения. Структурно библиотеку можно разделить на следующие модули:

- Модуль хранения графов является обязательным ядром системы. Оно инкапсулирует всю логику работы с графовыми структурами, их топологией, иерархией, атрибутами и портами. Любое взаимодействие с библиотекой начинается с создания экземпляра хранилища, так как все остальные модули используют его в качестве единственного источника данных.
- Модуль импорта и экспорта графов обеспечивает преобразование внешних форматов во внутренние объекты модуля хранения и обратно.
- Модуль укладки графов предоставляет набор алгоритмов для автоматического вычисления пространственных характеристик графа. Укладчики модифицируют исключительно геометрические атрибуты элементов в хранилище, не затрагивая их семантику или топологию.
- Модуль визуализации отвечает за графическую интерпретацию графовых структур, находящихся в модули хранения графов, а также обработку действий пользователя (масштабирование, навигация, выделение элементов).

Подобная декомпозиция позволяет использовать библиотеку как «конструктор», адаптируя её под конкретные задачи без избыточности кода. Такой подход минимизирует связанность компонентов и упрощает внедрение библиотеки в существующие сторонние проекты. Детальный обзор каждого модуля и особенности их программной реализации будут приведены в последующих главах.

3. Хранение графов

Эффективная организация хранения графовых структур является фундаментом для их дальнейшей обработки и визуализации. Простых графовых структур, где вершины изображаются точками, а дуги прямыми, зачастую не хватает для прикладных инженерных задач, где элементы обладают сложной семантикой и иерархической структурой. Для решения таких задач в VGL было решено использовать **иерархические атрибутированные графы с портами**. Определение такой структуры строится на объединении трех понятий из теории графов:

- **Иерархический граф** определяется как кортеж (G, T), в котором G – это граф произвольного типа, а T – дерево вложенности, вершины которого соответствуют элементам некоторой иерархии в G, дуги же отражают отношение их непосредственной вложенности [3].
- **Граф с портами** определяется как кортеж (V, E, P, π), в котором функция $\pi: P \to V$, сопоставляет каждому порту единственную вершину, которой он принадлежит. Множество дуг E определяется как подмножество объединения трех типов декартовых произведений: $E \subseteq (V \times V) \cup (P \times V) \cup (V \times P)$.
- **Атрибутированный граф с портами** определяется как кортеж $(V, E, P, A, \pi, \alpha_V, \alpha_E, \alpha_P)$, в котором функции $\alpha_V: V \to A, \alpha_E: E \to A, \alpha_P: P \to A$ расширяют структурную модель графа с портами $G = (V, E, P, \pi)$. Эти функции сопоставляют вершинам, дугам и портам конечные наборы атрибутов из множества A. Элементами множества A являются пары вида $(name, value)$, где $name$ – имя атрибута, а $value$ – его значение.

Для реализации описанной выше теоретической концепции была спроектирована программная модель данных, основу которой составляют следующие сущности:

- Атрибут – типизированная запись вида «ключ – значение», расширяющая семантику любого элемента графа. В библиотеке атрибуты разделяются на пользовательские, описывающие данные исходной задачи, и системные, создаваемые библиотекой для управления визуализацией и состоянием (например, параметры шрифта, толщина границ или флаг выделения элемента). Для значений поддерживаются следующие типы данных: строковый, числовой (целые и вещественные числа), логический, уникальные идентификаторы, а также сложные структуры в формате JSON. Атрибуты позволяют гибко настраивать как визуальное представление, так и прикладную логику без внесения изменений в программную модель данных.

Несмотря на то что атрибут является элементом графа, он является исключением и не может быть расширен за счет других атрибутов.

- Графовая модель – логический контейнер и корневой идентификатор. В рамках физического хранения, идентификатор модели является обязательным ключом для доступа к любому элементу. Графовая модель является элементом графа и может быть расширена атрибутами, которые хранят некоторый набор метаданных для обмена состоянием между модулями системы.
- Вершина — элементарная единица топологии и основной объект для алгоритмов обработки, является элементом, а значит обладает набором атрибутов, которые могут хранить например его геометрические параметры, такие как координаты и размеры.
- Фрагмент — вершина, содержащая вложенный граф. Это программная реализация, позволяет рассматривать фрагмент как «черный ящик» для вышестоящего уровня иерархии.
- Порт — специализированная вершина, помеченная соответствующим атрибутом. Такая унификация позволяет использовать для портов стандартные механизмы поиска и навигации, принятые для обычных вершин.
- Фиктивный порт — вспомогательный порт, помеченный соответствующим атрибутом. Данный элемент создается библиотекой автоматически, когда дуга должна связать вершины, находящиеся в разных фрагментах.
- Вершина с портами — специфический вид фрагмента, который не содержит других элементов, кроме портов.
- Дуга — элемент, связывающий две вершины, находящиеся в одном фрагменте. Информация о том, к каким конкретно портам привязана дуга, хранится в ее атрибутах. Это позволяет библиотеке, делегировать расчеты конкретных точек входа/выхода алгоритмам укладки и отрисовки.
- Цепочка — логическая абстракция, представляющая собой последовательность сегментированных дуг и фиктивных портов. Она описывает сквозной путь между вершинами, которые находятся в разных фрагментах. Благодаря декомпозиции связи на локальные сегменты, алгоритмы обработки (например, укладка) могут работать с каждым подграфом изолированно, не нарушая границ инкапсуляции.

Центральным компонентом модуля хранения является хранилище графовых моделей. Данный компонент инкапсулирует логику работы с элементами и может рассматриваться как

модель в контексте классического паттерна MVC. Хранилище служит единой точкой доступа для всех потребителей — от алгоритмов укладки до модуля визуализации — и может передаваться между ними как самостоятельная единица. Техническая реализация позволяет использовать для хранения элементов как оперативную память, так и дисковое пространство, в зависимости от задач пользователя. Основные возможности данного компонента:

- Один экземпляр хранилища может управлять множеством независимых графовых моделей. Разделение данных происходит по уникальному идентификатору модели, что позволяет работать с ними параллельно, сохраняя полную изоляцию данных в рамках одного объекта.
- Базовый набор операций для создания, поиска и модификации элементов графа дополнен внутренней логикой контроля целостности. Это гарантирует, что все изменения структуры проходят с сохранением корректного состояния связей и атрибутов в любой момент времени.
- Встроенный механизм обхода графовой модели в глубину с поддержкой настраиваемых обработчиков позволяет внешним алгоритмам работать с иерархией элементов и концентрироваться на логике обработки данных, не дублируя код рекурсивного обхода в каждом модуле.
- Встроенные средства копирования данных обеспечивают миграцию графовых моделей или их фрагментов между различными экземплярами хранилищ (например, из оперативной памяти в дисковое пространство). Процесс гарантирует полное сохранение структурной целостности и атрибутивной семантики переносимых данных.

Таким образом, разработанный модуль определяет набор сущностей с расширяемой семантикой и механизм хранения, отделяющий данные от логики их обработки. Это позволяет алгоритмам укладки, навигации и отрисовки взаимодействовать с графовыми структурами через единый интерфейс, абстрагированный от способа их физического размещения.

4. Визуализация графов

Если модуль хранения определяет программную модель данных и способы управления для графовых структур, то модуль визуализации отвечает за их графическую интерпретацию. В рамках архитектуры VGL визуализация рассматривается не как жестко детерминированный процесс отрисовки элементов, а как динамическое преобразование набора атрибутов, закрепленных за элементами, в визуальные примитивы. Такой подход позволяет полностью отделить логику представления от структурных данных: модуль визуализации выступает внешним потребителем хранилища графовых моделей, восстанавливая визуальный облик всей

модели или её отдельных частей на основе системных и пользовательских атрибутов элементов, входящих в них.

Процесс отрисовки опирается на иерархический обход графовой модели, описанный в предыдущей главе. Модуль визуализации последовательно анализирует каждый полученный элемент и транслирует семантику его атрибутов в параметры графического контекста, формируя облик конкретных элементов:

- Визуализация вершины начинается с расчета её геометрических размеров, динамически определяемых на основе списка выбранных пользовательских атрибутов. Использование моноширинного шрифта позволяет точно вычислить ширину и высоту текстовой области по количеству знаков и размеру кегля. С учетом внутренних отступов и толщины границ полученный текстовый блок вписывается в выбранную геометрическую форму (прямоугольник, эллипс или ромб), что и определяет итоговые габариты вершины. Результаты расчета фиксируются в системных атрибутах, позволяя либо сразу выполнить отрисовку в координатах по умолчанию, либо использовать эти данные в модуле укладки для расчета финальной позиции с последующим сохранением координат в соответствующих атрибутах.
- Визуализация порта технически реализуется аналогично вершине, при этом алгоритмам укладки рекомендуется располагать их на границах элементов, к которым они относятся. Фиктивные порты отображаются в виде малых окружностей серого цвета. Реальный порт графически полностью идентичен вершине, что позволяет через его атрибуты задавать любую форму, цвет и состав отображаемых пользовательских атрибутов.
- Визуализация дуги представляет собой отрисовку линии, стиль которой (пунктирный или сплошной), цвет и толщина задаются атрибутами. Для направленных связей (дуг) на целевом конце формируется наконечник-стрелка, отсутствующий у ненаправленных. Дуга может включать текстовую метку, расчет размеров которой аналогичен логике вершины. Координаты точек маршрутизации (изломов) задаются модулем укладки; по умолчанию соединение строится по кратчайшему пути между геометрическими центрами фигур с автоматическим расчетом точек привязки на их границах. При отрисовке цепочек, состоящих из сегментов в разных фрагментах, обеспечивается их бесшовная стыковка для создания эффекта непрерывной линии связи сквозь все уровни иерархии.
- Визуализация фрагмента принципиально отличается от вершины тем, что его итоговые размеры напрямую зависят от размеров и расположения всех входящих в

него элементов, что делает их производными от содержимого нижних уровней иерархии. Расчет текстовой области заголовка фрагмента выполняется по тем же принципам, что и для вершины, однако позиционирование данной области делегируется модулю укладки. Наиболее простой формой фрагмента является прямоугольник, хотя допустимы и другие варианты в зависимости от возможностей конкретного алгоритма укладки. Важной особенностью является то, что внешние связи с внутренними элементами должны проходить через специализированные порты (реальные или фиктивные), располагаемые на границе фрагмента. Это обеспечивает визуальную упорядоченность иерархических переходов, позволяя рассматривать фрагмент как целостный функциональный блок.

После определения правил отрисовки элементов и общей философии их расположения необходимо рассмотреть программный инструментарий, реализующий эти принципы. В системе VGL разработан набор компонентов, позволяющий использовать библиотеку как для автономной генерации изображений на серверах, так и в составе интерактивных приложений. В этом наборе каждый последующий компонент расширяет возможности предыдущего и может быть рассмотрен как представление в контексте классического паттерна. Всего их три:

- **Статический компонент** — низкоуровневое ядро отрисовки, отвечающее исключительно за трансляцию элементов в графические примитивы, не содержит логики обработки событийной модели графического интерфейса, что позволяет использовать его для генерации отчетов на сервере. Результатом работы является графическое изображение, которое может быть сохранено в файл, передано по сети или выведено на печать. Важной особенностью компонента является поддержка выборочной отрисовки области: вместо формирования всего изображения целиком, компонент может отрисовать только заданную прямоугольную область. Это позволяет существенно экономить ресурсы памяти и процессора, а также служит фундаментом для работы механизмов оптимизации в последующих компонентах.
- **Кэширующий компонент** — слой оптимизации, выступающий декоратором для статического компонента. Его использование необходимо при работе с большими изображениями, содержащими большое количество элементов, так как без этого компонента невозможно добиться плавности при перемещении видимой области или масштабировании изображения. Согласно исследованиям [12], критические задержки при отрисовки значительно затрудняют когнитивный анализ структуры данных. Для решения этой проблемы реализован механизм, который запрашивает у статического компонента отрисовку области, размер которой превышает текущие

границы видимости, и сохраняет результат в памяти. При смещении видимой области в границах кэша изображение формируется мгновенно путем копирования из памяти. Перерисовка запрашивается только в тех случаях, когда требуемая область оказывается за границами кэшированного региона. Обновление кэша вынесено в фоновый поток, что сохраняет отзывчивость интерфейса и сводит к минимуму сценарии, в которых пользователю приходится ожидать завершения отрисовки.

- **Интерактивный компонент** — графический компонент, отвечающий за управление областью видимости изображения, а также обработку действий пользователя, таких как сигналы от мыши и клавиатуры. Он использует кэширующий компонент, и в целях оптимизации в нем реализована система слоев, которые накладываются друг на друга: изображение со структурой графа находится на самом нижнем уровне, тогда как динамические объекты, такие как рамка выбора или подсветка элементов, отрисовываются поверх на прозрачных плоскостях. Это позволяет ускорить отклик интерфейса, не затрагивая основное изображение и не задействуя вычислительные ресурсы нижестоящих компонентов.

Таким образом, разработанный модуль определяет правила отрисовки на основе семантики атрибутов, что позволяет динамически интерпретировать элементы графа в графические примитивы. При этом общая философия позиционирования элементов создает базу для работы алгоритмов укладки для достижения оптимального визуального результата. Разделение программного инструментария на статический, кэширующий и интерактивный компоненты позволяет полностью изолировать процесс отрисовки от логики управления интерфейсом. Применение механизмов выборочной отрисовки области, кэширования и системы наложенных слоев обеспечивает плавную навигацию по изображениям графов с большим количеством элементов, сохраняя возможность использования библиотеки для генерации отчетов на сервере.

5. Укладка графов

Развитие алгоритмов автоматической укладки прошло путь от упрощенных математических моделей до сложных систем визуализации инженерных данных. В этой ретроспективе можно выделить три ключевых стадии:

- Исторически первой стадией была визуализация графа как системы безразмерных точек, соединенных прямыми линиями. Основной задачей здесь являлось достижение эстетической симметрии и равномерного распределения вершин, что

решалось преимущественно силовыми алгоритмами. Эти подходы детально систематизированы в профильных фундаментальных работах [3, 7]. Однако отсутствие учета физических размеров вершин и текстовых пометок у дуг, делало их малопригодными для проектирования современных систем.

- С развитием CASE-средств и языков моделирования (UML) возникла необходимость отображать вершины как прямоугольные блоки с текстом. Сложность данной задачи заключалась в том, чтобы вершины не перекрывали друг друга, а дуги не проходили поверх вершин. Наиболее эффективными как с точки зрения производительности, так и с точки зрения визуального результата стали алгоритмы, основанные на поуровневой укладке графа [9, 20].
- Дальнейшее усложнение структур привело к тому, что «плоское» визуальное представление графа становилось перенасыщенным и малопригодным для анализа. Решением стала концепция иерархических графов, основанная на принципе декомпозиции: части графа группируются во вложенные подграфы (фрагменты), что существенно снижает когнитивную нагрузку на пользователя. Этот этап ознаменовал принципиальный сдвиг в теории укладки. Если классические алгоритмы оперировали фиксированными метриками элементов, то в иерархических моделях процесс становится рекурсивным: внутренняя геометрия элементов фрагмента определяет его габариты, что, в свою очередь, напрямую влияет на всю топологию уровня выше [1, 3, 7, 19].

Подход, реализованный в библиотеке VGL, логически продолжает эту эволюцию, объединяя иерархическую декомпозицию с детальной маршрутизацией связей. Ключевым отличием является перенос точек соединения с границ вершин на специализированные интерфейсные элементы — порты. В рамках данной модели порт выступает не абстрактной точкой привязки, а самостоятельным геометрическим объектом с фиксированными размерами. Это накладывает дополнительные ограничения на алгоритмы укладки: траектории должны рассчитываться с учетом не только вложенности структур, но и габаритов самих портов для обеспечения корректных углов примыкания.

Модуль укладки графов реализован как совокупность автономных вычислительных компонентов, логика которых полностью изолирована от других модулей. Архитектурно эти компоненты выступают функциональными аналогами методов контроллера в парадигме MVC. Каждый компонент инкапсулирует конкретный алгоритм размещения и взаимодействует напрямую с хранилищем графовых моделей, принимая внешние параметры для настройки алгоритма и определения границ вычислений — от обработки всей модели до

укладки конкретного фрагмента. Несмотря на свободу реализации внутренних алгоритмов, все они объединены общей философией, которая базируется на следующих принципах:

- Алгоритм укладки должен строго следовать правилам интерпретации атрибутов в геометрические примитивы и их пространственное расположение (в частности — для фрагментов, портов и маршрутов дуг), изложенным в главе «Визуализация графов».
- Область ответственности ограничивается вычислением пространственного расположения элементов, маршрутизацией дуг и расчетом габаритов для фрагментов. Алгоритм не должен изменять параметры визуального стиля, такие как размеры текстовых полей, характеристики шрифтов, цветовые схемы, геометрические формы элементов и т.д.
- Алгоритм укладки должен корректно обрабатывать вложенные структуры. Для этого может быть использован иерархический обход графовой модели, описанный в главе «Хранение графов».
- По завершении работы алгоритм укладки фиксирует результаты вычислений в хранилище графовых моделей, обновляя координаты вершин, маршруты дуг, а также расположение и размеры фрагментов в соответствующих атрибутах.
- Алгоритм укладки должен обеспечивать корректную работу с портами. Порядок портов и их координаты должны учитываться как обязательные параметры при расчете расположения элементов и маршрутизации дуг.

В библиотеке VGL реализованы следующие алгоритмы укладки, которые позволяют эффективно визуализировать графовые структуры различных типов. Каждая укладка адаптирована для работы с иерархическими атрибутированными графами с портами:

- **Иерархическая укладка** на базе метода Сугиямы [8, 20] реализует рекурсивный обход структуры, начиная с наиболее вложенных фрагментов. Внедрение концепции цепочек на уровне хранения графов позволяет отойти от обработки связей между элементами из разных фрагментов. Весь граф рассматривается как набор независимых фрагментов, что существенно упрощает и ускоряет алгоритм. В рамках данной укладки все порты разделяются на входные (располагаются на верхней границе фрагмента или вершины) и выходные (на нижней). При этом реальные порты имеют определенный порядок, который выступает жестким ограничением и не может нарушаться алгоритмом, в то время как фиктивные порты лишены фиксированного порядка, что дает алгоритму необходимую степень свободы для

оптимизации пересечений и выравнивания линий «на лету». Такая геометрия в сочетании с выравниванием вершин по вертикальным осям минимизирует изломы дуг, что делает изображение более читаемым. Завершающая стадия определяет стиль линий: прямые дуги отрисовываются сплошными, а обратные — пунктирными. При этом для обратных дуг соблюдается общий принцип: они выходят из нижней границы элемента и входят в верхнюю, что полностью согласуется с принятой концепцией расположения портов.

- **Круговая укладка** [16] размещает вершины по периметру концентрических окружностей, что наиболее эффективно для анализа сетевых структур без выраженного направления потока данных. Реализация алгоритма опирается на последовательные проходы «вверх» и «вниз» по уровням иерархии: на восходящем этапе вычисляются габариты вложенных фрагментов, а на нисходящем — уточняются их финальные координаты и углы поворота. Использование двух проходов несколько снижает производительность по сравнению с однопроходными методами, однако это позволяет достичь более эстетичного результата за счет итерационного уточнения позиций элементов. В рамках данной укладки порты распределяются по границам окружностей, при этом реальные порты сохраняют свой жесткий порядок, а фиктивные служат свободными точками для оптимизации межуровневых связей. Минимизация пересечений внутри окружности достигается за счет эвристик группировки наиболее связанных элементов и циклической перестановки смежных элементов. Завершающая стадия маршрутизации использует «скругленные вставки», которые позволяют дугам обтекать неинцидентные вершины по радиусу. Это предотвращает визуальное наложение дуг на элементы и сохраняет чистоту иерархических переходов даже при высокой плотности графа. Пошаговая логика работы алгоритма, адаптированного под специфику иерархии и портов VGL, представлена ниже в виде трех основных этапов:

  1. Расчет размеров фрагментов. Используем восходящий обход дерева вложенности T – это гарантирует, что при обработке текущего фрагмента уже известны геометрические размеры всех его составляющих элементов. На их основе вычисляется радиус R для позиционирования внутренних вершин и фрагментов, а также определяется r как максимальный размер одного из портов. Итоговый размер фрагмента определяется формулой $2R + r + \mathrm{c}, \text{где } \mathrm{c} - \text{константа}$. Данное значение остается неизменным на протяжении всего алгоритма.

2. Расчет позиций элементов. Используем нисходящий обход дерева вложенности T. Процесс позиционирования внутри фрагмента выполняется в строгой последовательности. Сначала полученные от родительских структур координаты части портов конвертируются в базовый порядок, на который накладывается жестко заданный порядок реальных портов. В результате формируется набор портов с точными или аппроксимированными позициями, а также множество полностью свободных портов. Опираясь на зафиксированные позиции, выстраивается начальная последовательность внутренних элементов, которая затем оптимизируется эвристической сортировкой для получения их финального топологического порядка. Отсортированные элементы расставляются на несущей окружности. Далее выполняется вычисление оптимального угла поворота сформированного кольца элементов с целью минимизации длины связующих дуг до портов с известными позициями. На завершающем шаге производится финальное распределение аппроксимированных и свободных портов по периметру фрагмента.
3. Маршрутизация дуг. На завершающем этапе выполняется маршрутизация дуг. Благодаря применению концепции «цепочек», задача маршрутизации строго локализована: алгоритм обрабатывает исключительно сегменты, соединяющие элементы внутри границ одного фрагмента (все межуровневые связи предварительно декомпозированы). Для каждого такого локального соединения вычисляются точные координаты точек входа и выхода: позиция определяется как точка пересечения луча, исходящего из геометрического центра инцидентного элемента, с его фактической границей. Между вычисленными точками строится базовая прямолинейная траектория. В случае возникновения коллизии (если прямая пересекает тело неинцидентного элемента) применяется алгоритм обтекания препятствий: базовая траектория модифицируется так, чтобы дуга плавно огибала препятствие по контуру вспомогательной окружности.

В рамках представленных алгоритмов основное внимание было сосредоточено на задачах вычисления габаритов и точного позиционирования элементов. Маршрутизация связей реализовывалась по остаточному принципу с использованием достаточно стандартных подходов (прямолинейные соединения и базовое обтекание препятствий). Тем не менее, при анализе графов с высокой плотностью дуг такой подход может приводить к избыточному

визуальному шуму и засорению итоговой схемы. Для решения этой проблемы перспективным вектором развития выглядит применение методов жгутования дуг [5]. Данная техника позволила бы стягивать близлежащие дуги в единые пучки, существенно повышая читаемость.

6. Импорт и экспорт графов

Модуль импорта и экспорта графов представляет собой набор независимых компонентов, логика которых полностью изолирована от задач визуализации и хранения данных. Архитектурно эти компоненты выступают функциональными аналогами методов контроллера в парадигме MVC: их роль заключается в преобразовании внешних форматов во внутренние объекты библиотеки и обратно. Компоненты взаимодействуют напрямую с хранилищем графовых моделей, обеспечивая интерпретацию топологии и метаданных независимо от типа источника данных — будь то локальный файл, сетевой поток или область памяти. Несмотря на разнообразие поддерживаемых форматов, все компоненты данного модуля объединены общей философией, базирующейся на четырех принципах:

- Любой процесс обмена данными (импорт или экспорт) должен быть абстрагирован от физического носителя. Использование принципов потоковой обработки позволяет изолировать логику разбора формата от специфики работы с файловой системой или сетевыми протоколами.
- Процесс импорта обязан обеспечивать строгую валидацию входных данных. Любое нарушение синтаксиса или логической целостности графа должно сопровождаться детальным описанием ошибки с указанием контекста (номер строки, некорректный фрагмент), необходимого для оперативной отладки внешних файлов.
- Процесс экспорта должен обеспечивать сохранение не только топологии, но и полного визуального состояния графа. Это подразумевает обязательную фиксацию координат, габаритов элементов, маршрутов дуг и примененных стилей, что гарантирует идентичность модели при её повторной загрузке.
- Реализация должна быть универсальной по отношению к иерархическим структурам. Независимо от ограничений конкретного формата, процесс импорта или экспорта должен корректно восстанавливать или сохранять вложенность фрагментов, типизированные атрибуты и метаданные, описанные в главе «Хранение графов».

Для практической реализации описанных выше принципов в библиотеку VGL включен набор компонентов, адаптированных под специфику инженерных данных и иерархических атрибутированных графов с портами:

- **Компоненты для импорта и экспорта графов в формате GraphML** [22] на базе XML. Формат GraphML выступает в качестве основного способа хранения и обмена данными для библиотеки VGL, т.к. он обеспечивает полную поддержку объектной модели описанной в главе “Хранение графов”, а именно вложенные графы, типизированные атрибуты и порты. Стоит так же отметить что при экспорте, в итоговый файл, записываются не только топология, но и результаты работы модулей визуализации и укладки: координаты элементов, маршруты дуг и габариты фрагментов. Это позволяет восстановить результаты работы при последующем импорте.
- **Компонент для импорта графов в формате DOT** [21] обеспечивает совместимость с экосистемой инструментов Graphviz. Поддержка данного формата позволяет использовать библиотеку как средство визуализации и интерактивного анализа для графов, сгенерированных сторонними инженерными и диагностическими утилитами.
- **Компонент для импорта графов в формате GML** [13] является востребованным в задачах сетевого анализа и академических исследованиях благодаря своей высокой читаемости. Несмотря на то что официальная спецификация GML описывает исключительно «плоские» структуры и не поддерживает иерархию и порты, в VGL реализован механизм восстановления вложенности и портов через использование специальных атрибутов. Таким образом, плоская структура формата преобразуется в полноценную иерархическую модель с портами.

Текущий набор компонентов обеспечивает решение основных задач по обмену данными, потоковой загрузке и сохранению визуального состояния моделей. Список поддерживаемых форматов будет расширяться по мере развития библиотеки. Благодаря открытости архитектуры сторонние разработчики могут реализовывать собственные модули импорта и экспорта под специфические нужды, не дожидаясь выхода новых версий библиотеки и не внося изменений в её основной код.

7. Анализ существующих библиотек и приложений для работы с графами

Выбор инструмента для визуализации сложных инженерных систем требует баланса между алгоритмической мощностью, производительностью графической подсистемы и гибкостью программного управления. В данной главе проведен обзор графовых решений: от

низкоуровневых алгоритмических библиотек до развитых систем визуализации. Цель анализа — определить место разрабатываемой библиотеки среди существующих инструментов, выявить их сильные и слабые стороны, а также обозначить сценарии их применения.

Текущий рынок графового ПО представлен широким спектром инструментов, различающихся по архитектуре, языкам реализации и назначению: yFiles [25], Gephi Toolkit [10], Tulip Library [23], Cytoscape.js [6], Graphviz [11], Higres [18], JGraphT [15], igraph [14], OGDF [24], NetworkX, Boost Graph Library, JGraphX, GraphStream, Prefuse, Sigma.js, G6.js, D3.js, Vis.js, Graph-tool, JUNG, Piccolo2D, GoJS, KeyLines, NetBeans Visual Library, Zest. Подробный разбор каждого из них избыточен и приведет к потере фокуса исследования. Чтобы сохранить системность изложения, мы ограничим область анализа, последовательно просеивая инструменты по ключевым критериям.

Начнем процесс просеивания с исключения инструментов, не имеющих активного цикла поддержки: Prefuse, JUNG, Piccolo2D, JGraphX (mxGraph), GraphStream, Higres, Zest и NetBeans Visual Library. Отсутствие обновлений в течение длительного времени делает их интеграцию в современные системы нецелесообразной из-за рисков несовместимости с актуальными программными средами и неоправданного роста затрат на сопровождение такого кода.

Далее стоит исключить графовые СУБД (Neo4j, JanusGraph), предназначенные для персистентного хранения и обработки запросов к связанным данным. Подобные системы не содержат механизмов автоматизированной укладки и интерактивной визуализации. В рамках разработки графической подсистемы СУБД рассматриваются исключительно как возможные внешние источники данных, но не как инструменты построения интерактивных схем.

Также стоит исключить инструменты, базирующиеся на веб-окружении: D3.js, Sigma.js, Cytoscape.js, G6.js, Vis.js, GoJS и KeyLines. Несмотря на развитый функционал данных библиотек, их использование в нативном Java-ПО требует встраивания компонентов WebView. Подобный подход создает значительную архитектурную избыточность, усложняет прямой доступ к структурам данных в памяти JVM и ограничивает производительность интерфейса. При визуализации графов размерностью более 10 000 элементов наличие прослойки между графическим движком браузера и основным кодом приложения становится критическим узким местом.

Значимым критерием является технологическая совместимость с целевой средой. Это ограничение обуславливает отказ от NetworkX вследствие низкой производительности Python-интерпретатора на массивных графах и Graph-tool из-за его жесткой привязки к контейнеризированной инфраструктуре (Docker), неприемлемой для кроссплатформенного

десктопного ПО. Вне фокуса исследования остается и Boost Graph Library: при всей своей алгоритмической полноте она лишена графического ядра, а специфика её реализации на шаблонах C++ делает создание JNI-оберток избыточно трудоемким.

После первичного отсева инструментов, не соответствующих базовым требованиям по поддержке и архитектуре, оставшаяся группа требует детального разбора. Оценка проводится по вектору «от модели к реализации»: от фундаментальных возможностей до прикладных аспектов встраивания в инженерное ПО. В итоговый перечень для детального функционального сопоставления вошли нативные Java-библиотеки (yFiles, Gephi Toolkit, JGraphT) и группа низкоуровневых решений (igraph, OGDF, Tulip, Graphviz). Несмотря на сложность интеграции внешнего кода и ориентированность некоторых инструментов на генерацию статических схем, статус данных решений как индустриальных стандартов обосновывает их сохранение в выборке. Их анализ позволит определить технологический предел существующих систем и сформировать перечень требований к разрабатываемой библиотеке, подтвердив необходимость её создания в текущем технологическом ландшафте.

Эффективность применения рассматриваемых библиотек (yFiles, JGraphT, Gephi, Tulip, igraph, OGDF, Graphviz) в инженерных задачах зачастую ограничена архитектурой их внутренних моделей данных. В большинстве решений (JGraphT, igraph, Gephi) поддержка вложенных графов носит декларативный характер: алгоритмы автоматического размещения оперируют плоской топологией и не учитывают границы контейнеров. В инструментах с нативной иерархией (yFiles, Graphviz) работа со сложными структурами осложнена либо трудоемкостью программной настройки, либо статической природой кластеров. Аналогичное несоответствие наблюдается в реализации портов: если в yFiles и Graphviz они являются лишь логическими точками привязки, то в OGDF или JGraphT специализированный интерфейс для них отсутствует. Это приводит к трассировке ребер в геометрический центр узла и затрудняет построение схем с ортогональной укладкой без создания значительных надстроек над графическим ядром. Дополнительным барьером выступает изоляция атрибутов данных от их визуального воплощения. В yFiles или Tulip для отображения метаданных требуется разработка собственных компонентов отрисовки, а отсутствие механизмов автоматической синхронизации вынуждает разработчика вручную обновлять графический слой при смене внутреннего состояния объектов. Данные структурные ограничения предопределяют сложности не только при управлении графом в памяти, но и при попытках его корректного сохранения или передачи через внешние форматы.

Поддержка стандартов (GraphML, DOT, GML) в рассматриваемых библиотеках носит фрагментарный характер, что ограничивает их использование в единых инженерных

процессах. Решения общего назначения (JGraphT, igraph, Gephi) поддерживают DOT и GraphML, однако ограничиваются передачей топологии, игнорируя визуальные параметры и иерархию. Среда yFiles, напротив, сохраняет графические атрибуты в расширенном GraphML, но лишена нативной поддержки DOT, что исключает прямое взаимодействие с экосистемой Graphviz. Библиотеки, изначально разработанные на C++ (Tulip, OGDF), отдают приоритет собственным (TLP) или упрощенным (GML) форматам, что затрудняет трансляцию их специфических данных в Java-приложения без написания внешних адаптеров. Общей проблемой для всех инструментов остается несовместимость схем метаданных: при передаче графа инженерные атрибуты (порты, физические параметры, вложенность) либо игнорируются, либо сохраняются в нетипизированном виде. В результате стандартные форматы превращаются в средство передачи лишь «голой» топологии, а задача корректного восстановления полной инженерной модели перекладывается на разработчика, вынужденного создавать индивидуальные парсеры для каждой библиотеки.

Анализ алгоритмов укладки выявляет несоответствие между методами визуализации общих сетей и требованиями инженерного проектирования. Большинство открытых библиотек (JGraphT, igraph, Gephi, Tulip) ориентированы на силовые алгоритмы, которые эффективны для анализа связей, но не гарантируют сохранения структурной иерархии. Это приводит к потере целостности составных графов: алгоритмы зачастую игнорируют границы контейнеров, размещая дочерние узлы вне родительских областей. Качественные иерархические методы (алгоритм Сугиямы), минимизирующие пересечения, полноценно представлены лишь в yFiles и Graphviz, однако они функционируют как «черные ящики», затрудняя жесткую привязку ребер к конкретным портам. Для библиотеки OGDF характерно наличие сложной математической базы ортогональной трассировки, но её адаптация в Java-экосистеме требует значительных усилий по настройке параметров обхода тел компонентов. В отсутствие нативной поддержки портов большинство рассматриваемых решений используют маршрутизацию дуг от центра к центру, что критически снижает читаемость инженерных схем. Таким образом, рынок предлагает выбор между дорогостоящими закрытыми продуктами и трудоемкой разработкой собственных механизмов контроля геометрии на базе инструментов, изначально не рассчитанных на работу с портами и вложенностью.

Завершающим критерием анализа является простота интеграции библиотек в Java-экосистему и их способность обеспечивать интерактивное взаимодействие. Лидером сегмента остается yFiles, предлагающий нативные высокоуровневые компоненты для Swing и JavaFX. Библиотека поддерживает высокую производительность за счет механизмов виртуализации и

многоуровневой детализации, что позволяет сохранять плавность навигации при масштабировании схем. В противоположность этому, Graphviz ориентирован на статическую генерацию: его использование в Java-интерфейсах ограничено отображением готовых файлов, что исключает динамическое манипулирование узлами. Такие платформы как Gephi, обладают мощными средствами навигации и обработки данных, но несмотря на наличие программных интерфейсов для управления логикой, их графические движки тесно интегрированы с родительскими платформами (NetBeans, OSGi), что затрудняет встраивание интерактивного компонента в стороннее ПО в качестве изолированного виджета. Библиотеки общего назначения, такие как JGraphT, обладают развитым математическим аппаратом, но их визуализация часто вынесена в отдельные модули (например, JGraphX), что требует ручной синхронизации объектной модели и графического представления. Инструменты с C++ ядром (igraph, OGDF, Tulip) при встраивании через JNI механизмы создают риски нестабильности и усложняют кроссплатформенное развертывание. Отсутствие поддержки консольного режима во многих инструментах, описанных выше, ограничивает их применение на серверах: автоматическая генерация отчетов становится невозможной без настройки виртуального графического окружения.

8. Заключение

В статье представлена архитектура и возможности Visual Graph Library — специализированного библиотечного решения для работы с иерархическими атрибутированными графами с портами. Целью данной библиотеки является устранение технологического разрыва между низкоуровневыми алгоритмическими библиотеками и закрытыми графическими системами, чье использование затруднено в открытых инженерных проектах. Проведенный анализ показал, что существующие решения зачастую игнорируют специфику инженерных задач, связанных с вложенными графами и/или портами или накладывает критические ограничения на стабильность и развертывание в рамках Java-экосистемы.

Visual Graph Library реализует системный подход, объединяющий модель хранения сложных графовых структур, алгоритмы автоматической укладки и подсистему отрисовки в рамках единой модульной архитектуры. Ключевой особенностью данной системы является ее ориентированность на прикладные инженерные задачи, требующие работы с вложенностью и семантикой портов в среде Java. Разработанные подходы применимы для визуализации внутренних представлений программ в компиляторах (например, GCC), средах исполнения (например, Python), а также специализированных системах, таких как Cloud Sisal [2].

Визуальный анализ таких структур используется для отладки трансформаций кода и верификации алгоритмов оптимизации. Поддержка иерархии позволяет корректно группировать инструкции в базовые блоки, а строгая семантика портов — точно отслеживать зависимости по данным. Это обеспечивает построение читаемых схем, облегчающих разработчикам анализ внутреннего состояния транслятора.

Основные результаты:

- Реализован механизм гибридной схемы хранения на диске и в памяти, поддерживающий вложенные графы, порты и типизированные атрибуты, что позволяет корректно моделировать потоки данных и управляющие графы.
- Реализован набор специализированных компонентов, в модуле визуализации графов, обеспечивающий стабильную работу как в интерактивных интерфейсах, так и в серверных сценариях без жесткой зависимости от графического окружения ОС.
- Реализована поддержка портов и вложенности в алгоритмах Сугиямы и круговой укладки, что позволяет автоматизировать процесс построения читаемых схем для сложных инженерных задач.
- Разработан модуль импорта и экспорта, поддерживающий сериализацию иерархических атрибутированных графов с портами в формат GraphML, а также их десериализацию из форматов GraphML, DOT и GML.

Направления дальнейших исследований и разработки:

- Разработка графического компонента для иерархического отображения графа с поддержкой отложенной загрузки вложенных графов.
- Развитие библиотеки для решения проблем масштабируемости и обработки графов сверхбольшого размера. Оптимизация алгоритмов кэширования и индексации в рамках гибридной модели хранения для минимизации задержек при обращении к диску на графах с миллионами узлов.
- Оптимизация производительности алгоритмов укладки для графов сверхвысокой плотности путем внедрения параллельных вычислений.
- Внедрение алгоритма жгутования в качестве опционального режима маршрутизации, необходимого для предварительного визуального анализа графов с высокой плотностью дуг.

Библиотека распространяется с открытым исходным кодом, что позволяет использовать её как базу для создания собственных систем для визуализации и анализа сложно структурированной информации большого объема на основе графовых моделей.

Список литературы

1. **Касьянов В. Н.** Иерархические графы и графовые модели: вопросы визуальной обработки // Проблемы систем информатики и программирования. Новосибирск: ИСИ СО РАН, 1999. – С. 7–32.
2. **Касьянов В.Н., Гордеев Д.С., Золотухин Т.А., Касьянова Е.В., Кондратьев Д.А.** Система облачного параллельного программирования CPPS: визуализация и верификация Cloud Sisal программ : моногр. / Под ред. В.Н. Касьянова ; Ин-т систем информатики им. А.П. Ершова СО РАН. – Новосибирск: ИПЦ НГУ, 2020. – 256 с. – (Сер.: Конструирование и оптимизация программ; Вып. 22). ISBN 978-5-4437-1123-2. DOI: https://doi.org/10.31144/978-5-4437-1123-2.
3. **Касьянов В. Н., Евстигнеев В. А.** Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003. – 1104 с.
4. **Касьянов В. Н., Золотухин Т. А.** Visual Graph — система для визуализации сложноструктурированной информации большого объема на основе графовых моделей // Научная визуализация. 2015. Т. 7, № 4. – С. 44–59.
5. **Apanovich Z.V., Kislicyna T.A., Vinokurov P.S.** Tools for Visual Analysis of Information Content of Portals Included in Linked Open Data Cloud // Proceedings of the 13th All-Russian Scientific Conference "Digital libraries: Advanced Methods and Technologies, Digital Collections". Voronezh, Russia, October 19-22, 2011. CEUR Workshop Proceedings. 2011. – Volume 803. – P. 113–120.
6. **Cytoscape**. [Электронный ресурс]. URL: https://cytoscape.org/ (дата обращения: 24.02.2026).
7. **Di Battista G., Eades P., Tamassia R., Tollis I. G.** Graph Drawing: Algorithms for Visualization of Graphs. Prentice Hall, 1999. – 397 p.
8. **Eiglsperger M., Siebenhaller M., Kaufmann M.** An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing // Journal of Graph Algorithms and Applications. – 2005. – Vol. 9, № 3. – P. 305–325.
9. **Gansner E. R., Koutsofios E., North S. C., Kiem-Phong Vo** A Technique for Drawing Directed Graphs // IEEE Transactions on Software Engineering. – 1993. – Vol. 19, № 3. – P. 214–230.
10. **Gephi**. [Электронный ресурс]. URL: https://gephi.org/ (дата обращения: 24.02.2026).
11. **Graphviz.** [Электронный ресурс]. URL: https://graphviz.org/ (дата обращения: 24.02.2026).
12. **Herman I., Melancon G., Marshall M. S.** Graph visualization and navigation in information visualization: a survey // IEEE Trans. on Visualization and Computer Graphics. – 2000. – Vol. 6, Issue 1. – P. 24–43.
13. **Himsolt M.** GML: A portable graph file format. – Technical Report, University of Passau, 1996. – 8 p.
14. **igraph.** [Электронный ресурс]. URL: https://igraph.org/ (дата обращения: 24.02.2026).
15. **JGraphT.** [Электронный ресурс]. – URL: https://jgrapht.org/ (дата обращения: 24.02.2026).

16. **Kasyanov V. N., Merculov A. M., Zolotuhin T. A.** A circular layout algorithm for attributed hierarchical graphs with ports // Journal of Physics: Conference Series. – 2021. – Vol. 2099, № 1. – Article ID 012051.
17. **Krasner G. E., Pope S. T.** A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80 // Journal of Object-Oriented Programming. – 1988. – Vol. 1, № 3. – P. 26–49.
18. **Lisitsyn I. A., Kasyanov V. N.** Higres – visualization system for clustered graphs and graph algorithms // Lecture Notes in Computer Science. – 1999. – Vol. 1731. – P. 82–89.
19. **Sugiyama K.** Graph drawing and applications. For software and knowledge engineers. – World Scientific, 2002. – 232 p.
20. **Sugiyama K., Tagawa S., Toda M.** Methods for Visual Understanding of Hierarchical System Structures // IEEE Transactions on Systems, Man, and Cybernetics. – 1981. – Vol. SMC-11, № 2. – P. 109–125.
21. **The DOT Specification.** Graphviz.org. [Электронный ресурс]. URL: https://graphviz.org/doc/info/lang.html (дата обращения: 24.02.2026).
22. **The GraphML Specification.** [Электронный ресурс]. URL: http://graphml.graphdrawing.org/ (дата обращения: 24.02.2026).
23. **Tulip.** [Электронный ресурс]. URL: https://tulip.labri.fr/ (дата обращения: 24.02.2026).
24. **OGDF**. [Электронный ресурс]. URL: https://ogdf.uos.de/ (дата обращения: 24.02.2026).
25. **yFiles.** [Электронный ресурс]. – URL: https://www.yworks.com/products/yfiles (дата обращения: 24.02.2026).
26. **Zolotuhin T.** Visual Graph: an interactive system for the visualization of hierarchical attributed graphs // Bulletin of the Novosibirsk Computing Center. Series: Computer Science. Issue 37. – 2014. – P. 163–180.