Сравнительный анализ и оценка эффективности гибридного подхода к генерации DSL в среде UEPE

4. Сравнительный анализ

В рамках данного исследования была поставлена задача не просто разработать инструментарий для создания предметно-ориентированных языков (DSL), но и предложить методологию, способную конкурировать с устоявшимися промышленными стандартами, снижая при этом порог вхождения в область языковой инженерии. Для объективной оценки предложенной системы **DslTools** необходимо провести глубокий сравнительный анализ с существующими парадигмами разработки языков.

Анализ строится на сопоставлении трех ключевых архитектурных подходов:

1. **Проекционная парадигма (Projectional Editing)**, наиболее ярко представленная системой JetBrains MPS.
2. **Классическая парсерная парадигма (Parser-based)**, стандартом которой является фреймворк Eclipse Xtext.
3. **Генеративная парадигма (Pure LLM)**, основанная на прямом транслировании требований в код с помощью больших языковых моделей.

Сравнение проводится в контексте концепции **UEPE (Unified Executable Parsing Environment)**, реализованной в DslTools, где приоритетами являются скорость цикла «редактирование-исполнение» (Edit-Run), снижение когнитивной нагрузки на разработчика и надежность генерируемого кода.

4.1. Критерии сравнения и методологическая база

Для формализации сравнения были выделены критические метрики, определяющие эффективность инструментария языковой инженерии [1]:

- **Когнитивная нагрузка и порог вхождения (Barrier to Entry):** Оценка объема специализированных знаний (теория компиляторов, мета-языки, API фреймворков), необходимых для создания минимально жизнеспособного продукта (MVP) языка.[3]
- **Скорость прототипирования (Prototyping Velocity):** Временные затраты на итерацию изменения грамматики. Ключевым фактором здесь является наличие или отсутствие этапов промежуточной кодогенерации и пересборки среды (Build/Deploy cycles).[4]
- **Архитектурная детерминированность (Architectural Determinism):** Способность системы гарантировать корректность синтаксиса и семантики. Данный критерий

критичен при сравнении с вероятностными моделями (LLM), склонными к галлюцинациям.[1]

- **Гибкость нотации и редактирования:** Возможности системы по поддержке различных форм представления кода (текст, таблицы, диаграммы) и удобство работы с ними (Refactoring, Copy-Paste, VCS integration).[6]

4.2. Анализ проекционных систем (JetBrains MPS)

JetBrains MPS (Meta Programming System) представляет собой вершину эволюции подхода LOP (Language Oriented Programming). Фундаментальное отличие MPS заключается в отказе от парсинга текстового представления. Вместо этого разработчик и пользователь работают напрямую с Абстрактным Синтаксическим Деревом (AST), которое «проецируется» на экран в виде, имитирующем текст, таблицы или диаграммы.[6]

Преимущества подхода:
Архитектура MPS устраняет целый класс проблем, свойственных текстовым языкам. Поскольку исходный код хранится непосредственно как граф объектов, исчезают проблемы синтаксической неоднозначности (ambiguity) и необходимость в сложных алгоритмах парсинга (GLR, LALR). Это позволяет свободно комбинировать различные языки в одном файле — например, встраивать SQL-запросы или математические формулы непосредственно в Java-код без использования строковых литералов, что подтверждается исследованиями.9 Проекционная природа также открывает возможности для использования нетекстовых нотаций, что делает DSL более доступными для предметных экспертов, не являющихся программистами.2
Критические недостатки в контексте задач исследования:
Однако анализ литературы и опыта эксплуатации выявляет существенные барьеры, препятствующие массовому внедрению MPS для быстрого прототипирования:

1. **Высокий порог вхождения («The Cliff»):** Изучение MPS требует погружения в сложную экосистему мета-моделирования. Разработчик должен освоить концепции Structure, Editor, Constraints, Behavior и Generator, а также специфический язык BaseLanguage. Исследования показывают, что студенты и начинающие разработчики испытывают значительные трудности с освоением инструментария, что приводит к высокому проценту отказа от использования.[3]
2. **Эффект «Зловещей долины» редактирования (Uncanny Valley):** Несмотря на то что редактор MPS визуально напоминает текстовый, он не ведет себя как таковой. Привычные паттерны работы (выделение текста, копирование части строки, глобальный поиск по тексту) работают иначе или требуют специальной реализации. Это создает постоянное когнитивное трение для пользователей, привыкших к классическим IDE.[2]
3. **Тяжеловесность инфраструктуры:** MPS является ресурсоемким приложением. Процессы индексации, перестройки моделей и генерации Java-кода требуют значительных вычислительных мощностей и времени. Это противоречит требованию мгновенной обратной связи, заложенному в концепцию UEPE.[4]

4. **Проблемы интеграции и контроля версий:** Хранение кода в формате XML делает невозможным использование стандартных инструментов слияния (git merge/diff) без специализированных плагинов, что усложняет командную работу за пределами экосистемы JetBrains.[6]

Вывод по MPS: Система является мощным решением для долгосрочных промышленных проектов, где затраты на обучение окупаются гибкостью, но избыточна и слишком сложна для задач быстрого прототипирования и обучения.[3]

4.3. Анализ классических парсерных систем (Eclipse Xtext)

Eclipse Xtext является стандартом де-факто для разработки текстовых DSL. Он опирается на классический конвейер компиляции: грамматика (EBNF) преобразуется генератором (ANTLR) в парсер, который строит AST. Инструментарий тесно интегрирован с экосистемой Eclipse EMF (Eclipse Modeling Framework).[6]

Преимущества подхода:
Ключевым преимуществом Xtext является работа с обычными текстовыми файлами. Это гарантирует полную совместимость с любыми системами контроля версий и позволяет редактировать код в любом текстовом редакторе, что снижает зависимость от конкретного инструмента (Tool Lock-in) по сравнению с MPS.6 Кроме того, Xtext предоставляет зрелый инструментарий для статического анализа и рефакторинга.7
Критические недостатки в контексте задач исследования:

1. **Длительный цикл обратной связи (Compilation Bottleneck):** Главным ограничением Xtext является необходимость полной перегенерации артефактов языка (парсера, лексера, Java-классов модели) при любом изменении грамматики. Этот процесс, включающий запуск генераторов MWE2 и компиляцию плагинов Eclipse, занимает значительное время, разрывая поток разработки.[5]
2. **Ручная реализация семантики:** Xtext автоматизирует только синтаксический анализ. Реализация валидации, вывода типов и генерации кода (через JvmModelInferrer или Xtend) полностью ложится на плечи разработчика. Это именно та рутинная работа, которую предлагаемый в работе метод стремится автоматизировать с помощью ИИ.[1]
3. **Сложность экосистемы Eclipse:** Архитектура, основанная на OSGi-бандлах и EMF, создает значительный «архитектурный шум» (boilerplate), усложняющий понимание работы системы для новичков.[7]

Вывод по Xtext: Xtext обеспечивает надежность и стандартизацию, но страдает от медленного цикла разработки и необходимости ручного кодирования семантики, что замедляет прототипирование.[11]

4.4. Анализ генеративного подхода (Pure LLM / NL2DSL)

Современные подходы, использующие большие языковые модели (LLM) для прямой генерации кода по описанию на естественном языке (Natural Language to DSL), предлагают радикально иной путь, устраняя необходимость в формальном описании грамматики.[1]

Преимущества:

- **Нулевой порог вхождения:** Отсутствует необходимость изучения EBNF или API инструментов. Языком описания является естественный язык.

Недостатки и риски:

- **Галлюцинации и отсутствие детерминизма:** Исследования показывают, что без жестких ограничений LLM склонны генерировать синтаксически некорректный код (высокий Unparsed Rate) или использовать вымышленные конструкции (Hallucination Rate). Вероятностная природа моделей не позволяет гарантировать, что один и тот же запрос даст одинаковый результат, что неприемлемо для инженерных задач.[1]
- **Сложность поддержки:** Код, сгенерированный без формальной модели, сложно поддерживать и рефакторить, так как у системы нет понимания его структуры.

4.5. Сравнительная оценка системы DslTools

Предложенная в работе система **DslTools** реализует гибридный подход, объединяющий строгий детерминизм синтаксического анализа с гибкостью генеративного ИИ.

4.5.1. Архитектурные преимущества

В отличие от Xtext и MPS, DslTools использует **интерпретируемую грамматику**. Объект GrammarObject [1] строится динамически в памяти при старте или обновлении, что исключает фазу кодогенерации парсера и компиляции. Это обеспечивает реализацию механизма **Hot Reload**, позволяющего менять правила языка на лету с сохранением контекста сессии (или его изолированным перезапуском), что приближает скорость разработки DSL к скорости написания скриптов на Python.[1]

4.5.2. Роль ИИ в гибридной модели

DslTools не использует ИИ для парсинга (избегая ошибок структуры), но использует его для **генерации семантики** (Visitor-ов). Это решает главную проблему Xtext — необходимость ручного написания кода обхода дерева. ИИ, получая на вход строгую EBNF-схему и пользовательские аннотации, генерирует код на Python для обработки узлов AST. Таким образом, разработчик получает надежность детерминированного парсера и скорость разработки, свойственную LLM-генераторам.[1]

4.5.3. Сводная таблица сравнения

В Таблице 3 представлены итоговые результаты сравнительного анализа

рассматриваемых систем.

Таблица 3. Сравнительный анализ инструментальных средств разработки DSL

Критерий сравнения	**JetBrains MPS**	**Eclipse Xtext**	**Pure LLM Generation**	**DslTools (Предлагаемое решение)**
Базовая парадигма	Проекционная (AST)	Парсерная (Генерация кода)	Генеративная (Вероятностная)	**Гибридная (Интерпретация + LLM)**
Способ определения синтаксиса	Структура моделей (XML)	Грамматика EBNF (Текст)	Промпты (Естественный язык)	**Грамматика EBNF + Аннотации**
Реализация семантики	Ручная (Java/BaseLanguage)	Ручная (Java/Xtend)	Неявная / Черный ящик	**Автоматическая (LLM-Visitor)**
Цикл изменений (Edit-Run)	Средний (Rebuild Model)	Медленный (Compile & Deploy)	Быстрый (One-shot)	**Мгновенный (Hot Reload)**
Порог вхождения	Критически высокий	Высокий	Низкий	**Низкий (Знание EBNF)**
Гарантия корректности	100% (By design)	100% (Static Analysis)	Низкая (Галлюцинации)	**Высокая (Детерминированный парсер)**
Требования к ресурсам	Высокие (JVM, Индексы)	Высокие (Eclipse Platform)	Зависит от API	**Низкие (Docker-контейнер)**

4.6. Вывод по сравнительному анализу

Проведенный анализ показывает, что DslTools занимает уникальную нишу между

тяжеловесными промышленными "комбайнами" (MPS, Xtext) и ненадежными генераторами на чистом ИИ. Система жертвует некоторой гибкостью (например, отсутствием проекционных нетекстовых нотаций, доступных в MPS) ради радикального снижения сложности разработки и ускорения итерационного цикла. Это делает предложенный подход оптимальным для задач быстрого прототипирования языков, обучения теории компиляторов и создания DSL средней сложности, где критична скорость получения работающего интерпретатора.

---

5. Заключение

5.1. Итоги исследования

В рамках данной курсовой работы было проведено исследование и разработка программного комплекса **DslTools**, предназначенного для автоматизации процесса создания предметно-ориентированных языков (DSL). Основной гипотезой работы являлось предположение, что интеграция детерминированных методов синтаксического анализа с генеративными возможностями больших языковых моделей (LLM) позволит существенно сократить трудоемкость разработки языковых процессоров без потери надежности, свойственной формальным методам.

В ходе работы была спроектирована и реализована архитектура **Unified Executable Parsing Environment (UEPE)**, которая обеспечивает выполнение полного цикла разработки языка в единой среде с поддержкой горячей перезагрузки (Hot Reload). Это позволило преодолеть ограничения классических компилируемых подходов (Xtext), связанные с длительным циклом сборки, и ограничения проекционных подходов (MPS), связанные с высоким порогом вхождения и сложностью инструментария.

5.2. Соотнесение результатов с поставленными задачами

Результаты работы соотносятся с задачами, сформулированными во введении (раздел 1.2), следующим образом:

1. **Задача 1: Разработка модели представления грамматики.**
   - *Статус:* **Выполнена.**
   - *Результат:* Разработана структура GrammarObject, инкапсулирующая граф правил EBNF и словарь терминальных символов (RegEx). Данная модель поддерживает динамическую инстанциацию, что стало основой для механизма интерпретации грамматик без промежуточной кодогенерации парсера.[1]
2. **Задача 2: Формализация метода автоматической генерации семантических обработчиков на базе паттерна Visitor.**
   - *Статус:* **Выполнена.**
   - *Результат:* Формализован алгоритм преобразования правил грамматики в контекстные промпты для LLM. Реализован механизм генерации классов Python,

реализующих интерфейс Visitor, где логика обхода дерева и выполнения операций синтезируется моделью на основе структуры AST и пользовательских аннотаций. Это подтверждает возможность автоматизации написания рутинного кода интерпретатора.[1]

3. **Задача 3: Реализация прототипа системы DslTools как интерпретируемой среды исполнения.**
    - *Статус:* **Выполнена.**
    - *Результат:* Создан рабочий прототип системы, развертываемый в контейнерах Docker. Реализован веб-интерфейс (Language Workbench), поддерживающий редактирование грамматики, визуализацию деревьев разбора (CST/AST) и исполнение кода на целевом языке. Подтверждена работоспособность механизма **Isolated Re-instantiation**, обеспечивающего чистоту состояния при горячей замене грамматики.[1]
4. **Задача 4: Исследование влияния пользовательских аннотаций на точность генерации программного кода.**
    - *Статус:* **Не рассмотрена в текущем тексте (Запланирована на будущее).**
    - *Комментарий:* В представленном объеме работы основной упор был сделан на архитектурную реализацию среды и интеграцию с API LLM. Вопрос количественной оценки того, как детальность и формат пользовательских подсказок (prompts) внутри грамматики влияют на метрики корректности генерируемого кода (Unparsed Rate, Semantic Accuracy), требует проведения масштабного эксперимента на наборе различных DSL.
    - *План дальнейших исследований:* Данная задача переносится в план дальнейших исследований. Планируется создание тестового набора (benchmark suite) из 10-15 эталонных DSL различной сложности и проведение серии экспериментов для выявления корреляции между объемом аннотаций и качеством работы LLM-генератора. Это необходимо для выработки методических рекомендаций по написанию «LLM-friendly» грамматик.

5.3. Научная и практическая значимость

Научная новизна работы заключается в предложенном **гибридном методе**, который впервые рассматривает LLM не как замену парсеру, а как дополнение к нему в рамках паттерна Visitor, управляемое формальной грамматикой. Практическая значимость подтверждается возможностью использования DslTools в образовательном процессе для демонстрации принципов работы компиляторов, а также в индустрии для быстрого прототипирования микро-языков, где использование MPS или Xtext экономически нецелесообразно.

5.4. Перспективы развития

Помимо исследования влияния аннотаций (Задача 4), дальнейшее развитие системы DslTools предполагается в следующих направлениях:

- **Решение проблемы ко-эволюции (Co-evolution):** Разработка механизмов автоматической миграции существующих скриптов DSL при изменении грамматики, что является сложной задачей в текстовых системах по сравнению с проекционными.[5]
- **Оптимизация производительности:** Внедрение кэширования результатов генерации семантики и переход на более легковесные локальные модели (например, квантованные версии Qwen2.5) для снижения задержек при генерации.[1]

Таким образом, работа демонстрирует жизнеспособность подхода, объединяющего строгость формальных грамматик с гибкостью современного ИИ, открывая новые возможности в области демократизации разработки языковых инструментов.

Источники

1. Report.pdf
2. Experimental Comparison of Editor Types for Domain-Specific ..., дата последнего обращения: декабря 31, 2025, https://www.mdpi.com/2076-3417/12/19/9893
3. Enhancing Educational Support for JetBrains MPS with a Retrieval ..., дата последнего обращения: декабря 31, 2025, https://www.researchgate.net/publication/390280053_Enhancing_Educational_Support_for_JetBrains_MPS_with_a_Retrieval-Augmented_LLM_Chatbot_A_Structured_Knowledge_Integration_Approach
4. Performance - MPS Platform Docs - mbeddr, дата последнего обращения: декабря 31, 2025, http://mbeddr.com/mps-platform-docs/home/performance/
5. Supporting Meta-model-based Language Evolution and Rapid ..., дата последнего обращения: декабря 31, 2025, https://arxiv.org/html/2401.17351v1
6. What are the main differences between Jetbrains' MPS and Eclipse ..., дата последнего обращения: декабря 31, 2025, https://stackoverflow.com/questions/2603134/what-are-the-main-differences-between-jetbrains-mps-and-eclipse-xtext
7. Which language workbench is better Xtext or Jetbrains MPS ... - Quora, дата последнего обращения: декабря 31, 2025, https://www.quora.com/Which-language-workbench-is-better-Xtext-or-Jetbrains-MPS-for-creating-DSLs-I-want-to-know-pros-and-cons-of-selecting-one-or-another
8. How Does MPS Work? - Concepts - JetBrains, дата последнего обращения: декабря 31, 2025, https://www.jetbrains.com/mps/concepts/
9. A Language Workbench in Action - MPS - Martin Fowler, дата последнего обращения: декабря 31, 2025, https://martinfowler.com/articles/mpsAgree.html
10. Language and IDE Modularization, Extension and Composition with ..., дата последнего обращения: декабря 31, 2025, https://voelter.de/data/pub/Voelter-GTTSE-MPS.pdf
11. Developing Business Applications using Jetbrains MPS, дата последнего обращения: декабря 31, 2025,

https://tomassetti.me/business-applications-jetbrains-mps-daniel-stieger/
12. Tool Lock-in vs. Semantic Lock-in | by Markus Voelter | Medium, дата последнего обращения: декабря 31, 2025, https://markusvoelter.medium.com/tool-lock-in-vs-semantic-lock-in-4f87600c7d44