НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
май–июнь 2026 Том 26 № 3 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
May–June 2026 Vol. 26 No 3 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

ИТМО

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2026-26-3-565-573
УДК 004.4′2

Метод предметно-ориентированного анализа программного кода

Дмитрий Владимирович Кознов✉

Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация
d.koznov@spbu.ru✉, https://orcid.org/0000-0003-2632-3193

Аннотация
Введение. Современные компании, имеющие в составе своих продуктов и технологий сложное программное обеспечение, заинтересованы в технологической независимости по его разработке. Более того, поскольку такое программное обеспечение часто оказывается очень объёмным (речь идёт о миллионах строк кода), то для его тестирования, сопровождения и трансформации требуются многофункциональные инструменты, которые невозможно получить готовыми из-за труднорешаемости многих базовых задач анализа кода (при этом часто эти задачи оказываются неразрешимыми, например, анализ указателей и сборка мусора в C/C++). Вместе с тем такие кодовые базы используют возможности языков программирования не полностью, поэтому классические задачи по анализу кода для этих частных случаев могут оказаться разрешимыми. **Метод.** Предложен подход для разработки предметно-ориентированных решений по анализу программного кода, предназначенных для конкретных больших кодовых баз. Такие решения оказываются востребованными для задач фаззинга, символьного исполнения и генерации тестов, а также для автоматических трансформаций/оптимизаций кода, поиска уязвимостей и т. д. Создание собственных предметно-ориентированных решений, работающих с конкретной кодовой базой, возможно, если компании имеют ресурсы для использования открытых проектов (Eclipse, Low Level Virtual Machine, Microsoft Visual Studio Code и др.), а высокая стоимость целевого проекта оправдывает такие затраты (эффективные решения помогают экономить большие средства, значительно повышая качество). **Основные результаты.** Разработанный метод состоит из следующих шагов: анализ проблемы и идей, проектирование решения, разработка требований, разработка самого решения, апробация, внедрение и передача решения. Требования включают создание шаблонов кода, на которых решение должно работать и которые ограничивают использование целевого языка программирования в рамках текущей кодовой базы. При этом разработка требований, решения и апробация выполняются параллельно, тестирование совмещено с разработкой. С помощью предложенного метода созданы инструменты для статического анализа C-приложений для нужд фаззинга, средство поиска клонов и рефакторинга большой кодовой базы программного обеспечения сетевых устройств и другие решения. В реализованных проектах было задействовано от одного до трех разработчиков, длительность проектов составляла от одного до двух лет. **Обсуждение.** Представленный метод оказался ресурсозатратным. Анализ реализованных проектов показал, что при исходном наличии готового технического задания можно в несколько раз снизить ресурсы по реализации решений по анализу кода. Метод направлен на итеративное выявление требований к решению в ситуации, когда готовое техническое задание невозможно создать, а также существенно понижает риски по разработке невостребованных решений.

Ключевые слова
анализ кода, статический анализ, большие кодовые базы, предметно-ориентированный подход, модель процесса разработки

Ссылка для цитирования: Кознов Д.В. Метод предметно-ориентированного анализа программного кода // Научно-технический вестник информационных технологий, механики и оптики. 2026. Т. 26, № 3. С. 565–573. doi: 10.17586/2226-1494-2026-26-3-565-573

© Кознов Д.В., 2026

Domain-specific code analysis approach

Dmitry V. Koznov✉

St. Petersburg State University, Saint Petersburg, 199034, Russian Federation

d.koznov@spbu.ru✉, https://orcid.org/0000-0003-2632-3193

Abstract.

Modern companies that incorporate complex software systems as part of their market products are increasingly seeking technological independence in the development and maintenance of these systems. Such companies typically manage very large codebases, often comprising millions of lines of code. Testing, maintaining, and transforming these codebases requires multi-functional analysis and development tools, which cannot simply be obtained off the shelf. This is largely due to the inherent difficulty of many fundamental program analysis problems — indeed, several of these problems are formally undecidable (for example, pointer analysis and garbage collection for C programs). At the same time, large industrial codebases often use only a limited subset of the features available in their respective programming languages. Consequently, program analysis problems that are undecidable in the general case may become tractable for these restricted, domain-specific subsets. This observation creates opportunities for specialized approaches. This article proposes an approach for developing domain-specific program analysis solutions tailored to particular large industrial codebases. Such solutions are in high demand for tasks including fuzzing, symbolic execution, automated test generation, static analysis, code transformation and optimization, and vulnerability detection. The development of such custom domain-specific analysis tools becomes feasible because code-owning companies typically possess substantial engineering resources. These resources can be leveraged in combination with open-source frameworks and ecosystems such as Eclipse, Low Level Virtual Machine, and Microsoft Visual Studio Code. Furthermore, the economic impact is often significant: effective analysis solutions lead to substantial cost savings and quality improvements, which justifies the investment in specialized tool development. The proposed method consists of the following key stages: problem and idea analysis, solution design, requirements development, implementation, testing and validation, deployment and transfer. A central element of this method is the development of code templates that define the subset of the basic programming language used within the codebase. These templates provide a formal foundation for analysis and constrain the variability that tools must support. Notably, requirements development, tool implementation, and validation proceed in parallel, with testing tightly integrated into the development process. Using this method, several tools have been successfully developed, including a static analysis tool for C applications tailored for fuzzing, a clone detection and refactoring tool for a large network device software codebase, and other domain-specific solutions. Each project typically involved between one and three developers and lasted one to two years. While effective, the proposed method is resource-intensive. Analysis of completed projects shows that when a fully specified technical assignment is available from the outset, the required resources for implementing program analysis solutions can be reduced by several times. However, the strength of the method lies in situations where no detailed technical specification can be produced in advance. In such cases, the method enables iterative elicitation of requirements, reducing the risk of developing tools that do not meet real project needs.

Keywords

code analysis, static analysis, large codebases, domain-specific approach, development process model

For citation: Koznov D.V. Domain-specific code analysis approach. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2026, vol. 26, no. 3, pp. 565–573 (in Russian). doi: 10.17586/2226-1494-2026-26-3-565-573

Введение

Многие современные крупные компании в области телекоммуникаций, банковской сферы и др., целевые продукты и сервисы которых включают сложное, многофункциональное программное обеспечение (ПО), заинтересованы в технологической независимости и часто хотят иметь максимально полный набор собственных средств разработки. Это оказывается возможным ввиду развития движения открытых разработок (open source), и, в частности, наличия таких проектов, как Eclipse, Low Level Virtual Machine (LLVM), Microsoft Visual Studio Code (VS Code) и др. На основе этих проектов возможно создать новые среды разработки (frameworks), средства тестирования, средства поиска клонов и анализа уязвимостей [1] и другие.

Дополнительным обстоятельством оказывается наличие у таких компаний больших кодовых баз (миллионы строк кода). Эти кодовые базы оказываются консервативными и статичными — их код работает в десятках систем и его кардинальное изменение затруднено[1]. Для анализа, сопровождения и модификации такого кода требуются средства анализа, которые способны извлекать необходимую информацию и оказывать помощь в оптимизации и тестировании, гарантируя корректность преобразований.

Вместе с тем подобный код, несмотря на свои размеры, использует возможности языков программирования не полностью (например, концепцию языков проектов [3, 4]). При этом используется некоторое подмножество универсального языка программирования, и нерешаемые в общем виде задачи по анализу кода могут быть решены (например, статический поиск динамических массивов в программах на языке С).

[1] Термин legacy означает сложный и объемный программный код действующих систем, имеющий большое бизнес-значение, в разработке которого использованы устаревшие технологии и языки программирования — Turbo Pascal, Power Builder, Fortran, PL/1 и т. д. [2]. Таким образом, подобные системы нельзя демонтировать, но сопровождать и развивать их также весьма непросто. Рассматриваемые в работе большие кодовые базы, в целом, имеют ряд черт legacy, однако в полной мере им не являются (в частности, для реализации телекоммуникационных систем используется язык С, который не устаревает, также не устаревают различные открытые инфраструктуры работы с программами на этом языке).

В настоящей работе предлагается метод для разработки предметно-ориентированных решений по анализу программного кода. Такие решения оказываются востребованы для задач фаззинга, символьного исполнения и генерации тестов а также для автоматических трансформаций/оптимизаций кода. Для больших проектов разработка собственных эффективных средств анализа кода оказывается востребованной, так как компании-владельцы этого кода имеют ресурсы для создания собственных, уникальных решений и мотивацию для этого из-за высокой стоимости целевого проекта.

Несмотря на разнообразие алгоритмов и подходов к анализу кода [5–7], наличие открытых инструментов для создания таких решений, как Eclipse, Clang, LLVM и др., средств реализации предметно-ориентированных языков программирования (JetBrains MPS, Eclipse xText, а также обзор [8]), на настоящий момент отсутствует метод создания предметно-ориентированных средств анализа кода. Разработанный метод может помочь сконцентрировать усилия разработчиков подобных решений для максимально эффективного достижения конечного результата в рамках ограниченных ресурсов и избежать ошибок и просчетов.

Предметно-ориентированный анализ программного кода

Большие кодовые базы являются предметом особого внимания компаний-владельцев. Речь идет о миллионах и десятках миллионов строк кода, которые созданы сотнями разработчиков в течение 10–20 лет, а иногда и в рамках более длительного периода. Этот код работает в десятках и сотнях уже выпущенных на рынок системах или является частью основной инфраструктуры компании, может иметь сложные связи с другими системами.

Как правило, в процессе его разработки компания была целиком сосредоточена на решении своих бизнес-задач — было важно получить первые результаты, завоевать место на рынке, быстро адаптироваться к новым платформам, стандартам и т. д. При этом вопросам архитектуры часто не уделялось должного внимания, в итоге, как это ни странно, большая кодовая база появилась неожиданно, обозначившись в виде ряда нерешаемых в обычном, разработческом режиме, проблем.

Например, выявились важные сценарии работы целевых систем, в которых их быстродействие оказывается неудовлетворительным. Или стало очень трудно добавлять новую функциональность, поскольку оказались «выбранными» лимиты доступной оперативной памяти (а это важно как для сетевых устройств, так и для информационных систем, работающих с большим объемом данных). Или катастрофически понизилось качество некоторых подсистем ввиду сложности кода, недостаточной стройности архитектурных решений и необходимости частых изменений этой подсистемы широкой аудиторией прикладных программистов.

Для решения подобных задач требуются различные средства анализа программного кода. Без таких средств оказывается невозможным сопровождение и развитие больших кодовых баз — от внедрения новых архитектурных решений (например, оптимизации взаимодействия кода на разных языках программирования), до разделения собранной, соединенной вместе разноплановой функциональности (такое разделение может позволить раздельно управлять работой такой функциональности, что, в свою очередь, поможет достичь лучшей производительности в ряде бизнес-сценариев). Также важной задачей является автоматизация тестирования, для успешного решения которой также требуются средства анализа кода.

Большие кодовые базы, созданные на универсальных языках программирования, формируют подмножества этих языков, которые в работах [3, 4] были названы языками проектов. Речь идет не только об использовании и неиспользовании тех или иных конструкций базовых языков, но также и о применении их определенным образом (например, «умных» указателей языка C). Как правило, в больших проектах существует определенная практика использования базового языка, которую соблюдают для удобства тестирования и модификаций системы. Специфическое применение базового языка программирования в конкретном проекте можно задавать набором шаблонов.

В качестве примера рассмотрим задачу поиска динамических массивов в кодовой базе телекоммуникационного ПО, где они активно применяются для анализа сетевых буферов переменной длины. Одним из шаблонов этой кодовой базы, используемым для работы с такими массивами, является следующая функция:

```
void foo(int *a, int n, int i)
{
 if (i < n) *(a + i) = 0;
}
```

Данный шаблон содержит использование динамического массива *a* длиной *n* и является минимизированным фрагментом кода, т. е. реальные C-функции могут содержать значительное количество дополнительных инструкций, которые не имеют отношения к задаче поиска динамических массивов. Подобные шаблоны, исходно заданные архитекторами, помогают не только сузить и упростить анализ кода, но сделать нерешаемую в общем виде задачу — статическое определение динамических массивов в языке C (данная задача неразрешима по причине слабой синтаксической поддержки массивов в языке C, а также наличия адресной арифметики) — разрешимой.

Метод

Рассмотрим описание предлагаемого метода предметно-ориентированного анализа программного кода. Метод состоит из 7 шагов (рисунок).

Шаг 1. Анализ проблемы и идей. Выполняется проверка возможности решить имеющуюся проблему.

Шаг 2. Проектирование. Происходит создание концепции решения.

Шаг 3. Разработка требований. Осуществляется создание необходимых шаблонов кода и выявление других предметно-ориентированных требований к решению.

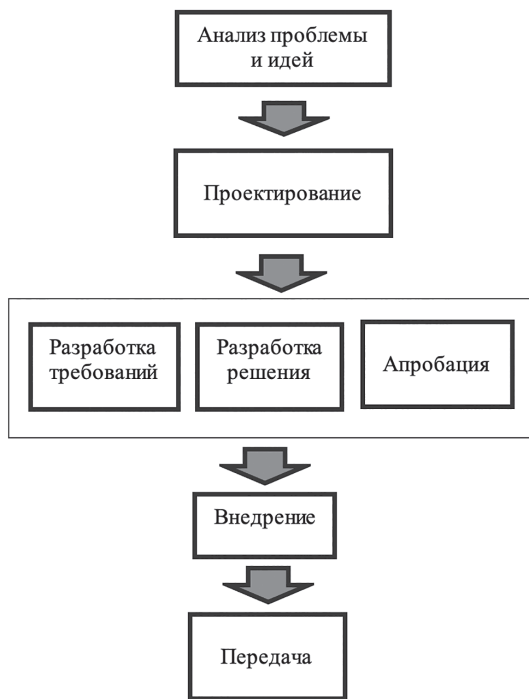

Рисунок. Метод предметно-ориентированного анализа программного кода

Figure. Domain-specific program code analysis method

Шаг 4. Разработка решения.

Шаг 5. Апробация. Проводится пилотное внедрение решения.

Шаг 6. Внедрение. Реализуется передача решения целевой группе пользователей.

Шаг 7. Передача. Происходит организация сопровождения решения.

Опишем шаги подробно.

Анализ проблемы и идей. Шаг 1 является первым при разработке предметно-ориентированных средств анализа программного кода. На данном шаге «встречаются» проблемы разработчиков и идеи по их разрешению. И проблема, и идеи обсуждаются, и если идеи оказываются осуществимыми, то далее выявляются и фиксируются основные предметно-ориентированные свойства будущего решения (их, как правило, немного, их аккуратное выявление критично для успешности разработки решения). Примером такого свойства может быть эффективная обработка решением кода большого объема. Указанные свойства будем называть *базовыми требованиями*, отличая их от низкоуровневых требований, которые будут выявляться на шаге 3. Важным примером базовых требований являются ограничения по быстродействию и потреблению памяти решения, требуемая степень его корректности, безопасности.

Проектирование. На шаге 2 выбираются базовые технологии разработки (как правило, предметно-ориентированные средства анализа программного кода редко создаются с «чистого листа»). Предпочтение отдается открытым технологиям, однако требуется их тщательное изучение и проверка на предмет удовлетворения базовым требованиям решения. В частности, базовое требование по обработке исходных текстов больших объемов часто оказывается вызовом при использовании стандартных инфраструктур (например, создание парсера с помощью технологии xText), и для его успешной реализации необходимо глубоко в них внедряться.

Невзирая на то, что полностью спроектировать решение в самом начале не удается ввиду итеративного выявления требований (в частности, новых шаблонов кода, на которых оно должно работать), максимум исходных, уже имеющихся к этому моменту требований, должны быть учтены заранее. Как правило, грамотное проектирование позволяет избежать существенных переделок решения, хотя полностью от этого застраховаться не удается.

Далее шаги 3–5 выполняются параллельно. Это позволяет значительно снизить риски создания невостребованного решения, а также способствует его бесшовному внедрению в целевой процесс разработки.

Разработка требований. На шаге 3 решается, на каком подмножестве примеров кода будущее решение должно работать. Следует отметить, что эти примеры часто появляются не все и не сразу, поэтому работа по дополнению их списка может происходить параллельно с шагами 3 и 4. Такой подход имеет свои риски — новые примеры могут сильно усложнить разработку инструмента, отбросить ее в начало. В подобных случаях нужно стремиться достичь соглашения между полнотой функциональности, сроками выпуска решения и затратами на его разработку.

Следует подчеркнуть, что итеративная разработка требований к предметно-ориентированному решению является целесообразной, поскольку создать предварительно полные и сбалансированные требования некому, так как разработчики и архитектуры целевого ПО, для которого предназначено решение, не обладают ни соответствующей квалификацией, ни временем. При этом оказывается, что регулярная небольшая активность в течение длительного периода по выявлению требований предпочтительнее (хотя это, без сомнения, сильно зависит от культуры разработки в компании и состояния базового проекта). В пользу такого подхода можно сослаться на консервативный характер больших кодовых баз и текущих проектов, ведущихся на их основе. Таким образом, революционной, бурной активности, свойственной начальным стадиям бизнес-проектов, в таких проектах не наблюдается. Также следует отметить, что в ходе постепенного выявления и обсуждения требований разработчики решения знакомятся с предметной областью, что, зачастую, невозможно сделать иным способом ввиду отсутствия документации, а также из-за сложности и большого объема исходного кода.

Разработка решения. Шаг 4 заключается в реализации требований и архитектуры. Кроме того, в рамках разработки выполняется разработческое тестирование решения — создаются модульные, интеграционные и регрессионные тесты.

Апробация. Шаг 5 необходим для регулярной демонстрации реализованных возможностей решения целевым разработчикам (будущим пользователям решения), а также для сбора и учета полученной обратной связи. Все это значительно снижает риски недопонимания между разработчиками решения и его будущими

пользователями, что особенно важно, когда первые не являются экспертами в предметной области последних (а это случается довольно часто при создании предметно-ориентированных решений в области анализа программного кода). Объем работ по апробации нарастает по мере развития проекта: в начале она выполняется в виде демонстрации новых возможностей самими разработчиками решения на встречах с представителями пользователей, но постепенно новые версии решения начинают передаваться пользователям для экспериментов. В процессе апробации также выполняется пользовательское тестирование.

Следует отметить, что часто встречающееся мнение о том, что мы сначала все сделаем, а лишь потом передадим результат пользователям, имеет много рисков — получившееся решение может не подойти как в большом, так и в малом и быть отвергнутым. Кроме того, участие пользователей в разработке путем постоянной демонстрации и выдачи обратной связи создает эффект привыкания к решению — пользователи могут начать уже во время разработки считать решение своим, что существенно упростит внедрение, снимая психологические барьеры.

Вместе с тем такой подход, совместно с уточняющимися требованиями, может сильно замедлить разработку (предположительно в два и более раза).

Внедрение. Шаг 6 подразумевает, во-первых, согласие пользователей (целевых разработчиков и архитекторов) применять созданное решение и прояснение различных чувствительных моментов, связанных с местом решения в целевом процессе работы, обязательностью и дисциплиной его использования и другими аспектами. Во-вторых, внедрение означает передачу решения пользователям и ответы на их вопросы, исправление пострелизных ошибок, ликвидацию выявленных неудобств и прочее. В больших компаниях на шаге 6 к разработчикам решения может появиться большое количество запросов (от нескольких десятков до нескольких тысяч), и надо быть к этому готовым.

Передача. Шаг 7 осуществляет деликатную деятельность — речь идет о том, что команда разработки решения отдает его целиком на сопровождение пользователям. На данном шаге важно, что предметно-ориентированные решения по анализу кода являются «живыми», к ним могут предъявляться новые требования (в частности, по поддержке новых шаблонов кода). Вместе с тем может оказаться нецелесообразным содержать исходную группу разработчиков для выполнения этой работы. Кроме того, многие команды по созданию целевого ПО хотят иметь все свои инструменты разработки, так сказать, «внутри себя» для уменьшения рисков. По этой причине один или два разработчика со стороны пользователей целесообразно обучить сопровождать созданное решение.

Вместе с тем иногда сопровождение решения так и остается обязанностью подразделения, в рамках которого оно было создано, поскольку пользователи не хотят, например, изучать новый язык программирования (в одном из рассматриваемых проектов целевые разработчики программировали на языке C/C++, а решение было создано на языке Java). Более того, для сопровождения необходимо знакомство с набором разработки решения, а это может оказаться существенно непрофильной активностью для целевой команды.

Комментарии к особенностям работы метода. Необходимо отметить, что тестирование не выделено в отдельный шаг/вид деятельности предложенного метода, поскольку решение должно работать на фиксированном наборе шаблонов кода, что значительно упрощает процесс проверки его работоспособности. Также итеративная апробация позволяет с помощью пользователей найти значительное количество ошибок. Наконец, тестирование выполняется силами самих разработчиков в процессе создания таких решений. Важно подчеркнуть, что требования по качеству предметно-ориентированных средств анализа кода значительно ниже, чем к различным другим продуктам в силу ограниченности бюджета таких проектов, а также доступности команды разработки/поддержки в процессе эксплуатации (т. е. обычно ошибки могут быть всегда оперативно исправлены).

Апробация и ограничения метода

Представленный метод опробован на ряде проектов крупной телекоммуникационной компании и доказал свою эффективность. С его помощью были созданы инструменты для статического анализа C-приложений для нужд фаззинга, создано средство поиска клонов и рефакторинга большой кодовой базы ПО сетевых устройств и некоторые другие решения. В проектах было задействовано от одного до трех разработчиков, длительность проектов составила от одного до двух лет. Также частично данный метод был использован для первичных обсуждений идей подобных проектов и создания первых прототипов в небольших и средних компаниях. Разработанный метод хорошо себя зарекомендовал при реализации индустриальных предметно-ориентированных языков (Domain Specific Languages, DSLs) — данные проекты были существенно сложнее, включали 4–10 разработчиков, и имели длительность от двух до трех лет. При этом в результате проектов по созданию DSLs удалось уменьшить объем целевого кода, с которым имеют дело целевые разработчики, создавая новые продукты на основе старых, более чем в 10 раз, что вызвало большой энтузиазм у разработчиков и способствовало активному внедрению созданных инструментов в производственные процессы. С помощью предложенного метода был создан инструмент рефакторинга Lua-приложений для большой кодовой базы телекоммуникационных систем, позволивший уменьшить объем кода на 5–7 %, что важно, поскольку данный код работает на целевых телекоммуникационных устройствах (сетевых маршрутизаторах и коммутаторах), имеющих жесткие ограничения по объему памяти.

Следует отметить, что невзирая на итеративность метода и поддержку параллельных активностей (шагов 2 и 3, шагов 4 и 5) риски неиспользования результатов все равно сохраняются. С одной стороны, не все лица со стороны пользователей, принимающие решение, были так или иначе вовлечены в разработку решений, с другой стороны, не все результаты оказались удачными.

Необходимо подчеркнуть важность шага 1 предлагаемого метода. На данном шаге при апробации было отклонено большое количество предметно-ориентированных проектов по анализу кода в связи с невозможностью достичь реалистичных результатов. Оказалось, что разработчики часто ставят недостижимые задачи, или не сразу очевиден метод, которым поставленные задачи можно решить.

Анализ реализованных проектов показал, что метод ресурсозатратен, однако при наличии с самого начала полностью готового технического задания можно в несколько раз снизить ресурсы по реализации предметно-ориентированных решений. Не все компании готовы на такие затраты. Более того, для поддержки итеративности и связей с пользователями требуются регулярные совещания-встречи в течение длительного времени (по сути, во время всего течения проекта). Не каждая корпоративная культура может это обеспечить, не каждая целевая команда готова к такой концентрации.

Сравнение и соотнесение

Различные задачи анализа кода активно решаются при разработке компиляторов и Integrated Development Environment (IDE) [7]. Например, варианты алгоритмов анализа указателей для языка C/C++, включая точную сборку мусора [9], задачу вывода типов для языка Python [10] и многие другие. Однако все эти решения являются универсальными, т. е. ориентированы на обработку всех программ на соответствующем языке программирования: даже в том случае, когда универсальность невозможна, в таких решениях требуется обеспечить хорошие результаты для массового пользователя, иначе данная возможность не будет включена в итоговый продукт. Кроме того, решения задач по анализу кода в таких системах почти невозможно повторно использовать вне их контекста, поскольку их эффективность сильно зависит от использования структур данных, а также целевых сервисов, для которых они реализуются.

Для предметно-ориентированных языков программирования [8, 11] диапазон случаев при анализе их программного кода сужается, поскольку, фактически, рассматриваются узкоспециализированные языки, имеющие меньше конструкций, а используемые конструкции содержат менее широкий спектр семантики. Однако и в этом случае требуется полнота в рамках представленного языка, которая, впрочем, на практике часто нарушается — и тогда создаваемые решения попадают в рассматриваемый класс, поскольку работают даже в рамках фиксированного языка, на наборе определенных шаблонов и случаев, причем последние могут не отсекаться аккуратно предметно-ориентированным языком[1]. В контексте предметно-ориентированных языков имеются исследования, нацеленные на упрощение разработки средств статического анализа посредством повторного использования для разных языков [12–14]. Но такой подход в некотором роде противоположен предлагаемому методу, поскольку он пытается обобщать статический анализ, а в настоящей работе он специализирован, и рассмотрены не языки программирования, а более узкий контекст — большие кодовые базы (фактически неявно заданный предметно-ориентированный язык — язык проекта [3, 4]). Имеются также работы, ориентированные на гибкий статический анализ, который допускает пользовательскую адаптацию (user customization) [15, 16], однако и в этом случае акцент делался на создании универсальных статических анализаторов.

Следует также упомянуть про предметно-ориентированное визуальное моделирование [17–19], однако в его рамках задачи по анализу кода ограничиваются обратным проектированием (reverse engineering), а также поддержанием циклической разработки (round-trip engineering), которые оказываются, по факту, предметно-ориентированными (так как не существует универсальных решений). Важно, что данные задачи редко решаются так, чтобы результатами можно было полноценно пользоваться на практике.

Существующие средства анализа программного кода, прежде всего, средства статического анализа [5, 6], обычно имеют многофункциональный программный интерфейс для настройки и интеграции в различные проекты, для решения разных целевых задач. Однако такие средства обычно имеют фиксированный спектр целевых задач, а также различные технологические ограничения — потребление памяти, быстродействие, ограничения по объему обрабатываемых программ и другие. Все это затрудняет повторное использование как самих алгоритмов, так и их реализаций.

Имеются также открытые инфраструктуры, предоставляющие различные средства — парсер и дерево синтаксического разбора программы, то же самое, но в терминах универсального языка типа байт-кода LLVM или IL.NET. Эти средства не имеют ограничений, свойственных предыдущему классу подходов, однако являются инструментами, «кирпичами» для создания предметно-ориентированных средств. Тем не менее и эти средства часто имеют ограничения по работе с большими объемами программ, как например, технология Eclipse.xText. Достоинством таких подходов является их открытость и возможность как выбора уровня абстрактности (например, в LLVM можно работать как на уровне дерева исходной программы, так и на уровне LLVM байт-кода), так и возможность замены различных отдельных функций.

Необходимо отметить, что готовые средства анализа являются все-таки отдельными продуктами с четко выделенным спектром применимости, и хоть и имеют средства настройки на задачу, но слабо подходят для разработки предметно-ориентированных решений. Для этой цели подходят открытые инфраструктуры (например, LLVM, Eclipse, Microsoft Visual Studio Code) — они полезны в качестве инструментов, базовых архитектур и прочего, однако не имеют руководств по

[1] Аккуратность индустриальных предметно-ориентированных языков, создаваемых в рамках одной компании для своих собственных нужд, часто страдает в угоду быстроте реализации и удовлетворению главных требований пользователей.

созданию предметно-ориентированных средств анализа кода, и перед применением требуют экспериментов и часто — существенных затрат на изучение.

Предлагаемый в настоящей работе подход к управлению требованиями отличается от классических [20, 21], поскольку требования формулируются в течение всего жизненного цикла решения и достаточно быстро реализуются, а реализация проверяется на соответствие требованиям, при этом часто выявляются недопонимания с обеих сторон. Таким образом, в рамках предложенного метода не создается спецификации требований: из-за относительно небольшого объема кода решения и активной работы всех сторон требования существуют лишь в головах участников проекта. Еще одной особенностью работы с требованиями в рамках представленного метода — концентрация вокруг шаблонов кода, на которых должно работать решение, что также является новым.

Следует подчеркнуть, что существует большое количество работ по использованию Large Language Model (LLM) для анализа кода. В 2024–2026 гг. исследования по этой теме смещаются от парадигмы «черного ящика» к гибридным схемам, где LLM получает структурированные входные данные от классических статических анализаторов, языковых серверов IDE и т. д., используя их для уточнения вывода.

В [22] предлагается использовать LLM для анализа зависимостей в частичных программах, помогая восстановить недостающий контекст (символы/типы/потоки данных) с тем, чтобы строить более полные зависимости и оценивать риски интеграции разрабатываемых фрагментов кода. В [23] показано, что комбинация LLM с классическими приемами анализа/инструментирования кода может существенно повышать тестовое покрытие. В [24] развивается интеграция искусственного интеллекта и IDE: входные данные для LLM статически дополняют типовой/связывающей информацией (typed holes и др.), что снижает долю некорректных ответов. В [25] предлагаются специальные наборы данных (benchmarks), которые выявляют сильные/слабые стороны LLM в задачах, близких к статическому анализу кода и помогают понять, где нужны формальные гарантии, а где достаточно эвристик. В целом, инструменты анализа кода на основе LLM демонстрируют убедительный исследовательский прогресс. На уровне прототипов LLM уже интегрируются в IDE и другие инструменты разработки в качестве компонент-помощников. Однако для широкого промышленного применения сохраняются следующие системные ограничения [26]. Во-первых, отсутствуют формальные гарантии корректности получаемых результатов, что особенно критично для телекоммуникаций. Во-вторых, ограничена воспроизводимость результатов. Качество вывода существенно зависит от формулировки запроса, объема контекста и версии модели, что затрудняет сертификацию и долговременную поддержку. В-третьих, остаются вопросы масштабируемости и стоимости. Анализ больших кодовых баз требует сложной оркестровки контекста, комбинирования с классическими статическими анализаторами, что усложняет интеграцию в производственные процессы и цепочки программных средств. В-четвертых, не до конца решены вопросы конфиденциальности и интеллектуальной собственности при использовании внешних моделей. Таким образом, текущая стадия в этой области служит переходом от исследовательских прототипов к инженерной адаптации. При этом в силу перспективности гибридных подходов предметно-ориентированный анализ кода может быть впоследствии эффективно интегрирован с LLM. Следует признать, что предметно-ориентированный анализ активно используется при разработке IDE для предметно-ориентированных языков, которые вряд ли могут быть заменены LLM, так как такие языки позволяют значительно повысить уровень абстрактности программирования, уменьшая количество ошибок и объем создаваемого кода. При этом задачи анализа кода в средствах поддержки предметно-ориентированных языков требуют высокой точности, что является пока что серьезным препятствием для LLM, если их использовать в реализации DSL.

Заключение

Предложенный метод в значительной степени опирается на корпоративные особенности, в частности способность работников и процессов компании фокусироваться на выбранной задаче долгое время. Метод оказывается неприменимым в компаниях, подверженных хаосу, или в так называемых «своевольных» компаниях, в которых отдельные коллективы разработчиков имеют большие преференции и свободу воли (компания только следит за тем, чтобы команды обеспечивали определенные бизнес-показатели).

Необходимо отметить, что предметно-ориентированный анализ кода не является панацеей и не может решить все проблемы сопровождения и трансформации больших кодовых баз. Важно понимать, что самым трудным при внедрении инновационных методов в устоявшиеся процессы является изменение способов работы, организационно-административных отношений и новых правил. В случае больших кодовых баз это может быть строгое следование соглашениям кодирования и архитектурным образцам (возможно, с регулярной автоматизированной проверкой), или регуляризация коммуникаций между командой платформы и прикладными командами для уменьшения ошибок при реализации возможностей в рамках продуктов платформы. Вместе с тем ясное описание метода с обсуждением его разных аспектов может помочь разработчикам разных компаний в создании своих собственных методов по созданию средств анализа кода и других инноваций.

Предложенный подход до определенной степени решает вопрос с жизненным циклом средств разработки программного обеспечения. Альтернативой ему является свободный рынок таких средств (почти отсутствующий в Российской Федерации), открытое распространение (требует значительных усилий для организации сообщества). Рассмотренный в работе вариант — внутреннее использование — ограничивает время жизни инструментария (по оценкам автора — до 10 лет, хотя в отдельных случаях может быть и больше), связывает его с авторами (уход авторов из компании приводит к

проблемам по сопровождению), а также несет некоторые дополнительные риски. Тем не менее, этот подход является выходом для компаний, имеющих большие бюджеты на разработку программного обеспечения.

Интересным дальнейшим направлением исследования является создание методики работы с требованиями в рамках представленного в работе метода: целесообразно усилить классификацию требований, исследовать специфику их итеративной разработки, соотнестись с имеющимися моделями и методами, а также оценить влияние итеративности в разработке требований на увеличение ресурсов разработки. Также интересным направлением является расширение метода на случаи создания новых предметно-ориентированных языков программирования и соответствующих инфраструктур поддержки (frameworks).

Перспективным направлением настоящей работы является интеграция предложенного подхода с большими языковыми моделями. Это можно сделать, используя LLM для предметно-ориентированного анализа кода, варьируя параметры анализа и экономя на реализации различных видов статического анализа. Данная интеграция будет возможна, когда соответствующие LLM-ориентированные подходы достигнут необходимой зрелости. Также можно использовать представленный метод для разработки гибридных средств анализа кода, использующих LLM. Необходимо подчеркнуть, что в данной сфере также необходимы дополнительные исследования, в то время как описанный метод посвящен созданию средств разработки, готовых к промышленному применению.

Литература

1. Арутюнян М.С. Статический анализ исходного и исполняемого кода на основе поиска клонов кода. Диссертация на соискание ученой степени кандидата технических наук. М.: ФГБУН Институт системного программирования им. В. П. Иванникова Российской академии наук, 2025. 133 с.
2. Ali M., Hussain S., Ashraf M., Paracha K. Addressing software related issues on legacy systems – a review // International Journal of Scientific & Technology Research. 2020. V. 9. N 3. P. 3738–3742.
3. Boulychev D., Koznov D., Terekhov A.A. On project-specific languages and their application in reengineering // Proc of the 6th European Conference on Software Maintenance and Reengineering. 2002. P. 177–185. doi: 10.1109/csmr.2002.995802
4. Кознов Д.В., Ольхович Л.Б. Визуальные языки проектов // Системное программирование. 2005. Т. 1. С. 148–167.
5. Белеванцев А.А. Многоуровневый статический анализ исходного кода для обеспечения качества программ. Диссертация на соискание ученой степени доктора физ.-мат. наук. М.: ФГБУН Институт системного программирования им. В. П. Иванникова Российской академии наук, 2018. 229 с.
6. Gosain A., Sharma G. Static analysis: A survey of techniques and tools // Advances in Intelligent Systems and Computing. 2015. V. 343. P. 581–591. doi: 10.1007/978-81-322-2268-2_59
7. Muchnick S. Advanced Compiler Design Implementation. Morgan Kaufmann, 1997. 888 p.
8. Negm E., Makady S., Salah A. Survey on domain specific languages implementation aspects // International Journal of Advanced Computer Science and Applications. 2019. V. 10. N 11. doi: 10.14569/ijacsa.2019.0101183
9. Berezun D., Boulytchev D. Precise garbage collection for C++ with a non-cooperative compiler // Proc. of the 10th Central and Eastern European Software Engineering Conference in Russia. 2014. P. 15. doi: 10.1145/2687233.2687244
10. Бронштейн И.Е. Вывод типов для языка Python // Труды Института системного программирования РАН. 2013. Т. 24. С. 161–190.
11. Фаулер М. Предметно-ориентированные языки программирования. М.: Издательский дом Вильямс, 2011. 572 с.
12. Wang J., Huang Y., Wang S., Wang Q. An approach to detecting bugs in pattern-based bug detectors // arXiv. 2021. arXiv:2109.02245. doi: 10.48550/arXiv.2109.02245
13. Mey J., Kühn T., Schöne R., Aßmann U. Reusing static analysis across different domain-specific languages using reference attribute grammars // arXiv. 2020. arXiv:2002.06187. doi: 10.48550/arXiv.2002.06187
14. Zhang X., Zhou Y., Tan S.H. Efficient pattern-based static analysis approach via regular-expression rules // Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2023. P. 132–143. doi: 10.1109/saner56733.2023.00022
15. Mendonça D.S., Kalinowski M. Towards practical reuse of custom static analysis rules for defect localization // Proc. of the 19th

References

1. Arutyunyan M.S. Static Analysis of the Source and Executable Code Based on Code Clone Detection: dissertation for the degree of candidate of technical sciences. Moscow, ISP RAS, 2025, 133 p. (in Russian)
2. Ali M., Hussain S., Ashraf M., Paracha K. Addressing software related issues on legacy systems – a review. International Journal of Scientific & Technology Research, 2020. vol. 9, no. 3, pp. 3738–3742.
3. Boulychev D., Koznov D., Terekhov A.A. On project-specific languages and their application in reengineering. Proc of the 6th European Conference on Software Maintenance and Reengineering, 2002, pp. 177–185. doi: 10.1109/csmr.2002.995802
4. Koznov D.V., Olkhovich L.B. Visual project languages. Systems programming. 2005, vol. 1, pp. 148–167. (in Russian)
5. Belevantsev A.A. Multi-level Static Analysis of the Source Code to Ensure Program Quality: dissertation for the degree of doctoral dissertation in physics and mathematics. Moscow, ISP RAS, 2018, 229 p. (in Russian)
6. Gosain A., Sharma G. Static analysis: A survey of techniques and tools. Advances in Intelligent Systems and Computing, 2015, vol. 343, pp. 581–591. doi: 10.1007/978-81-322-2268-2_59
7. Muchnick S. Advanced Compiler Design Implementation. Morgan Kaufmann, 1997, 888 p.
8. Negm E., Makady S., Salah A. Survey on domain specific languages implementation aspects. International Journal of Advanced Computer Science and Applications, 2019, vol. 10, no. 11, doi: 10.14569/ijacsa.2019.0101183
9. Berezun D., Boulytchev D. Precise garbage collection for C++ with a non-cooperative compiler. Proc. of the 10th Central and Eastern European Software Engineering Conference in Russia, 2014, pp. 15. doi: 10.1145/2687233.2687244
10. Bronstein I.E. Type deduction for Python. Proceedings of the Institute for System Programming of the RAS, 2013, vol. 24, pp. 161–190. (in Russian)
11. Fowler M. Domain-Specific Languages. Addison-Wesley Professional, 2010, 640 p.
12. Wang J., Huang Y., Wang S., Wang Q. An approach to detecting bugs in pattern-based bug detectors. arXiv, 2021. arXiv:2109.02245. doi: 10.48550/arXiv.2109.02245
13. Mey J., Kühn T., Schöne R., Aßmann U. Reusing static analysis across different domain-specific languages using reference attribute grammars. arXiv, 2020. arXiv:2002.06187. doi: 10.48550/arXiv.2002.06187
14. Zhang X., Zhou Y., Tan S.H. Efficient pattern-based static analysis approach via regular-expression rules. Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2023, pp. 132–143. doi: 10.1109/saner56733.2023.00022
15. Mendonça D.S., Kalinowski M. Towards practical reuse of custom static analysis rules for defect localization. Proc. of the 19th Brazilian Symposium on Software Quality, 2020, pp. 1–10. doi: 10.1145/3439961.3439985

Brazilian Symposium on Software Quality. 2020. P. 1–10. doi: 10.1145/3439961.3439985
16. Li Z., Liu Z., Wong W.K., Ma P., Wang S. Evaluating C/C++ vulnerability detectability of query-based static application security testing tools // IEEE Transactions on Dependable and Secure Computing. 2024. V. 21. N 5. P. 4600–4618. doi: 10.1109/tdsc.2024.3354789
17. Кознов Д.В. Методология и инструментарий предметно-ориентированного моделирования. Диссертация на соискание ученой степени доктора технических наук. СПб.: СПбГУ, 2016. 430 с.
18. Терехов А.Н., Брыксин Т.А., Литвинов Ю.В. QReal: платформа визуального предметно-ориентированного моделирования // Программная инженерия. 2013. № 6. С. 11–19.
19. Kelly S., Tolvanen J.-P. Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, 2008. 444 p.
20. Вигерс К.И. Разработка требований к программному обеспечению. М.: ИТД «Русская Редакция», 2004. 576 с.
21. Липаев В.В. Тестирование крупных комплексов программ на соответствие требованиям // Бизнес-информатика. 2008. № 2 (4). С. 16–24.
22. Rong X., Yadavally A., Nguyen T.N. Large Language Model-aided partial program dependence analysis // Proc. of the IEEE/ACM 48th International Conference on Software Engineering (ICSE ’26). 2026.
23. Yang C., Chen J., Lin B., Wang Z., Zhou J. Advancing code coverage: incorporating program analysis with Large Language Models // ACM Transactions on Software Engineering and Methodology (TOSEM). 2026. V. 35. N 5. P. 1–31. doi: 10.1145/3748505
24. Blinn A., Li X., Kim J.H., Omar C. Statically contextualizing Large Language Models with typed holes // Proc. of the ACM on Programming Languages. 2024. V. 8. N OOPSLA2. P. 468–498. doi: 10.1145/3689728
25. Venkatesh A.P.S., Sabu S., Mir A.M., Reis S., Bodden E. The emergence of Large Language Models in static analysis: a first look through micro-benchmarks // Proc. of the IEEE/ACM First International Conference on AI Foundation Models and Software Engineering. 2024. P. 35–39. doi: 10.1145/3650105.3652288
26. Sun W., Miao Y., Li Y., Zhang H., Fang C., Liu Y., et al. Source code summarization in the era of Large Language Models // Proc. of the IEEE/ACM 47th International Conference on Software Engineering (ICSE). 2025. P. 1882–1894. doi: 10.1109/ICSE55347.2025.00034

16. Li Z, Liu Z., Wong W.K., Ma P., Wang S. Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. IEEE Transactions on Dependable and Secure Computing, 2024, vol. 21, no. 5, pp. 4600–4618. doi: 10.1109/tdsc.2024.3354789
17. Koznov D.V. Methodology and Tools for Domain-specific Modeling: dissertation for the degree of doctor of technical sciences. St. Petersburg, SPbU, 2016, 430 p. (in Russian)
18. Terekhov A.N., Bryksin T.A., Litvinov Yu.V. QReal: A platform for visual, domain-specific modeling. Programmnaya Ingeneria, 2013, no. 6, pp. 11–19. (in Russian)
19. Kelly S., Tolvanen J.-P. Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, 2008, 444 p.
20. Wiegers K. Software Requirements. Microsoft Press, 1999, 350 p.
21. Lipaev V.V. Large software systems testing for conformance requirements. Biznes-Informatika, 2008, no. 2 (4), pp. 16–24. (in Russian)
22. Rong X., Yadavally A., Nguyen T.N. Large Language Model-aided partial program dependence analysis. Proc. of the IEEE/ACM 48th International Conference on Software Engineering (ICSE ’26), 2026.
23. Yang C., Chen J., Lin B., Wang Z., Zhou J. Advancing code coverage: incorporating program analysis with Large Language Models. ACM Transactions on Software Engineering and Methodology (TOSEM), 2026, vol. 35, no. 5, pp. 1–31. doi: 10.1145/3748505
24. Blinn A., Li X., Kim J.H., Omar C. Statically contextualizing Large Language Models with typed holes. Proc. of the ACM on Programming Languages, 2024, vol. 8, no. OOPSLA2. pp. 468–498. doi: 10.1145/3689728
25. Venkatesh A.P.S., Sabu S., Mir A.M., Reis S., Bodden E. The emergence of Large Language Models in static analysis: a first look through micro-benchmarks. Proc. of the IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, 2024, pp. 35–39. doi: 10.1145/3650105.3652288
26. Sun W., Miao Y., Li Y., Zhang H., Fang C., Liu Y., et al. Source code summarization in the era of Large Language Models. Proc. of the IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 1882–1894. doi: 10.1109/ICSE55347.2025.00034

Автор

Кознов Дмитрий Владимирович — доктор технических наук, доцент, профессор, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, sc 8885649400, https://orcid.org/0000-0003-2632-3193, d.koznov@spbu.ru

Author

Dmitry V. Koznov — D.Sc., Associate Professor, Professor, St. Petersburg State University, Saint Petersburg, 199034, Russian Federation, sc 8885649400, https://orcid.org/0000-0003-2632-3193, d.koznov@spbu.ru

Статья поступила в редакцию 28.01.2026
Одобрена после рецензирования 29.04.2026
Принята к печати 25.05.2026

Received 28.01.2026
Approved after reviewing 29.04.2026
Accepted 25.05.2026

Работа доступна по лицензии
Creative Commons
«Attribution-NonCommercial»