На правах рукописи

Головешкин Алексей Валерьевич

Устойчивая алгоритмическая привязка к коду программы

Специальность 2.3.5 —
Математическое и программное обеспечение
вычислительных систем, комплексов и компьютерных сетей

АВТОРЕФЕРАТ
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2022

Работа выполнена в Федеральном государственном автономном образовательном учреждении высшего образования «Южный федеральный университет».

Научный руководитель: **Михалкович Станислав Станиславович**
кандидат физико-математических наук, доцент

Официальные оппоненты: **Легалов Александр Иванович**
доктор технических наук, профессор, профессор департамента программной инженерии факультета компьютерных наук ФГАОУ ВО «Национальный исследовательский университет «Высшая школа экономики»

Зуев Евгений Александрович
кандидат физико-математических наук, руководитель лаборатории операционных систем, языков программирования и компиляторов АНО ВО «Университет Иннополис»

Ведущая организация: Федеральное государственное учреждение «Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша Российской академии наук»

Защита диссертации состоится «15» декабря 2022 года в 15:00 на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В.П. Иванникова Российской академии наук» по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки «Институт системного программирования им. В.П. Иванникова Российской академии наук».

Отзывы на автореферат в двух экземплярах, заверенные печатью учреждения, просьба направлять по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25, учёному секретарю диссертационного совета 24.1.120.01.

Автореферат разослан «____» _____ 2022 года.

Ученый секретарь
диссертационного совета
24.1.120.01,
кандидат физ.-мат. наук — Зеленов Сергей Вадимович

Общая характеристика работы

Актуальность темы исследования. Многочисленные эксперименты и наблюдения, описанные в научной литературе, показывают, что изучение кода является основной активностью программиста при работе над задачей и занимает до 70% рабочего времени. Изучая код, программист находит фрагменты, относящиеся к задаче, после чего наиболее активно перемещается по коду в пределах найденного набора фрагментов. Потеря информации о найденных фрагментах замедляет работу, поскольку поиск приходится проводить заново. Совокупность фрагментов кода, решающих общую задачу, в научной литературе принято называть словом *функциональность* (concern). Функциональности, которые реализуются фрагментами кода, рассредоточенными между элементами приоритетной декомпозиции программы (классами, методами в случае ООП), называются *сквозными* или *прорезающими* (crosscutting). Прорезающие функциональности присутствуют в большинетве проектов. Например, при разработке компилятора реализация каждой языковой конструкции прорезает грамматику и классы, отвечающие за этапы компиляции, а при разработке веб-сайта реализация очередной возможности, предоставляемой пользователю, может прорезать представления, контроллеры и сервисы.

Результаты, полученные в настоящей диссертационной работе, позволяют сохранить информацию об участках кода, относящихся к прорезающей задаче-функциональности, связать с каждым из них пометку-закладку и по требованию найти текущие версии этих участков в отредактированной программе.

Ключевыми понятиями, вводимыми в работе, являются понятия привязки и переприязки. *Привязка* — это построение и сохранение описания участка кода в виде набора структур специального вида, называемых *контекстами*. *Переприязка* — поиск актуальной версии этого участка по его описанию, сохранённому в момент привязки, и повторная привязка к найденному участку. Посредством разработанного нами инструмента с участками кода, к которым произведена привязка, можно связать пометки-закладки. Так формируется *разметка* кода — совокупность пометок, связанных с участками кода, относящимися к прорезающим функциональностям. Пометки, относящиеся к одной функциональности, предполагается объединять в группу. С каждой пометкой и группой может быть связан некоторый комментарий. Таким образом разметка задаёт семантическое (смысловое) описание функциональностей, представленных в коде. Разметка хранится вместе с проектом, но отделена от текста программы. Требование *устойчивости* привязки означает устойчивость к редактированию кода: переприязка должна успешно производиться всегда независимо от того, как поменялся сам участок кода или его окружение.

Построение контекстов, описывающих участок, к которому производится привязка, осуществляется по легковесному абстрактному синтаксическому дереву (АСД) программы, захватывающему достаточный для построения контекстов структурный «каркас» программы — информацию о крупных синтаксиче-

ских сущностях. Под синтаксической сущностью (синтаксическим элементом) будем понимать часть программы, соответствующую некоторому нетерминальному символу грамматики языка (например, в ООП-программе крупными синтаксическими сущностями будут пространства имён, классы, члены классов). Легковесное дерево можно построить в ходе *легковесного парсинга* — варианта синтаксического анализа, при котором подробно анализируются только нужные разработчику парсера части программы. В настоящем диссертационном исследовании предлагается метод легковесного парсинга на основе *упрощённых грамматик*. В них вместо описания структуры неважных участков кода предлагается использовать специальный терминальный символ Any. Актуальность метода связана не только с задачей устойчивой привязки к коду программы, но и с остальными задачами, в которых применяется легковесный парсинг (реверс-инжиниринг, обработка неправильных программ, обработка вставок кода на другом языке и др.).

Степень разработанности темы исследования. Основополагающими работами отечественных учёных, посвящёнными вопросам привязки и прорезающих функциональностей, являются труды А. Л. Фуксмана (1979) и М. М. Горбунова-Посадова (1996, 1999, 2000). В них предлагается разделять программу на некоторую базовую версию («основу») и сосредоточенно описываемые «расширяющие функции» или «вертикальные слои», инкапсулирующие в себе прорезающие функциональности. Оба автора считают, что прорезающие функциональности должны встраиваться в нужные места базовой программы при генерации интегрированной программы. Привязываться к этим местам предполагается по номеру строки и столбца, а также с помощью поиска подстрок, что абсолютно неустойчиво к редактированию основы.

Идея разделения основной программы и прорезающих функциональностей воплощается в прикладных парадигмах аспектно-ориентированного (Г. Кичалес и др., 1997), функциональностно-ориентированного (Feature-Oriented) (К. Прехофер, 1997; С. Апель и др., 2013), дельта-ориентированного (И. Шефер и др., 2010; Л. Камарго и др., 2021) программирования. В них способы привязки к местам в основной программе также неустойчивы к редактированию: запоминаются имена синтаксических сущностей и сигнатуры методов.

На практике полностью избавиться от прорезающих функциональностей, вплетённых в код проекта, невозможно: многие из них слишком сложно (как идейно, так и физически) отделить от «основной» логики программы. Активно ведутся научные исследования того, как программисты работают с интегрированной программой и сохраняют знание о предназначении её участков (М. Робиллард и др., 2004; Р. Минелли и др., 2014, 2015; Д. Рубин и др., 2016; П. Ригби и др., 2016; Г. Ди Роза и др., 2020; М. Шреер и др., 2021). Обозначаются следующие проблемы: программисты тратят большую часть рабочего времени на изучение кода, а не на редактирование; накопленная в ходе изучения информация теряется при переключении между задачами; информация о функциональностях постепенно «утекает» из проекта, что обусловлено и свой-

ствами человеческой памяти, и уходом разработчиков, и нехваткой времени на актуализацию документации. Разметка кода, основанная на устойчивой привязке, помогает справиться со всем перечисленным.

Устойчивая привязка к коду имеет свою специфику, но является родственной таким хорошо исследованным вопросам, как поиск клонов кода (Т. Камия и др., 2002; Л. Цзян и др., 2007; Ч. К. Рой и др., 2009; В. Ванг и др., 2020), поиск плагиата (М. Джой и др., 1999; К. В. Бойер и др., 1999; Л. Пречельт и др., 2002; Х. Чирз и др., 2021), восстановление сценариев редактирования программы (Ж.-Р. Фаллери и др., 2014; Г. Дотцлер и др., 2016; В. Фрик и др., 2018). Неалгоритмическая (основанная на псевдокомментариях) привязка к фрагментам кода применяется в работах А. Н. Литвиненко и др. (2010), в CASE-системе Rational Rose (Rational Rhapsody), в инструменте генерации кода Eclipse Modeling Framework. В качестве базы в настоящем диссертационном исследовании используются работы М. С. Малёванного и С. С. Михалковича (2016), непосредственно посвящённые проблеме устойчивой алгоритмической привязки.

Предлагаемая в настоящем исследовании концепция упрощённых грамматик развивает подход к осуществлению легковесного парсинга, именуемый «островными грамматиками» (А. ван Дерсен, 1999; Л. Мунен, 2001), а также концепцию специального символа *Any* (Х. Мессенбек, 1990; М. С. Малёванный, 2015) и схожую с ней концепцию «ограниченных морей» (Я. Курш и др., 2015). При этом используются такие фундаментальные научные результаты, как подход к спецификации формальных языков, разработанный Н. Хомским (1956, 1959), метод нисходящего синтаксического анализа LL(1), описанный Ф. Льюисом и Р. Стирнзом (1968), метод восходящего синтаксического анализа LR(1), введённый Д. Кнутом (1965).

Объект и предмет исследования. Объектом исследования является структурированный текст, в частности код на языке программирования. Предметом исследования являются методы его легковесного синтаксического анализа на основе островных грамматик и методы устойчивой алгоритмической привязки к такому тексту.

Цели и задачи работы. Целью работы является создание метода устойчивой алгоритмической привязки к коду на языке программирования и последующая разработка инструмента разметки кода на основе данного метода. Инструмент разметки предполагается использовать в задачах разработки и сопровождения программного обеспечения, чтобы сократить время, необходимое для поиска участков кода, важных для текущей решаемой проблемы.

Для достижения цели решаются следующие задачи.

1. Разработка теории легковесных грамматик с символом *Any* и упрощённых грамматик.
2. Разработка и алгоритмическая реализация методов LL(1) и LR(1) синтаксического анализа, позволяющих осуществить разбор программы на основе упрощённой грамматики.

3. Разработка генератора легковесных парсеров и специализированного языка для описания упрощённых грамматик.
4. Разработка нескольких упрощённых грамматик и экспериментальная проверка сгенерированных легковесных парсеров.
5. Разработка и реализация моделей и методов, позволяющих осуществлять привязку к синтаксическим сущностям программы и их последующий поиск в изменившемся коде.
6. Разработка и алгоритмическая реализация методов, позволяющих осуществлять привязку к произвольным участкам кода программы и их последующий поиск в изменившемся коде.
7. Разработка инструмента с графическим интерфейсом, использующего указанные модели и алгоритмы для предоставления программисту возможности разметить код программы.
8. Проведение экспериментов для оценки устойчивости привязки.

Соответствие паспорту специальности. Цели и задачи диссертационного исследования соответствуют направлениям исследований, предусмотренным паспортом специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей». Область исследования настоящей диссертации включает в себя модели, методы и алгоритмы анализа программ (пункт 1 паспорта специальности), языки программирования и семантику программ (пункт 2), модели, методы, архитектуры, алгоритмы, языки и программные инструменты организации взаимодействия программ и программных систем (пункт 3).

Методология и методы исследования. В процессе решения задачи легковесного парсинга автор опирается на методы теории алгоритмов и теории формальных языков и грамматик. При решении задачи устойчивой привязки используются методы теории информации. Предлагаемые модели хранения контекстов формализуются в терминах теории множеств. Для проверки полученных результатов проводятся вычислительные эксперименты. Доказательства завершимости и корректности предложенных алгоритмов проводятся с применением теории формальных языков и теории сложности алгоритмов.

Научная новизна.

1. Предложен метод легковесного парсинга на основе упрощённых грамматик, развивающий метод островных грамматик путём задания строгой конструктивно сформулированной связи между «полной» грамматикой G и упрощённой грамматикой G_s. Наличие такой связи позволяет модифицировать для легковесного парсинга на основе упрощённых грамматик методы LL(1) и LR(1) синтаксического анализа. В отличие от других методов легковесного парсинга, для упрощённых грамматик доказано, что легковесный парсер, сгенерированный по грамматике G_s, упрощённой относительно LL(1) грамматики G, распознаёт любую правильную программу

из языка $L(G)$. Для случая LR(1) сформулированы существенные условия успешного разбора.

2. Разработан метод устойчивой алгоритмической привязки к синтаксическим элементам кода, отличающийся от других методов «запоминания» места в коде тем, что предназначен для применения к изменяющейся программе и предлагает новые модели контекстов, описывающие элемент, к которому производится привязка, и новый алгоритм переприязки, осуществляющий поиск элемента в отредактированной программе на основе ранее построенных для него контекстов. Поиск происходит успешно, даже если в программу внесено большое количество правок. По сравнению с ближайшим аналогом метод производит в 2 раза меньше ошибочных переприязок и в 3 раза реже требует производить переприязку вручную.

3. Предложен метод выделения многострочных фрагментов кода и встраивания соответствующих им узлов в АСД, сохраняющий корректность АСД. Он уточняет и формализует подход, применяемый в инструментах, полагающихся на использование псевдокомментариев для обрамления участков кода. Во множестве участков кода задаётся подмножество многострочных фрагментов, выделение которых согласуется с синтаксической структурой программы; метод устойчивой привязки к синтаксическим элементам программы обобщается на такие фрагменты.

Теоретическая значимость. Результаты, полученные при решении задачи легковесного парсинга, могут быть использованы для развития парсинга на основе островных грамматик, в частности для построения более совершенных алгоритмов легковесного парсинга с доказанной корректностью.

Предложенные модели и алгоритмы устойчивой привязки могут быть использованы для построения более специализированных моделей и алгоритмов, в том числе учитывающих информацию об отношениях между элементами функциональностей и информацию о конкретном сценарии редактирования кода.

Практическая значимость. Реализованные алгоритмы легковесного парсинга с символом *Any*, генератор легковесных парсеров LanD, язык описания легковесных грамматик позволяют создавать легковесные грамматики языков, в 20–50 раз более короткие по сравнению с полными грамматиками. Генерируемые парсеры могут использоваться для широкого круга задач, таких как реверс-инжиниринг, обработка вставок кода на другом языке, разбор незавершённого кода и кода с ошибками, собственно задача разметки кода.

Реализованный инструмент разметки кода позволяет более чем в 100 раз сократить время, необходимое для последующих переходов к однажды помеченным фрагментам кода. Разметка кода по ходу разработки позволяет осуществить долгосрочное документирование кода. Посредством разметки при коллективной работе над проектом могут происходить обмен информацией в команде, работающей над одной задачей, и выдача заданий, связанных с модификацией кода или добавлением кода в существующие места программы. Также

разметка кода может быть использована в процессе обучения программированию для выдачи заданий и последующей их проверки.

Инструмент разметки интегрирован в среды Visual Studio и YACC MC, результаты работы внедрены в четырёх организациях, занимающихся разработкой программного обеспечения (ООО «Кассир-софт», ООО «Чек-онлайн», ФГАНУ НИИ «Спецвузавтоматика», ИП Юрушкин Михаил Викторович), а также используются при разработке компилятора языка PascalABC.NET.

Степень достоверности и апробация результатов работы. Достоверность результатов исследования обеспечивается использованием формальных методов исследуемой области, математической строгостью изложения.

Для модифицированного алгоритма LL(1) синтаксического анализа доказано, что парсер, сгенерированный по упрощённой грамматике и использующий данный алгоритм, успешно разбирает программу, порождённую «полной» грамматикой. Для алгоритма встраивания в АСД узлов, соответствующих произвольным фрагментам кода, доказано сохранение корректности АСД.

Практическая применимость предложенных алгоритмов синтаксического анализа и алгоритма перепривязки подтверждена результатами вычислительных экспериментов.

Основные результаты работы докладывались на следующих международных, всероссийских, региональных конференциях и семинарах:

- Всероссийская научная конференция «Современные информационные технологии: тенденции и перспективы развития» — Ростов-на-Дону, **2018**, **2019**, **2021**;
- Международная научная конференция молодых ученых по программной инженерии (Spring/Summer Young Researchers' Colloquium on Software Engineering) — Великий Новгород, **2018**; Саратов, **2019**;
- Семинар кафедры алгебры и дискретной математики Мехмата ЮФУ — Ростов-на-Дону, **2019**;
- Всероссийская научная конференция «Научный сервис в сети Интернет» — Новороссийск, **2019**;
- International Young Scientists Conference on Computational Science — Online, **2021**;
- Семинар о развитии открытой распараллеливающей системы (OPC), Мехмат ЮФУ — Ростов-на-Дону, **2021**;
- Семинар кафедры информатики и вычислительного эксперимента Мехмата ЮФУ — Ростов-на-Дону, **2021**;
- Национальный Суперкомпьютерный Форум — Переславль-Залесский, **2021**.

Личный вклад автора. Все выносимые на защиту результаты получены лично автором диссертации. В совместных работах научному руководителю С. С. Михалковичу принадлежат постановка задач, определение основных направлений исследования и общее руководство.

Основные положения, выносимые на защиту:

1. Доказано для случая LL(1) грамматик, что легковесный синтаксический анализатор, сгенерированный по грамматике G_s, упрощённой относительно грамматики G, разбирает все программы, удовлетворяющие грамматике G. Для случаев LL(1) и LR(1) доказано, что легковесная грамматика G' с символом Any является упрощённой для некоторого множества «полных» грамматик G.
2. Разработаны эффективные модели хранения контекстов и алгоритмы устойчивой привязки к меняющемуся коду, позволяющие осуществлять правильную переприв язку в изменённом коде крупных промышленных проектов с вероятностью, близкой к 100%.
3. Приведены формальные условия для привязки к многострочному фрагменту кода. Доказано, что при соблюдении этих условий узел, соответствующий многострочному фрагменту, можно корректно встроить в абстрактное синтаксическое дерево программы.
4. Разработаны программные комплексы для генерации легковесных парсеров по упрощённым грамматикам и для привязки и переприв язки к меняющемуся коду, позволяющие экспериментально подтвердить эффективность моделей хранения контекстов и алгоритмов устойчивой переприв язки.

Объём и структура работы. Работа состоит из введения, пяти глав, заключения и списка литературы. Объем работы составляет 170 страниц текста с 21 рисунком и 12 таблицами. Список литературы содержит 162 наименования.

Содержание работы

Во **введении** обосновывается актуальность исследований, выполненных в рамках данной диссертационной работы; на основе анализа отечественной и зарубежной научной литературы определяется степень разработанности темы, формулируются цели и задачи исследования, научная новизна, научная и практическая значимость работы, приводится информация об апробации и о публикациях автора по теме исследования, формулируются положения, выносимые на защиту.

В **первой главе** проводится обзор предметной области. Анализируются работы, подтверждающие справедливость мотивации настоящего диссертационного исследования: посвящённые проблематике прорезающих функциональностей, вопросам их инкапсуляции и поиска в интегрированной программе, а также разностороннему исследованию поведения программиста в процессе разработки программы, в частности, в процессе определения участков кода, релевантных относительно решаемой задачи. Рассматриваются публикации в областях, непосредственно развиваемых в настоящей диссертации: посвящённые

поиску кода в задачах эволюции программного обеспечения и легковесному синтаксическому анализу.

Во **второй главе** решается задача легковесного парсинга программы. В задаче устойчивой привязки использование легковесного парсинга позволяет быстро прототипировать парсеры для языков, к коду на которых необходимо привязаться, и даже для конкретных размечаемых проектов. Важным преимуществом легковесного парсинга перед полным является то, что легковесный парсинг позволяет разбирать неправильные программы, в том числе находящиеся в процессе редактирования. Кроме того, самостоятельная генерация легковесных парсеров для разных языков позволяет работать с деревьями, имеющими единый языконезависимый формат.

Вводятся определения легковесной LL(1) грамматики с символом Any и упрощённой LL(1) грамматики. При этом применяются следующие обозначения:

- для грамматики $G = (N, T, P, S)$ N — множество нетерминальных символов, T — множество терминальных символов, P — множество продукций (правил вывода), $S \in N$ — стартовый символ;
- маленькими греческими буквами обозначаются, если не указано другое, последовательности символов из $(N \cup T)^*$ для той грамматики, в которой происходит порождение;
- через $\mathrm{lhs}(p)$ и $\mathrm{rhs}(p)$ обозначаются левая и правая части продукции p соответственно, запись $x \in \mathrm{rhs}(p)$, где $x \in N \cup T$, означает, что $\mathrm{rhs}(p) = \alpha x \beta$;
- через $\mathrm{SYMBOLS}(\gamma)$ обозначается множество терминальных символов, необходимых для записи всех ω: $\gamma \overset{*}{\Rightarrow} \omega, \omega \in T^*$;
- через Any^+ обозначается последовательность из одного и более символа Any.

Определение 1. Легковесной LL(1) грамматикой с символом Any будем называть LL(1) (LR(1)) грамматику $G = (N, T, P, S)$, удовлетворяющую следующим условиям:

1. $Any \in T$.
2. $\exists p \in P : Any \in \mathrm{rhs}(p)$.
3. Если является допустимым вывод $S \overset{*}{\Rightarrow} \alpha Any A \beta$, где $A \in N$, то в P не существуют одновременно такие различные продукции p_1 и p_2, $\mathrm{lhs}(p_1) = \mathrm{lhs}(p_2) = A$, что для некоторого $a \in T \backslash \{Any\}$ допустимы выводы $\mathrm{rhs}(p_1)\beta \overset{*}{\Rightarrow} Any^+ a \delta_2$ и $\mathrm{rhs}(p_2)\beta \overset{*}{\Rightarrow} a\delta_1$. Исключением является случай, когда p_1 имеет вид $A \to AnyA$ и p_2 имеет вид $A \to a\delta$.

Определение 2. Пусть $G = (N, T, P, S)$ — некоторая LL(1) грамматика, $Any \notin T$. Упрощённой относительно G будем называть легковесную LL(1) грамматику с символом Any $G_s = (N_s, T_s, P_s, S_s)$, удовлетворяющую следующим условиям:

1. $S_s = S$;
2. $T_s = T \cup \{Any\}$;
3. $P_s = \{p \in f(P) \mid \text{lhs}(p) = S_s \vee \exists p' \in P_s\colon\ \text{lhs}(p) \in \text{rhs}(p')\}$, где отображение $f : P \to \{p = A \to \alpha \mid A \in N, \alpha \in (N \cup T \cup \{Any\})^*\}$ удовлетворяет условиям:
   а. $\exists P' \subseteq P\colon P' = \{p \in P \mid f(p) \neq p\}$.
   б. $\forall p \in P \setminus P',\ f(p) = p$.
   в. $\forall p \in P',\ \exists n \in \mathbb{N}\colon$ p представима в виде $A \to \alpha_1\gamma_1\alpha_2\gamma_2...\alpha_{n+1}$ и $f(p)$ представима в виде $A \to \alpha_1 Any \alpha_2 Any...\alpha_{n+1}$.
   г. Обозначим через Any_i вхождение Any, возникшее как результат замены некоторого γ_i. Порождение $S_s \overset{*}{\Rightarrow} \alpha Any_{i_1} Any_{i_2}...Any_{i_n} b\beta$, где $i_1, i_2, ..., i_n, n \in \mathbb{N}$, $b \in T_s \setminus \{Any\}$, не является допустимым в G_s, если $b \in \text{SYMBOLS}(\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n})$.
4. $N_s = \{A \in N \mid \exists p \in P_s\colon\ \text{lhs}(p) = A\}$.

Также в работе даётся определение легковесной LR(1) грамматики с символом Any и упрощённой LR(1) грамматики.

Упрощённая грамматика описывается на разработанном автором языке LanD и передаётся на вход одноимённому генератору парсеров. Парсер, сгенерированный по упрощённой грамматике G_s, предполагается использовать для разбора программ из $L(G)$, однако очевидно, что существуют программы из $L(G)$, не порождаемые грамматикой G_s. В главе описываются модификации методов LL(1) и LR(1) синтаксического анализа, благодаря которым по мере разбора программы парсер трактует некоторые последовательности токенов как Any, тем самым «на лету» переходя от программы из $L(G)$ к программе из $L(G_s)$. За счёт этого успешно разбираются правильные относительно G программы и, возможно, некоторые неправильные. Доказываются следующие утверждения.

Утверждение 1. Синтаксический анализатор, сгенерированный по грамматике G_s, упрощённой относительно LL(1) грамматики G, успешно распознаёт любую программу $\omega \in L(G)$.

Утверждение 2. Пусть $G' = (N', T', P', S')$ — легковесная LL(1) грамматика с символом Any. G' является грамматикой, упрощённой относительно любой LL(1) грамматики $G = (N, T, P, S)$, удовлетворяющей следующим условиям:

1. $S = S'$.
2. $T = T' \setminus \{Any\}$.
3. $N' \subseteq N$.
4. Пусть $\mathfrak{P} = \{p \in P | \text{lhs}(p) \in N'\}$. $\forall p \in \mathfrak{P}, \exists! p' \in P'$ такая, что $p = p'$ или $p' = A \to \alpha_1 Any \alpha_2 Any...\alpha_{n+1}$ и $p = A \to \alpha_1\gamma_1\alpha_2\gamma_2...\alpha_{n+1}$, где $A \in N'$, $\alpha_i \in (N' \cup T)^*$, $\gamma_i \in (N \cup T)^*$, $n \in \mathbb{N}$.

5. $\forall p' \in P', \exists p \in P$ такая, что $p = p'$ или $p = A \to \alpha_1\gamma_1\alpha_2\gamma_2...\alpha_{n+1}$ и $p' = A \to \alpha_1 Any \alpha_2 Any ... \alpha_{n+1}$, где $A \in N'$, $\alpha_i \in (N' \cup T)^*$, $\gamma_i \in (N \cup T)^*$, $n \in \mathbb{N}$.
6. Пронумеруем все вхождения Any в правила грамматики G' и все γ, соответствующие им в G. Если порождение $S' \overset{*}{\Rightarrow} \alpha Any_{i_1} Any_{i_2} ... Any_{i_n} b\beta$, где $i_1, i_2, ..., i_n, n \in \mathbb{N}$, $b \in T' \setminus \{Any\}$, является допустимым в G', то $b \notin \text{SYMBOLS}(\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n})$.

Существование такой грамматики G очевидно в случае, когда во множестве T' есть токены, не участвующие в правилах грамматики G' в явном виде, что является одним из определяющих признаков легковесности.

Для упрощённой LR(1) грамматики и соответствующего модифицированного алгоритма парсинга формулируются существенные условия успешного разбора программы из полного языка парсером, сгенерированным по упрощённой грамматике, а также доказывается утверждение, аналогичное утверждению 2.

Введённые определения, предложенные алгоритмы и доказанные утверждения в совокупности подтверждают корректность избранного нами подхода к разработке легковесных парсеров. Легковесная грамматика G' с символом Any разрабатывается с нуля и итеративно улучшается по результатам проверки очередного сгенерированного парсера с тем, чтобы среди множества грамматик, относительно которых G' является упрощённой, оказалась грамматика G, порождающая, как минимум, все правильные программы на целевом языке.

Проводится эксперимент по итеративной разработке легковесной LL(1) грамматики с символом Any для языка C#. Для проверки сгенерированных парсеров используются репозитории трёх крупных проектов с открытым исходным кодом (всего 10845 файлов). Отмечается, что итоговый парсер успешно разбирает все передаваемые ему на вход программы.

В **третьей главе** вводятся расширения языка LanD: параметризованные разновидности Any, позволяющие ещё больше упростить правила грамматики, и определения парных конструкций, позволяющие учитывать глубину вложенности при обработке Any и игнорировать все токены, расположенные на большем уровне вложенности, чем токен, с которого началась обработка.

Описываются механизмы специфического восстановления от ошибок, основанного на обработке Any. Восстановление позволяет расширить множество неправильных программ, разбираемых парсером, а также успешно проводить разбор программы из $L(G)$ в случае, когда разработанная легковесная грамматика G_s с символом Any оказывается упрощённой относительно некоторой грамматики G', порождающей не все корректные программы из $L(G)$. Отмечается, что в одном из возможных сценариев восстановления требуется возврат во входном потоке, из-за чего в худшем случае оценка сложности алгоритмов парсинга составляет $O(n^2)$. Однако, в соответствии с экспериментально полученными данными, данный сценарий реализовывался всего в 0,06% случаев от общего количества восстановлений, поэтому в среднем сохраняется оценка $O(n)$.

```
namespace_content =    opening_directive*! (namespace|entity|general_attribute)*
opening_directive =    ('using'|'extern') Any ';'
namespace         =    'namespace' name '{' namespace_content '}'
entity            =    enum | class_struct_interface | method | field | property | water_entity
enum              =    common_beginning 'enum' name Any '{' Any '}' ';'?
class_struct_interface   =    common_beginning CLASS_STRUCT_INTERFACE name Any '{' entity* '}' ';'?
method            =    common_beginning type name arguments Any method_body
method_body       =    init_expression? ';' | block
field             =    common_beginning type name ('[' Any ']')? init_value? (',' name ('[' Any ']')? init_value?)* ';'
property          =    common_beginning type name property_body
property_body     =    (block (init_value ';')? | init_expression ';')
water_entity      =    AnyInclude('delegate', 'operator', 'this') (block | ';')+
common_beginning  =    entity_attribute* modifier*
modifier          =    MODIFIER | 'extern'
init_expression   =    '=>' Any
init_value        =    '=' init_part+
init_part         =    Any | type
name_atom         =    ID type_parameters?
name              =    name_atom (('.'|'::') name_atom)*
names_list        =    name (',' name)*
tuple             =    '(' type name? (',' type name?)* ')'
type_atom         =    ('unsigned'? ID | tuple) type_parameters? '?'? '*'*
type              =    type_atom (((.'|'::') type_atom) | ('[' Any ']'))*!
type_parameters   =    '<' (AnyAvoid(';') | type_parameters)* '>'
entity_attribute  =    '[' Any ']'
general_attribute =    GENERAL_ATTRIBUTE_START Any ']'
arguments         =    '(' Any ')'
block             =    '{' Any '}'
```

Рис. 1. Фрагмент легковесной LR(1) грамматики на языке LanD для языка C#.

В проводимом эксперименте легковесные LL(1) и LR(1) парсеры языков C# и Java, сгенерированные по разработанным нами легковесным грамматикам с символом Any, применяются к девяти крупным проектам с открытым исходным кодом (59138 файлов для C# и 27393 — для Java). Количество крупных синтаксических сущностей, выявленных легковесными парсерами, сравнивается с количеством сущностей, обнаруженных полными парсерами соответствующих языков. Делается вывод о том, что в реальных промышленных программах все искомые сущности успешно выявляются при легковесном анализе.

На рисунке 1 показаны правила легковесной LR(1) грамматики языка C#, используемой в эксперименте. Общий размер грамматики составляет 52 строки, что почти в 50 раз короче полной спецификации языка. Парсер, сгенерированный по данной грамматике, в дальнейшем применяется для построения АСД в задаче устойчивой привязки к коду.

В **четвёртой главе** решается задача устойчивой привязки к синтаксическим сущностям программы. Привязка осуществляется на основе абстрактного синтаксического дерева, построенного легковесным парсером. Каждая сущность a, к которой производится привязка, (*точка привязки*) описывается кортежем вида

$$\mathbf{BindingPoint_a} = (Type_a, H_a, I_a, S_a, N_a, C_a) \ ,$$

где $Type_a$ — нетерминальный символ грамматики, соответствующий a, H_a — контекст *заголовка*, I_a — *внутренний* контекст, S_a — контекст *областей*, N_a — контекст *соседей*, C_a — контекст *наиболее похожих*. Далее под a будем также понимать соответствующий синтаксическому элементу узел АСД.

H_a представляется в виде списка элементов-четвёрок вида

$$(Type, Priority, ComparisonMode, Words) \ ,$$

по одной для каждого листового непосредственного потомка узла a. В каждой четвёрке компонента $Type$ — терминальный или нетерминальный символ,

соответствующий элементу заголовка; $Priority \in \mathbb{R}^{\geq 0}$ отражает важность совпадения элемента при сравнении заголовков; $ComparisonMode \in \{Distance, ExactMatch\}$ определяет, как проводить сравнение двух таких элементов — путём вычисления редакционного расстояния или проверки строгого совпадения.

I_a хранится как тройка $(Text, Hash, Length)$. Далее тройки данного вида будем называть **TextOrHash**. Компонента *Text* хранит конкатенированный текст всех нелистовых непосредственных потомков узла a. Если этот текст оказывается слишком длинным, вместо него в компоненте *Hash* сохраняется его нечёткий хеш, а компонента *Text* остаётся пустой. Длина текста всегда сохраняется в *Length*.

S_a — список пар вида $(Type, H)$, построенных для всех предков узла a. Каждая пара хранит тип и контекст заголовка некоторого предка.

N_a хранится как пара $(Before, After)$, где каждая компонента также является парой вида $(All, Nearest)$. *Before* описывает синтаксические элементы, предшествующие a; *After* описывает элементы, следующие за a. *All* — это тройка **TextOrHash**, построенная для конкатенированного текста всех предшествующих (в *Before*) или последующих (в *After*) элементов, имеющих общего с узлом a непосредственного предка. *Nearest* — это кортеж **BindingPoint**, построенный для ближайшего к a синтаксического элемента типа $Type_a$ в пределах того же файла. Этот элемент может не иметь общего с a непосредственного предка, от него требуется только близость расположения в тексте программы. Для самого *Nearest* контекст соседей не строится, чтобы избежать привязки к абсолютно всему содержимому файла.

C_a — список точек привязки **BindingPoint**, построенных для элементов типа $Type_a$, наиболее похожих на a в момент привязки к нему. Например, когда a является одной из реализаций перегруженного метода, в C_a сохраняются описания других реализаций этого метода. Наиболее похожие элементы определяются с использованием части алгоритма 1. C_a не строится для элементов из самого контекста наиболее похожих.

В случае, если требуется найти точку привязки a в изменившемся файле (произвести перепривязку), строится АСД кода, присутствующего в актуальной версии этого файла, и формируется список кандидатов — массив кортежей **BindingPoint**, соответствующих представленным в коде сущностям типа $Type_a$. Функция Rebind алгоритма 1 используется для установления соответствия между всеми точками привязки одного типа, присутствовавшими в старой версии файла, и кандидатами, существующими в актуальной версии.

Для каждой точки и кандидата вычисляется расстояние — число в диапазоне $[0; 1]$, получаемое как скалярное произведение вектора поконтекстных расстояний и эвристически формируемого вектора весов. Веса — это числа, отражающие важность каждого из контекстов при перепривязке конкретной точки. В ходе итеративного процесса каждая точка перепривязывается к наиболее близкому кандидату, если для него выполняются условия, гарантирующие отсутствие неоднозначностей (строки 43–45 алгоритма 1).

Алгоритм 1 Алгоритм перепривязки.

```
1:  function CheckGap(v_1, v_2):
2:      return v_2 ≠ 0 ∧ v_1 · 2 ≤ v_2;
3:  function SimpleRebind(a, candidates = {c_1,c_2,...c_m}):
4:      predicates := [ AreEqual^{Core(H)}, AreEqual^{NotCore(H)}, AreEqual^{I} ];
5:      startIdx := Length(Core(H_a)) > 0 ? 1 : Length(H_a) > 0 ? 2 : Length(I_a) > 0 ? 3 : 0; idx := 0;
6:      if startIdx = 0 then return null;
7:      for (i from startIdx to Length(predicates)) do
8:          if ¬ (∃ k ∈ [1..Length(C_a)] : ∀ l ∈ [1..i], predicates[l](a, C_a[k]) ∧ AreEqual^{S}(a, C_a[k])) then
9:              idx := i;
10:             break;
11:     if idx = 0 then return null;
12:     for (i from idx to Length(predicates)) do
13:         if ∃! k ∈ [1..m] : ∀ l ∈ [1..i], predicates[l](a, c_k) ∧ AreEqual^{S}(a, c_k) then
14:             return c_k;
15:     return null;
16: function GetTotalDistances(a, candidates, distances, weights):
17:     for all c ∈ candidates do
18:         result[c] := distances[a][c] · weights[a] / ∑_{w ∈ weights[a]} w ;
19:     return result;
20: function Rebind(points = {a_1,a_2,...a_n}, candidates = {c_1,c_2,...c_m}):
21:     allPoints := ∪_{i∈[1..n]}(C_{a_i} ∪ {Nearest(Before(S_{a_i})), Nearest(After(S_{a_i}))}) ∪ points;
22:     unmatched := allPoints;
23:     for all p ∈ unmatched do
24:         match := SimpleRebind(p, candidates);
25:         if match ≠ null then
26:             unmatched := unmatched \ {p};
27:             candidates := candidates \ {match};
28:             if p ∈ points then autoResults[p] := {match};
29:     for all p ∈ unmatched do
30:         for all c ∈ candidates do
31:             distances[p][c] := GetDistances(p, c);
32:     oldResultLength := 0;
33:     do
34:         oldResultLength := Length(autoResults);
35:         for all p ∈ unmatched do
36:             weights[p] := GetWeights(p, candidates, distances);
37:             totalDistances[p] := GetTotalDistances(p, candidates, distances, weights);
38:             ordered[p] := OrderAsc(candidates, totalDistances[p]);
39:         for all p ∈ unmatched do
40:             c := ordered[p][1];
41:             bestDist := totalDistances[p][c];
42:             otherBestDist := Min_{p' ∈ unmatched \ {p}}(totalDistances[p][c])
43:             if bestDist ≤ MaxDist ∧ (Length(candidates) = 1
44:                 ∨ CheckGap(bestDist, totalDistances[p][ordered[p][2]]))
45:                 ∧ (otherBestDist = null ∨ CheckGap(bestDist, otherBestDist)) then
46:                 unmatched := unmatched \ {p};
47:                 candidates := candidates \ {c};
48:                 if p ∈ points then
49:                     autoResults[p] := ordered[p];
50:         for all p ∈ unmatched do
51:             UpdateNearestNeighboursDistances(p, candidates, distances[p]);
52:     while oldResultLength ≠ Length(autoResults);
53:     for all p ∈ unmatched do
54:         interactiveResults[p] := OrderAsc(candidates, totalDistances[p]);
55:     return (autoResults, interactiveResults);
```

Устойчивость привязки проверяется экспериментально. Для трёх проектов с открытым исходным кодом на языке C# некоторые версии их кодовых баз выбираются в качестве исходных. В них проводится привязка ко всем классам, методам, полям и свойствам. Затем в версиях этих проектов, отстоящих от исходных на временной промежуток от двух месяцев до двух лет, осуществляется перепривязка. Аналогичный эксперимент проводится для грамматики в формате генератора парсеров Yacc. Показано, что корректно проводимые перепривязки составляют 97,76% от общего числа отредактированных синтаксических элементов для Yacc и 99,8% для C#.

В худшем случае на каждой итерации цикла **do – while** функции Rebind алгоритма 1 происходит перепривязка одной точки и вычёркивание одного кандидата. Тогда общее количество итераций составляет $\min(n, m)$, а сложность функции Rebind составляет $O(\min(n, m) \cdot n \cdot m \cdot \log(m))$, где n — количество искомых точек привязки, m — количество кандидатов. Однако в проведённых экспериментах менее 50% точек перепривязываются на первой итерации всего для 1% включённых в эксперимент файлов. Если считать, что на каждой итерации цикла количество неперепривязанных точек сокращается в два и более раза, сложность перепривязки составляет $O(n \cdot m \cdot \log(m))$. В тексте диссертации представлены графики зависимости времени работы алгоритма от количества точек привязки, полученные в ходе проведения экспериментов.

На рисунке 2 представлен фрагмент окна среды разработки Visual Studio с интегрированной в неё панелью разметки кода, реализованной на базе описанных моделей и алгоритмов. В видимой части разметки, открытой в данной панели, присутствуют девять пометок, сгруппированных в соответствии с функциональностями, к которым они относятся: пять меток отнесены к функциональности «Инлайн-опции», четыре метки отнесены к функциональности «Новые модели», из них три — к подфункциональности «Контекст похожих».

В **пятой главе** модели и алгоритмы из предыдущей главы дополняются для обеспечения устойчивой привязки к произвольному однострочному или многострочному фрагменту кода.

Привязка к однострочному фрагменту осуществляется посредством привязки к объемлющей синтаксической сущности и построения дополнительного контекста *строки*, которым расширяется ранее введённая модель **BindingPoint**. В эксперименте исследуется устойчивость привязки к однострочным фрагментам, делается вывод о том, что для непустых и непробельных строк успешность перепривязки составляет от 98,1 до 99,5%.

Привязка к многострочному фрагменту сводится к привязке к синтаксической сущности за счёт встраивания узла, соответствующего этому фрагменту, в АСД программы. В грамматику на языке LanD добавляется конфигурационная секция, задающая токены, которые должны быть вставлены в код для обозначения границ фрагмента, который должен быть представлен в АСД. Этими границами, например, могут стать комментарии, начинающиеся с определённой последовательности символов. Ограниченный таким образом фрагмент называ-

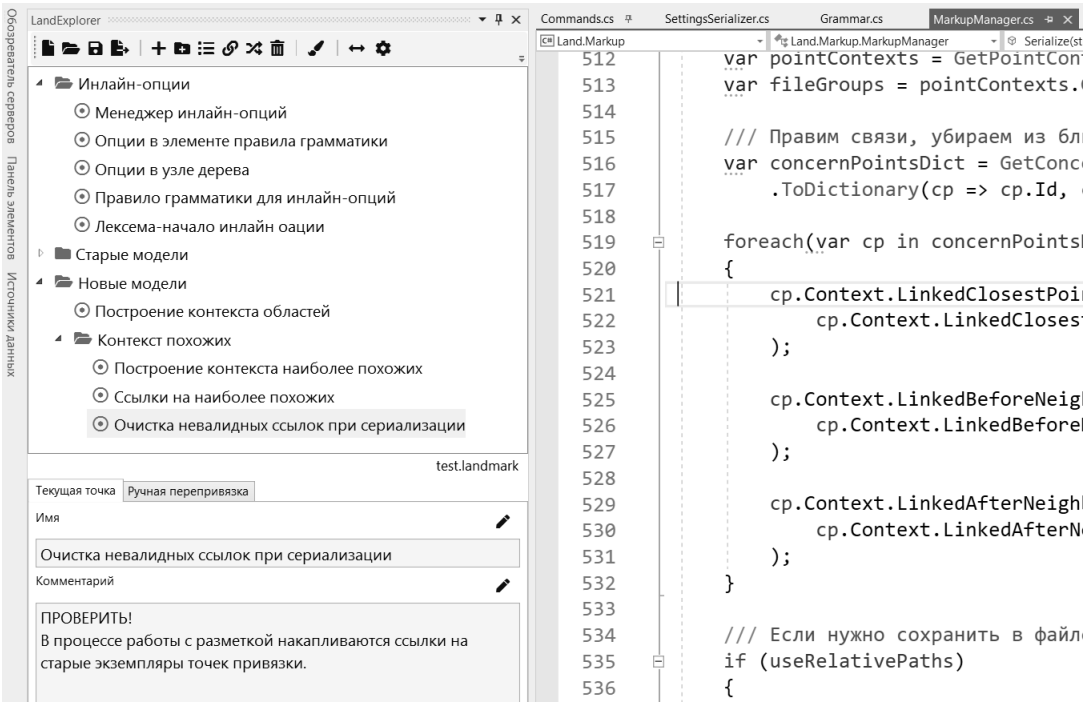

Рис. 2. Фрагмент окна IDE Visual Studio 2019 с интегрированной панелью разметки кода.

ется *обрамлённым*. Формулируются следующие *условия встраивания* узла f, соответствующего обрамлённому фрагменту, в АСД программы:

1. $r \subseteq f$;
2. существует узел n, принадлежащий АСД, такой, что для любого узла n', принадлежащего АСД и являющегося предком n либо самим n, $f \mathbin{!\sqcap} n'$ или $f \subset n'$, а любой узел n'' такой, что $n'' \subseteq f$, является потомком n.

Здесь r — корень АСД, а символами $\subseteq$, $!\sqcap$, $\subset$ обозначаются отношения нестрогого вложения, строгого пересечения, строгого вложения в смысле соответствующих узлам областей текста программы. Выполнение одного из условий для обрамлённого фрагмента означает корректность выделения этого фрагмента относительно синтаксической структуры программы.

Далее в диссертации приводится алгоритм, осуществляющий встраивание в АСД узлов, соответствующих корректно выделенным фрагментам. Для успешно встроенного узла доказываются утверждения о том, что он не нарушает корректность отношений между узлами дерева, существовавшими до встраивания, и сам не участвует в некорректных отношениях.

Утверждение 3. Пусть узел f, соответствующий пользовательскому блоку, встроен в АСД. Тогда для любых узлов n_1 и n_2, принадлежащих этому АСД и не совпадающих друг с другом и с f,

1. n_1 является предком n_2, если и только если n_1 являлся предком n_2 до встраивания f;
2. узлы n_1 и n_2 связаны отношением $\subseteq$, если и только если они были связаны этим отношением до встраивания f.
3. узлы n_1 и n_2 связаны отношением $\sqcap$, если и только если они были связаны этим отношением до встраивания f.

Утверждение 4. Пусть узел f, соответствующий пользовательскому блоку, встроен в АСД. Тогда для любого узла n', принадлежащего этому АСД и не совпадающего с f,

1. n' — предок $f \implies f \subseteq n'$;
2. f — предок $n' \implies n' \subseteq f$;
3. $n' \sqcap f \iff n'$ — предок f или f — предок n'.

В **заключении** кратко излагаются основные результаты настоящего исследования.

1. Предложены и формально описаны концепции легковесных LL(1) и LR(1) грамматик с символом Any, упрощённой LL(1) грамматики, упрощённой LR(1) грамматики, развивающие метод легковесного синтаксического анализа на основе островных грамматик.
2. Описаны и алгоритмически реализованы методы LL(1) и LR(1) синтаксического анализа, позволяющие осуществлять разбор программы, порождённой «полной» грамматикой G, на основе грамматики G_s, упрощённой относительно G; для случая LL(1) доказано, что парсер, сгенерированный по упрощённой грамматике G_s, разбирает все правильные программы, порождённые G; для случая LR(1) сформулированы условия, существенные для правильного разбора парсером, сгенерированным по упрощённой грамматике, программ, порождаемых «полной» грамматикой.
3. Реализован генератор легковесных парсеров LanD со встроенным языком описания упрощённых грамматик, использующий указанные алгоритмы; экспериментально подтверждена применимость генерируемых легковесных парсеров для анализа крупных программных проектов и выявления сущностей, интересных с точки зрения задачи привязки к коду.
4. Предложены и реализованы модели контекстов, описывающих участок программы, к которому необходимо осуществить привязку, и метод перепривязки, необходимый для поиска участка в изменённом коде; экспериментально подтверждена устойчивость выполняемой с их помощью привязки: для языка C# перепривязка осуществляется корректно в 99,8% случаев.
5. Предложен и алгоритмически реализован метод учёта произвольных участков кода программы в абстрактном синтаксическом дереве этой программы; сформулирован критерий корректности выделения таких участков, доказано, что встраивание узлов, соответствующих этим участкам, в абстрактное синтаксическое дерево не нарушает корректность этого дерева.
6. Реализована панель разметки кода, предназначенная для интеграции в различные интерактивные среды разработки, использующая разработанные модели и алгоритмы привязки и перепривязки.

Также описываются возможные направления дальнейших исследований:

- построение модели отношений между элементами размеченных прорезающих функциональностей и использование этой модели для проверки корректности вносимых программистом изменений, затрагивающих прорезающие функциональности;
- разработка алгоритмов перепривязки, учитывающих конкретные сценарии редактирования кода (например, при рефакторинге);
- разработка метода слияния разметок, относящихся к разным версиям кода, и создание специализированной утилиты.

Список публикаций по теме диссертации

Статьи в журналах из перечня российских рецензируемых научных журналов, в которых должны быть опубликованы основные научные результаты диссертаций на соискание учёных степеней доктора и кандидата наук

1. Головешкин А. В. Поиск и анализ сквозных функциональностей в размеченной грамматике языка программирования // Известия вузов. Северо-Кавказский регион. Технические науки. — 2017. — № 3. — С. 29–34.
2. Goloveshkin A. V., Mikhalkovich S. S. Tolerant parsing with a special kind of “Any” symbol: the algorithm and practical application // Trudy ISP RAN [Proc. ISP RAS]. — 2018. — Vol. 30, no 4. — P. 7–28.
3. Goloveshkin A. V. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol // Trudy ISP RAN [Proc. ISP RAS]. — 2019. — Vol. 31, no 3. — P. 7–28.
4. Головешкин А. В., Михалкович С. С. Устойчивая алгоритмическая привязка к произвольному участку кода программы // Программные системы: теория и приложения. — 2022. — Т. 13, №1. — С. 3–33.

Статьи в изданиях, индексируемых в реферативных базах Scopus и Web of Science

1. Goloveshkin A. V., Mikhalkovich S. S. Using improved context-based code description for robust algorithmic binding to changing code // Procedia Computer Science. — 2021. — Vol. 193. — P. 239–249.

Прочие публикации

1. Головешкин А. В. Сквозная функциональность и её анализ в грамматике языка программирования // Языки программирования и компиляторы — 2017. Труды конференции. — 2017. — С. 82–85.

2. Головешкин А. В., Михалкович С. С. LanD: инструментальный комплекс поддержки послойной разработки программ // Труды XXV всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». — 2018. — С. 53–56.
3. Головешкин А. В., Михалкович С. С. Привязка к произвольному участку программы в задаче разметки программного кода // Труды XXVI всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». — 2019. — С. 86–89.
4. Головешкин А. В., Михалкович С. С. Разметка сквозных функциональностей в коде программы // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции. — 2019. — С. 245–256.
5. Головешкин А. В. Устойчивая разметка прорезающих функциональностей в грамматике языка программирования // Материалы XXVIII всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». — 2021. — С. 143–146.

Свидетельства о государственной регистрации программ для ЭВМ

1. Свидетельство о государственной регистрации программы для ЭВМ №2022610266 Российская Федерация. Генератор легковесных LL(1) и LR(1) синтаксических анализаторов / А. В. Головешкин; правообладатель А. В. Головешкин (RU). — №2021681187; заявл. 20.12.2021; опубл. 11.01.2022, Бюл. № 1. — 1 с.
2. Свидетельство о государственной регистрации программы для ЭВМ №2022616186 Российская Федерация. Свидетельство об официальной регистрации программы для ЭВМ. Панель разметки кода / А. В. Головешкин, С. С. Михалкович; правообладатели А. В. Головешкин (RU), С. С. Михалкович (RU). — №2022613947; заявл. 17.03.2022; опубл. 05.04.2022, Бюл. № 4. — 1 с.
3. Свидетельство о государственной регистрации программы для ЭВМ №2022616984 Российская Федерация. Библиотека устойчивой алгоритмической привязки к коду программы / А. В. Головешкин, С. С. Михалкович; правообладатели А. В. Головешкин (RU), С. С. Михалкович (RU). — №2022613933; заявл. 15.03.2022; опубл. 18.04.2022, Бюл. № 4. — 1 с.

Головешкин Алексей Валерьевич
Устойчивая алгоритмическая привязка к коду программы
Автореф. дис. на соискание учёной степени канд. тех. наук

Подписано в печать ___.___.2022 г. Заказ №_____
Формат 60 × 90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография _____.