ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ т. 13, № 1(52), с. 3–33
научная статья — МАТЕМАТИЧЕСКИЕ ОСНОВЫ ПРОГРАММИРОВАНИЯ

УДК 004.4'23+004.415+004.02
10.25209/2079-3316-2022-13-1-3-33

Устойчивая алгоритмическая привязка к произвольному участку кода программы

Алексей Валерьевич **Головешкин**✉, Станислав Станиславович **Михалкович**
Южный Федеральный Университет, Ростов-на-Дону, Россия,
alexeyvale@gmail.com✉ (*подробнее об авторах на с. 33*)

Аннотация. При работе над задачей программист наиболее активно взаимодействует с конечным набором фрагментов кода. Информация об их расположении важна для быстрого перемещения между ними, для других разработчиков и как разновидность документации. Интегрированные среды разработки (IDE) позволяют связывать метки с участками кода, просматривать список меток и использовать их для быстрой навигации, однако связь между меткой и помеченным местом может теряться при редактировании кода, особенно при изменении за пределами IDE.

В предыдущих работах авторами предлагается интегрируемый в IDE инструмент, позволяющий устойчиво к изменению кода помечать крупные синтаксические сущности программы («привязываться» к ним). Описание помечаемого элемента строится по абстрактному синтаксическому дереву (АСД) программы и используется для алгоритмического поиска этого элемента в отредактированном позднее коде. Поиск осуществляется с успешностью от 99 до 100%.

Целью настоящей работы является устойчивая алгоритмическая привязка к *произвольному* участку кода. Для привязки к однострочному фрагменту кода предложены расширение модели, описывающей помечаемый фрагмент, и дополнительный алгоритм поиска. Введена необходимая формализация и предложен алгоритм встраивания в АСД узлов, соответствующих многострочным фрагментам; показано, что в результате такого встраивания не нарушается корректность АСД. В коде трёх крупных проектов на языке C# произведены привязки к случайно выбранным строкам. Ручной проверкой результатов поиска этих строк в отредактированном коде подтверждено, что привязка устойчива к редактированию кода.

Ключевые слова и фразы: разметка кода, алгоритмическая привязка к коду, разработка программного обеспечения, абстрактное синтаксическое дерево, оценка похожести кода

Для цитирования: Головешкин А. В., Михалкович С. С. *Устойчивая алгоритмическая привязка к произвольному участку кода программы* // Программные системы: теория и приложения. 2022. **Т. 13**, № 1(52). С. 3–33. http://psta.psiras.ru/read/psta2022_1_3-33.pdf

This Article in English: *http://psta.psiras.ru/read/psta2022_1_35-62.pdf*

Введение

Навигация по коду является одной из основных активностей программиста в ходе работы над задачей и интенсивно осуществляется на всех этапах работы: от первичного исследования кода до финальной проверки внесённых изменений [1, 2]. Наиболее активно переходы осуществляются в пределах конечного набора фрагментов кода, представляющих интерес в рамках решаемой задачи. Сохранение информации о расположении этих фрагментов для быстрого перемещения между ними, а также обмен этой информацией с другими разработчиками — проблема, которую программисты вынуждены решать при помощи вспомогательных внешних средств и инструментов, непосредственно для этого не предназначенных [3, 4]. Механизмы, предлагаемые интегрированными средами разработки (IDE) для пометки участков кода, неудобны в использовании и не обеспечивают достаточных возможностей для связывания дополнительной высокоуровневой семантической информации, касающейся решаемой задачи, с помеченными участками. Кроме того, разметка быстро перестаёт соответствовать актуальному состоянию кода: в результате редактирований текста программы теряется связь между меткой и помеченным участком. Как следствие, программисты крайне редко прибегают к помощи встроенных средств IDE [5, 6].

В наших работах [7–9] предложены модели, алгоритмы и инструменты, позволяющие пометить интересующие программиста участки кода программы (*привязаться* к ним), сопроводить метки развёрнутым комментарием и организовать их в виде иерархической структуры произвольной глубины, сохранить и загрузить такую разметку. Метки связываются с кодом *алгоритмически*: в момент, когда пользователь инициирует пометку некоторого фрагмента, производится синтаксический разбор программы и на основе полученного абстрактного синтаксического дерева (АСД) конструируется модель, описывающая данный фрагмент (см. раздел 2). В момент, когда пользователь запрашивает переход к ранее помеченному участку, данная модель используется для алгоритмического поиска помеченного фрагмента в актуальной версии программы и позволяет найти его (*переприв язаться*) даже в том случае, если фрагмент или его окружение были отредактированы. Таким образом, разметка является *устойчивой* к редактированию кода.

На рисунке 1 показан фрагмент окна IDE Visual Studio с интегрированной в неё разработанной нами панелью разметки. Видимая часть

разметки, открытой в панели, состоит из 8 меток, сгруппированных в соответствии с некоторой семантикой предметной области.

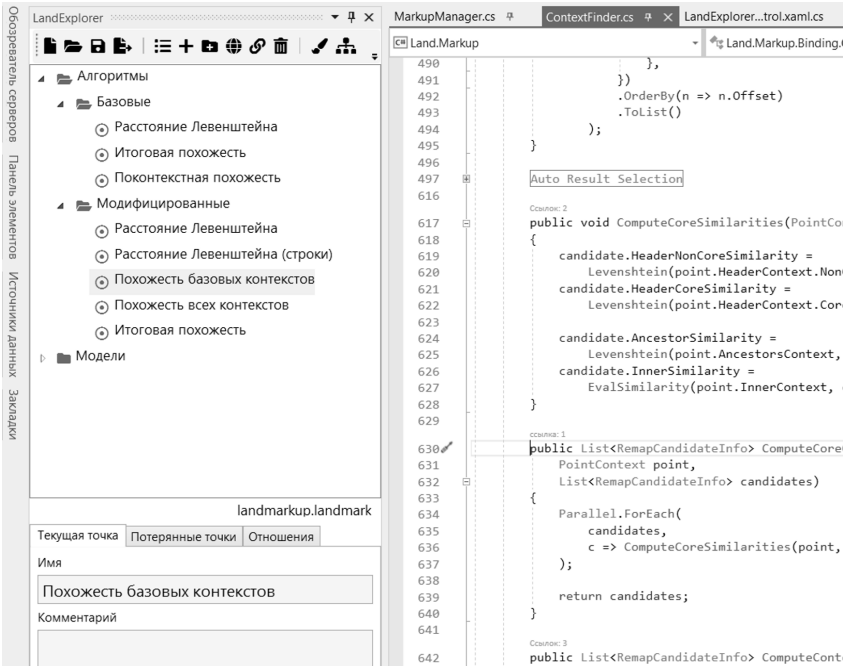


Рисунок 1. Фрагмент окна Visual Studio со встроенной панелью разметки

Предложенные в [9] модели и алгоритмы рассчитаны на привязку к крупным и средним синтаксическим сущностям программы, например, к классам и членам класса для программы на языке C#. Зачастую требуется пометить не метод или класс целиком, а конкретную строчку кода или несколько смежных строчек. Данная задача требует разработки дополнительных алгоритмов, а также модификации ранее созданных нами моделей. Проблема привязки к многострочным участкам кода впервые поднимается в [10]: отмечается, что привязку к нескольким смежным строкам можно выполнить так же, как и привязку к крупной синтаксической сущности, однако для этого необходимо, чтобы узел, соответствующий целевому фрагменту кода, присутствовал в АСД программы. В настоящей работе мы развиваем эту идею.

Основным вкладом настоящей работы являются:

(*1*) расширение модели, используемой для привязки к синтаксическим сущностям программы, и модификация алгоритма переприв язки, позволяющие осуществлять устойчивую привязку к одиночной строке кода;

(*2*) формализация концепции пользовательского блока — вспомогательной сущности, служащей для привязки к многострочным фрагментам кода; алгоритм, осуществляющий встраивание в АСД узлов, соответствующих пользовательским блокам, с сохранением корректности АСД;

(*3*) результаты экспериментов, демонстрирующие успешность применения описанных моделей и алгоритмов для устойчивой привязки к коду.

Далее работа структурирована следующим образом. В разделе 1 приводится обзор исследований, затрагивающих проблематику привязки и переприв язки. В разделе 2 кратко излагаются ранее полученные авторами результаты, на базе которых строится данное исследование. В разделе 3 описываются модели и алгоритмы, разработанные для адаптации инструмента разметки к привязке к произвольному участку кода программы, как однострочному, так и многострочному. В разделе 4 экспериментально проверяется устойчивость разметки кода, выполненной с использованием описанных моделей и алгоритмов.

1. Обзор предметной области

1.1. Привязка к коду

Проблема «запоминания» признаков, по которым можно было бы найти некоторое место в коде, давно и широко известна. В работе А. Л. Фуксмана [**11**] для получения интегрированной программы путём добавления расширяющих функций к некоторой основе требуется однозначно идентифицировать те точки основы, куда необходимо встроить фрагменты расширяющей функции. Эту идентификацию предполагается производить по текстовым координатам — номеру строки и столбца. В более поздних парадигмах, также основанных на идее разделения базовой версии программы и добавляемых к ней расширений [**12**–**14**], места встраивания расширений предлагается идентифицировать по имени класса, имени метода, сигнатуре, возможно, с использованием шаблонов. Все упомянутые способы в значительной степени неустойчивы к редактированию кода и не могут быть применены для привязки в задаче разметки.

Ещё одним подходом к поиску нужного места в кодовой базе является обозначение этого места непосредственно в коде при помощи *псевдокомментариев* — комментариев специального вида, обрабатываемых компилятором языка так же, как обрабатываются обычные комментарии, но имеющих специальное значение для некоторого внешнего инструмента. Псевдокомментарии интенсивно используются в различных системах разработки и сопровождения программного обеспечения [15, 16], данный подход устойчив к любому редактированию кода, однако может перестать работать при случайном изменении самих псевдокомментариев. Важно и то, что псевдокомментарии «замусоривают» код информацией, имеющей лишь технический смысл.

Развиваемый в настоящей работе подход основан на идее представления запоминаемого участка в виде набора структур специального вида — *контекстов* — и хранения данного набора-описания отдельно от кода [18, 19]. В исследовании [9] нами предложены модели контекстов, захватывающие достаточно информации для последующего успешного поиска крупных и средних синтаксических сущностей в изменившейся программе.

1.2. Поиск кода

Существуют проблемы, смежные с проблемой устойчивой привязки к коду: поиск клонов кода [20–22], поиск плагиата [23, 24], рекомендация кода [25]. Как и устойчивая привязка, они относятся к области анализа эволюции программного обеспечения. Общей подпроблемой всех перечисленных задач является оценка похожести программ, однако каждая из задач имеет свою специфику и дополнительные предположения, накладываемые на сравниваемые кодовые базы. Проблема устойчивой привязки к коду имеет следующие отличительные особенности:

- гарантируется, что сравниваются две версии одной и той же кодовой базы;
- для каждого фрагмента, помеченного в исходной версии кода, существует единственное верное соответствие в актуальной версии кода, либо нет ни одного соответствия.

Отдельная категория работ посвящена восстановлению конкретного сценария редактирования кода по имеющимся исходной и актуальной версиям [26–29], а также предсказанию дальнейших редактирований [30]. Для получения хороших результатов в этой области требуются

подробный анализ программы полным синтаксическим анализатором соответствующего языка и учёт нюансов языка при дальнейшем сравнении. В своём подходе мы полагаемся на легковесный анализ структуры программы (разбор до уровня тех сущностей, к которым планируется осуществлять привязку, и работу с обычным текстом на более мелком уровне) и обработку легковесных синтаксических деревьев. Все модели и алгоритмы, используемые непосредственно для привязки и перепривязки являются языконезависимыми и могут быть с минимальными затратами применены к любому структурированному тексту.

Отметим также, что алгоритм, предлагаемый нами для поиска помеченных однострочных фрагментов, идейно перекликается с алгоритмом LHDiff [**31**], разработанным для отслеживания строки кода на основе её содержимого и окружения. Однако, в отличие от [**31**], мы не считаем, что версия кода, существовавшая в момент «запоминания» строки, всегда доступна. Такую доступность не может гарантировать даже применение системы контроля версий, поскольку привязка может осуществляться в моменты, когда код находится в некотором промежуточном состоянии между коммитами. Мы сохраняем ограниченный объём информации, описывающей помечаемую строку и её окружение, и опираемся только на эту сохранённую информацию при последующем поиске, предварительно сужая область поиска до объемлющей синтаксической сущности.

2. Предварительные сведения

2.1. Легковесный синтаксический анализ

Одно из главных требований, предъявляемых нами к решению задачи устойчивой разметки кода, — языконезависимость моделей и алгоритмов. Инструмент разметки может быть встроен в различные IDE, предназначенные для различных языков. Кроме того, в одном проекте зачастую используется несколько языков программирования, и программисту может потребоваться учесть в одной разметке фрагменты кода на разных языках. Требование языконезависимости естественным образом распространяется и на формат, в котором должны быть представлены анализируемые инструментом синтаксические деревья.

Второе требование, касающееся АСД, — легковесность. Контексты, описывающие помечаемый фрагмент, разработаны в предположении,

что привязка осуществляется к «каркасу» программы — крупногабаритным и среднегабаритным сущностям: определения таких сущностей содержат достаточно большое количество информации, полезной для последуюшего поиска. Значительную часть программы, например, все тела методов, можно оставить неразобранной и не структурировать в дереве, работая с некоторого уровня детализации с обычным текстом. Такие деревья легче конструировать и быстрее обрабатывать.

Для добавления поддержки очередного языка можно использовать его существующий промышленный синтаксический анализатор. Основная сложность такого подхода состоит в том, чтобы выяснить, как в АСД, выстраиваемом данным анализатором, структурированы пригодные для привязки элементы: эта информация необходима для правильного построения моделей, описывающих помечаемый элемент. Для сбора такой информации требуется исследовать полную спецификацию языка (если она доступна). В [32, 33] отмечается времязатратность подобного исследования. Кроме того, необходимо реализовать конвертер АСД, обрезающий дерево и преобразующий его к единому формату.

Более перспективным подходом является самостоятельная реализация *островной грамматики* [32] — легковесной грамматики, в которой подробно описываются только те синтаксические сущности, к которым планируется осуществлять привязку — и последующая генерация синтаксического анализатора по ней. В теории островных грамматик подробно описанные сущности, важные в рамках той задачи, ради решения которой разрабатывается грамматика, называются *островами*. В рамках исследований [7, 8], посвящённых развитию концепции синтаксического анализа на основе островных грамматик, алгоритмы LL(1) и LR(1) синтаксического разбора модифицированы нами для работы с островными грамматиками, также разработаны генератор легковесных синтаксических анализаторов LanD и специальный язык для описания островных грамматик. При помощи LanD мы самостоятельно реализовали синтаксические анализаторы большого количества языков для инструмента разметки. На рисунке 2 продемонстрирован фрагмент разработанной нами легковесной LR(1) грамматики языка C#. Здесь специальный токен `Any` используется для обозначения областей, структура которых неважна и не должна быть разобрана и представлена в дереве.

```
enum = common 'enum' name Any '{' Any '}' ';'?
class_struct_interface = common CLASS_STRUCT_INTERFACE name Any '{' entity* '}' ';'?
method = common type name arguments Any (init_expression? ';' | block)
field = common type name ('[' Any ']')? init_value? (',' name ('[' Any ']')? init_value?)* ';'
property = common type name (block (init_value ';')? | init_expression ';')
water_entity = AnyInclude('delegate', 'operator', 'this') (block | ';')+

common = entity_attribute* modifier*
entity_attribute = '[' Any ']'
arguments = '(' Any ')'
block = '{' Any '}'
```

Рисунок 2. Фрагмент легковесной LR(1) грамматики языка C#

2.2. Привязка к крупным синтаксическим элементам

В работе [9] нами предложены модели и алгоритмы для устойчивой привязки к синтаксическим сущностям программы. Каждая помечаемая сущность a описывается кортежем следующего вида:

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

Здесь $Type_a$ — нетерминальный символ грамматики, соответствующий a. Поскольку привязка основывается на анализе АСД, привязка к сущности a фактически означает привязку к соответствующему ей узлу дерева. Далее по тексту нижний индекс компоненты кортежа, обозначающий конкретную помечаемую сущность, может быть опущен в общих рассуждениях, касающихся этой компоненты.

H_a — контекст *заголовка*, хранимый как список четвёрок вида $(Type, Priority, ComparisonMode, Words)$. Каждая четвёрка соответствуют непосредственному листовому потомку a. На рисунке 3 представлены заголовок метода и контекст заголовка, построенный для этого метода.

Type — символ легковесной грамматики, соответствующий элементу заголовка, *Priority* — неотрицательное вещественное число, отражающее важность совпадения элемента при сравнении двух заголовков; $ComparisonMode \in \{$“*Distance*”,“*ExactMatch*”$\}$ определяет, как оценивать похожесть соответствующих элементов: так, при сравнении имён методов (`name`) нужно вычислить редакционное расстояние, а при сравнении модификаторов (`MODIFIER`) имеет смысл только проверка строгого совпадения. Список *Words* конструируется путём разделения текста, соответствующего элементу заголовка, на числобуквенные

```
public override void Visit(BinOpNode binop) { … }
```

H_a = (

(MODIFIER,	0.1,	“ExactMatch”,	((1, “public”)),
(MODIFIER,	0.1,	“ExactMatch”,	((1, “override”)),
(type,	0.3,	“Distance”,	((1, “void”)),
(name,	0.8,	“Distance”,	((1, “Visit”)),
(arguments,	0.5,	“Distance”,	$\begin{pmatrix} (0.1, “(”), (1, “Bin”), (1, “Op”), \\ (1, “Node”), (1, “binop”), (0.1, “)”) \end{pmatrix}$)

)

Рисунок 3. Пример формирования контекста заголовка для метода

«слова» и прочие символы. Элемент списка — пара $(Priority, Text)$, где $Priority$ устанавливается равным 1 для числобуквенного текста и 0.1 для прочих символов. Также для H_a определяются операторы $\mathrm{Core}(H_a)$ и $\mathrm{NotCore}(\mathrm{H_a})$, вводящие дополнительное разделение элементов заголовка по их важности для правильной перепривязки. Операторы возвращают не пересекающиеся между собой списки элементов H_a.

I_a — *внутренний* контекст — тройка вида $(Text, Hash, Length)$, далее называемая **TextOrHash**. $Text$ — это текст внутренней части элемента (текст, соответствующий a, за вычетом областей, соответствующих непосредственным листовым потомкам a, и пробельных символов). $Hash$ — нечёткий хеш [**34**], сконструированный для этого текста. Если длина текста превышает некоторое значение, задаваемое параметром алгоритма привязки, сохраняется только нечёткий хеш, а компонента $Text$ остаётся пустой. Для коротких текстов, хеш которых невозможно сконструировать в силу технических ограничений применяемого алгоритма нечёткого хеширования, сохраняется только текст. Длина текста всегда хранится в $Length$.

S_a — контекст *областей*, хранимый как список пар $(Type, H)$. Каждый элемент соответствует одной из сущностей, объемлющих a (одному из предков узла a), и представлен её типом и заголовком.

N_a — контекст *соседей*, хранимый как пара $(Before, After)$, где обе компоненты имеют вид $(All, Nearest)$. $Before$ описывает соседей, предшествующих a; $After$ описывает соседей, следующих за a. All — это тройка **TextOrHash**, построенная для конкатенированного текста всех одноуровневых с a элементов (всех пригодных для привязки сущностей, имеющих общего с a предка), $Nearest$ — это **BindingPoint**, построенный для ближайшего в смысле расположения в тексте соседа a, имеющего тип $Type_a$. Например, если a — метод класса, в $\mathrm{All}\,(\mathrm{Before}(a))$

и All (After(a)) попадёт информация о всех членах класса, расположенных до и после a соответственно. В Nearest (Before(a)) окажется информация о ближайшем предшествующем методе, возможно, расположенном в другом классе. Аналогично заполняется Nearest (After(a)).

C_a — контекст *наиболее похожих*. Он представляет собой список из не более чем $n_C \in \mathbb{N} \cup \{0\}$ кортежей **BindingPoint**, соответствующих элементам типа $Type_a$, наиболее похожим на a в момент привязки. Например, если a — перегруженный метод, в C_a запоминаются другие его реализации.

В случае, если пользователь запрашивает переход к ранее помеченной сущности a (производится клик по соответствующему элементу в панели разметки), производится перепривязка — поиск a в актуальной версии кода. Строится АСД актуального кода программы и формируется список кандидатов — массив кортежей **BindingPoint**, соответствующих сущностям типа $Type_a$, представленным в текущей версии кода. Затем для каждого кандидата c выполняется поконтекстное сравнение $\mathbf{BindingPoint}_a$ и $\mathbf{BindingPoint}_c$ и формируется вектор расстояний — чисел в диапазоне $[0; 1]$, характеризующих непохожесть каждого из контекстов, описывающих a, на соответствующий контекст, описывающий c:

$$\begin{aligned}\mathbf{d_{ac}} = &(\mathrm{Dist}(\mathrm{Core}(H_a), \mathrm{Core}(H_c)), \mathrm{Dist}(\mathrm{NotCore}(H_a), \mathrm{NotCore}(H_c)),\\ &\mathrm{Dist}(I_a, I_c), \mathrm{Dist}(S_a, S_c), \mathrm{Dist}(\mathrm{All}(N_a), \mathrm{All}(N_c)),\\ &\mathrm{Dist}(\mathrm{Nearest}(N_a), \mathrm{Nearest}(N_c))).\end{aligned}$$

Вектор расстояний скалярно умножается на вектор весовых коэффициентов $\mathbf{w_a}$, отражающих важность каждого из контекстов для поиска помеченного элемента. Вектор весовых коэффициентов конструируется отдельно для каждого искомого элемента при помощи эвристического алгоритма. В результате получается оценка

$$\mathrm{Dist}(a, c) = \frac{\mathbf{d_{ac}} \cdot \mathbf{w_a}}{\sum\limits_{w \in \mathbf{w_a}} w}.$$

$\mathrm{Dist}(a, c) \in [0; 1]$ — расстояние между исходным элементом и кандидатом. Наилучшим считается кандидат, для которого эта оценка минимальна. Если для наилучшего кандидата c_1 и следующего за ним кандидата c_2 выполняется условие $\mathrm{Dist}(a, c_2) \neq 0 \wedge \mathrm{Dist}(a, c_1) \cdot 2 \leq \mathrm{Dist}(a, c_2)$, происходит автоматическая перепривязка и переход к c_1 в окне редактора, иначе пользователю инструмента разметки выдаётся список

кандидатов, ранжированных по возрастанию расстояния, и верное соответствие выбирается вручную.

3. Привязка к произвольному участку кода

3.1. Привязка к строке

Привязка к строке также осуществляется через привязку к некоторому узлу АСД, но предполагает запоминание дополнительной информации. Ранее введённая модель **BindingPoint** расширяется дополнительным контекстом *строки* $L = (HadSame, Inner, Outer)$. Он состоит из флага дублирования строки, внутреннего контекста и внешнего контекста соответственно. *Inner* — это тройка **TextOrHash**, построенная для текста строки, $Outer = (Before, After)$ — пара троек **TextOrHash**, построенных для конкатенированного текста всех предшествующих и всех последующих строк соответственно, расположенных в пределах той же, что и помечаемая строка, наименьшей объемлющей синтаксической сущности, которая может быть представлена как **BindingPoint**. Например, при привязке к строке внутри метода в *Before* попадёт информация о всех предшествующих строках, принадлежащих этому же методу, а в *After* — о всех последующих. *HadSame* принимает значение 1, если в момент привязки в пределах наименьшей объемлющей синтаксической сущности существуют другие строки, содержимое которых полностью совпадает с помечаемой, и 0 в противном случае.

Привязка осуществляется в два этапа. На первом этапе производится поиск наименьшей сущности a, объемлющей интересующую программиста строку, и эта сущность запоминается как $\mathbf{BindingPoint}_a$ с пустым контекстом L_a. На втором этапе конструируется L_a, описывающий нужную строку. Отметим, что наша реализация оптимизирована таким образом, чтобы избежать повторного построения и сохранения контекстов, описывающих сущность a, при привязке к другим строкам внутри неё.

Процесс переприязки также состоит из двух шагов: сначала осуществляется поиск сущности a в актуальной версии кода, затем, если произошла успешная переприязка a к некоторому c, запускается алгоритм 1. Текст, соответствующий c, разбивается на строки-кандидаты, для каждой строки-кандидата вычисляются описывающий её контекст

Алгоритм 1 Перепривязка строки

1: **function** Rebind(a, c)
2: $lines \leftarrow \text{GetLines}(c)$
3: $lineContexts \leftarrow \{line \in lines \mid \text{GetLineContext}(line)\}$
4: **for all** $L_c \in lineContexts$ **do**
5: $distances[L_c] \leftarrow (\text{Dist}(\text{Inner}(L_a), \text{Inner}(L_c)), \text{Dist}(\text{Outer}(L_a), \text{Outer}(L_c)))$
6: **end for**
7: $minInDist \leftarrow \text{Min}_{L_c \in lineContexts}(distances[L_c][1])$
8: $confusionFlag \leftarrow \text{HadSame}(L_a) \vee |\{L_c \in lineContexts \mid !\,\text{CheckGap}(minInDist, distances[L_c][1])\}| > 1$
9: $w^{\text{Inner}(L)} \leftarrow confusionFlag\,?\,MinW : MaxW$
10: $w^{\text{Outer}(L)} \leftarrow confusionFlag\,?\,MaxW : MinW$
11: **for all** $L_c \in lineContexts$ **do**
12: $totalDistances[L_c] \leftarrow \frac{distances[L_c] \cdot \left(w^{\text{Inner}(L)}, w^{\text{Outer}(L)}\right)}{\left(w^{\text{Inner}(L)} + w^{\text{Outer}(L)}\right)}$
13: **end for**
14: $orderedCandidates \leftarrow \text{OrderByAsc}\,(lineContexts, totalDistances)$
15: **return** $orderedCandidates[1]$
16: **end function**

17: **function** CheckGap(v_1, v_2)
18: **return** $v_2 \neq 0 \wedge v_1 \cdot 2 \leq v_2$
19: **end function**

L_c и вектор покомпонентных расстояний между L_a и L_c (строки 2–6 алгоритма 1). Затем в зависимости от того, существовали ли строки, идентичные помеченной, в момент привязки, и есть ли одинаковые или почти одинаковые по своему содержимому строки-кандидаты, похожие на искомую строку, эвристически корректируются веса компонент *Inner* и *Outer* (строки 7–10 алгоритма 1). $MaxW$ и $MinW$ — параметры, задающие максимальный и минимальный вес контекста. Приведённые в разделе 4 результаты получены при $MaxW = 1$ и $MinW = 0.25$. Для каждой строки-кандидата итоговое расстояние вычисляется как нормированное к диапазону $[0; 1]$ скалярное произведение вектора покомпонентных расстояний и вектора весов, после чего наилучший

кандидат считается актуальной версией искомой строки. Здесь и далее индексация массивов начинается с 1.

Расстояние между тройками u и v вида **TextOrHash**, образующими внутренний контекст строки и компоненты внешнего контекста, оценивается через приведённую в [9] функцию похожести $\mathrm{Sim}(u, v)$, связанную с функцией расстояния по формуле $\mathrm{Dist}(u, v) = 1 - \mathrm{Sim}(u, v)$. Здесь

$$\mathrm{Sim}(u,v) = \begin{cases} \theta\Big(1 - \mathrm{Dist}(\mathrm{Text}(u), \mathrm{Text}(v)), \frac{L_1}{L_2}\Big), \\ \qquad \forall k \in \{u, v\}, \mathrm{Length}(k) \le L_2 \\ \theta\Big(1 - \mathrm{TlshDist}(\mathrm{Hash}(u), \mathrm{Hash}(v)), \frac{L_1}{L_2}\Big), \\ \qquad \forall k \in \{u, v\}, \mathrm{Length}(k) \ge L_1 \\ 0, \quad \text{иначе}, \end{cases}$$

где расстояние между текстами вычисляется как нормированное к диапазону $[0; 1]$ расстояние Левенштейна [35], TlshDist — функция, возвращающая нормированное расстояние между двумя нечёткими хешами. θ — пороговая функция, возвращающая первый аргумент, если его значение больше либо равно второму аргументу, в противном случае возвращается второй аргумент. L_1 и L_2 — параметры алгоритма: L_1 — минимальная длина текста, для которой возможно вычислить нечёткий хеш; L_2 — максимальная длина текста, сохраняемого в компоненте *Text*. В наших экспериментах $L_1 = 25$ в соответствии с техническими ограничениями используемого алгоритма хеширования, $L_2 = 100$.

Для компоненты *Outer* итоговое расстояние рассчитывается по формуле

$$\begin{aligned} \mathrm{Dist}(\mathrm{Outer}(L_a), \mathrm{Outer}(L_c)) = \\ (\mathrm{Dist}(\mathrm{Before}\,(\mathrm{Outer}(L_a))\,, \mathrm{Before}\,(\mathrm{Outer}(L_c))) \\ + \mathrm{Dist}(\mathrm{After}\,(\mathrm{Outer}(L_a))\,, \mathrm{After}\,(\mathrm{Outer}(L_c))))/2\,. \end{aligned}$$

3.2. Многострочная привязка

В случае, если необходимо привязаться к многострочной области программы как к единому целому (такая область может состоять всего из нескольких строк либо быть довольно обширной, включающей несколько синтаксических сущностей), границы помечаемого фрагмента приходится указывать явно, чтобы он мог быть обнаружен на этапе

синтаксического анализа программы и соответствующий ему узел мог быть помещён в АСД. После этого можно использовать описанные в 2.2 модели и алгоритмы, применяемые для привязки к крупным синтаксическим сущностям. Отметим, что привязка к многострочным областям не является в полном смысле произвольной: в силу того, что узел, соответствующий области, нужно добавить в АСД, данная область должна удовлетворять определённым требованиям, которые будут рассмотрены далее в этом разделе.

3.2.1. *Обнаружение ограниченной многострочной области в процессе синтаксического анализа программы*

Чтобы распознать некоторым образом ограниченные многострочные фрагменты на этапе синтаксического анализа, можно доработать правила легковесной грамматики целевого языка программирования. Однако, как отмечено нами в [10], этот подход имеет ряд серьёзных недостатков, главный из которых — усложнение грамматики. Многострочные фрагменты могут выделяться в разных местах программы, например, внутри метода и на уровне членов класса. Содержимое фрагмента зависит от конкретного места, в котором он находится. Чтобы описать доступный для пометки участок, для каждого из возможных расположений необходимы отдельный нетерминальный символ и отдельная альтернатива существующего правила грамматики, делающая этот символ достижимым.

Мы придерживаемся принципиально иного подхода: единственное, что нужно сообщить легковесному синтаксическому анализатору, — какие конструкции языка выступают в роли границ для помечаемых программистом многострочных фрагментов. Для этого в грамматику на языке LanD добавляется специальная конфигурационная секция `CustomBlock`. На рисунке 4 показаны варианты описания данной секции для языка C#.

```
%CustomBlock {
    start("//+")
    end("//-")
    basetoken COMMENT
}
```

(*а*) обрамление псевдокомментариями

```
%CustomBlock {
    start("#region")
    end("#endregion")
    basetoken DIRECTIVE
}
```

(*б*) оформление в виде региона

Рисунок 4. Конфигурирование привязки к многострочным фрагментам для программы на языке C#

Опция `basetoken` задаёт токен, который может трактоваться как открывающая и закрывающая границы помечаемой многострочной области, опция `start` задаёт префикс, который должно иметь значение токена, указанного в `basetoken`, чтобы данный токен считался открывающей границей. Опция `end` имеет аналогичный смысл для закрывающей границы. Программист, размечающий код, самостоятельно задаёт содержательную часть границы многострочного фрагмента — всё, что идёт после `start` и `end` в тексте соответствующего токена. За счёт этого границы могут иметь ту же смысловую нагрузку, что и обычные лексемы типа `basetoken`. Отметим, что, во избежание коллизии между синтаксической сущностью программы и ограниченным многострочным фрагментом, токен, используемый в качестве `basetoken`, должен соответствовать чему-то, что пропускается в процессе синтаксического анализа, например, комментарию или директиве препроцессора.

На рисунке 4*а* продемонстрировано конфигурирование для случая, когда помечаемые многострочные фрагменты ограничиваются псевдокомментариями. Ниже на рисунке 5*а* показан текст программы, в котором таким способом выделено несколько многострочных фрагментов.

На рисунке 4*б* предложен другой вариант конфигурирования — помечаемые фрагменты можно заключать в регионы, таким образом разработчик может не только осуществлять привязку, но и лучше структурировать код в окне редактора.

Определение 3.1. Непрерывный участок кода, состоящий из нуля или более строк, ограниченный в соответствии с описанием секции `CustomBlock`, будем называть *обрамлённым*.

В процессе синтаксического разбора одновременно с построением АСД отслеживаются открывающие и закрывающие границы обрамлённых фрагментов. С учётом вложенностей данных фрагментов строится лес, в котором каждый узел соответствует некоторому обрамлённому участку. Затем происходит встраивание этого леса в АСД, данный процесс мы подробно описываем далее.

3.2.2. *Учёт обрамлённых фрагментов в АСД*

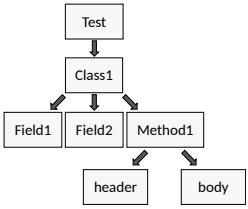
На рисунке 5 показан пример программы и структур, строящихся в процессе её разбора. В демонстрируемом случае лес обрамлённых

фрагментов состоит из одного дерева, поскольку есть фрагмент, объемлющий всю программу. По окончании разбора происходит встраивание данного леса в АСД, результатом является АСД, в котором присутствуют узлы, соответствующие обрамлённым фрагментам (рисунок 5*г*). К обрамлённым фрагментам, узлы которых встроены в АСД, можно осуществить привязку.

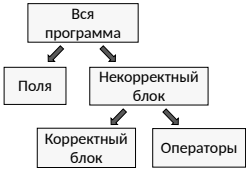
```
//+ Вся программа
namespace Test
{
    public class Class1
    {
        //+ Поля
        public int Field1 = 1;
        public double Field2 = 2;
        //- Поля

        //+ Некорректный блок
        //+ Корректный блок
        /// Комментарий к методу 1
        public void Method1()
        //- Корректный блок
        {
            //+ Операторы
            operator1;
            operator2;
            //- Операторы
        //- Некорректный блок
            operator3;
        }
    }
}
//- Вся программа
```

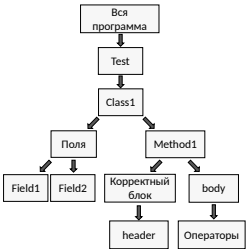
(*а*) пример программы с выделенными обрамлёнными фрагментами



(*б*) легковесное АСД программы



(*в*) дерево обрамлённых фрагментов



(*г*) АСД, полученное в результате встраивания

Рисунок 5. Встраивание дерева обрамлённых фрагментов в АСД

Внешний вид границ обрамлённого фрагмента дополнительно регламентируется нами: мы пробуем произвести встраивание фрагмента и разрешаем последующую привязку к нему, только если открывающая и закрывающая границы совпадают с точностью до префиксов, задаваемых опциями `start` и `end`, и совпадающая часть непуста. Благодаря этому инструмент разметки распознаёт потенциально ошибочные редактирования текста программы, нарушающие целостность сразу нескольких помеченных областей, и может проинформировать об этом программиста. Пример такого редактирования продемонстрирован на рисунке 6.

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ Первый фрагмент
            operator1;
            operator2;
            //- Первый фрагмент
            operator3;
            //+ Второй фрагмент
            operator4;
            operator5;
            operator6;
            //- Второй фрагмент
        }
    }
}
```

(*а*) исходная версия с двумя обрамлёнными фрагментами

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ Первый фрагмент
            operator1;
            operator5;
            operator6;
            //- Второй фрагмент
        }
    }
}
```

(*б*) актуальная версия

Рисунок 6. Потенциально опасное редактирование программы

Помимо ограничения, указанного выше, мы накладываем на обрамлённый фрагмент ряд условий, обязательных для успешного встраивания в АСД. Эти условия гарантируют, что обрамлённый фрагмент согласуется с синтаксической структурой программы. Все они отражены в определении 3.5.

Пусть a и b — узлы одного или различных деревьев, построенных по тексту одной и той же программы. Определим несколько отношений на множестве узлов этих деревьев.

Определение 3.2. Будем говорить, что узел a *вкладывается* в узел b, и обозначать этот факт как $a \subseteq b$ тогда и только тогда, когда область текста программы, соответствующая узлу a, строго вкладывается в область текста программы, соответствующую узлу b, или совпадает с ней.

По аналогии вводится отношение строгого вложения узлов, обозначаемого как $a \subset b$, и текстового совпадения узлов, обозначаемого как $\overset{\mathrm{T}}{=}$. $a \subseteq b \equiv a \subset b \vee a \overset{\mathrm{T}}{=} b$. Отметим, что текстово совпадающие узлы могут не совпадать в обычном смысле, то есть, могут являться двумя различными узлами одного или разных деревьев.

Определение 3.3. Будем говорить, что узел a *пересекается* с узлом b, и обозначать этот факт как $a \sqcap b$ тогда и только тогда, когда существует

непустая область текста программы, вложенная и в область текста, соответствующую a, и в область текста, соответствующую b.

Определение 3.4. Будем говорить, что узел a *строго пересекается* с узлом b, и обозначать этот факт как $a \mathbin{!\sqcap} b$ тогда и только тогда, когда $a \sqcap b$ и ни один узел из данной пары не вложен в другой.

Определение 3.5. Обрамлённый фрагмент кода будем называть *пользовательским блоком* тогда и только тогда, когда его открывающая и закрывающая границы совпадают с точностью до префиксов, задаваемых опциями `start` и `end`, совпадающая часть этих границ непуста, и соответствующий ему узел f в дереве, принадлежащем лесу обрамлённых фрагментов, удовлетворяет одному из *условий встраивания*:

(*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.

Первое условие встраивания позволяет добавить f в АСД, сделав его новым корнем дерева. На рисунке 7*а* показаны два пользовательских блока, узлы которых можно встроить в АСД программы в качестве корня и его непосредственного потомка. Обратите внимание на то, что область текста, соответствующая пользовательскому блоку, может быть существенно больше области, соответствующей корню АСД, поскольку пустые строки и комментарии игнорируются синтаксическим анализатором, но могут войти в состав пользовательского блока. В приведённом примере корню АСД до встраивания пользовательских блоков соответствует текст с 8 по 15 строку. Второе условие описывает тот узел n, принадлежащий АСД, к непосредственным потомкам которого можно добавить узел f, не соответствующий первому условию. Как и в случае с первым условием встраивания, области, пропускаемые синтаксическим анализатором, могут быть включены в помеченный фрагмент, поскольку определение пользовательского блока допускает не только строгую вложенность f в потенциальных предков, но и строгое пересечение с ними. На рисунке 7*б* показаны два пользовательских блока, выходящие за пределы метода `Method1`.

На рисунке 5 продемонстрировано, как аналогичный пользовательский блок (`Корректный блок`) встраивается в АСД. Очевидно, что в этом

```
1|
2|  //+ Область 2
3|
4|  // Copyright (c) Alexey Goloveshkin
5|  // This code is distributed under
6|
7|  //+ Область 1
8|  namespace Test
9|  {
10|   public class Class1
11|   {
12|      public int Field1 = 1;
13|      public double Field2 = 2;
14|   }
15|  }
16|  //- Область 1
17|
18|  //- Область 2
```

(*а*) с охватом всей программы

```
1|  namespace Test
2|  {
3|    public class Class1
4|    {
5|      public int Field1 = 1;
6|
7|      //+ Область 2
8|      /// <summary>
9|      //// Метод 1
10|     /// </summary>
11|     /// <param name="n">Целое число</param>
12|     //+ Область 1
13|     public void Method1(int n)
14|     //- Область 1
15|     //- Область 2
16|     {
17|       if (n > 0)
18|       {
19|         System.Console.WriteLine("N > 0");
20|       }
21|     }
22|   }
23|  }
```

(*б*) с охватом заголовка метода

Рисунок 7. Примеры пользовательских блоков

случае расширяется область текста, относимая к его родительскому узлу `Method1`.

Проверку второго условия и встраивание соответствующих этому условию узлов осуществляет алгоритм 2, запускаемый рекурсивно в процессе префиксного обхода АСД. В алгоритме входной параметр n — текущий узел АСД, в параметре *fragments* при посещении корня передаётся список корней всех деревьев из леса обрамлённых фрагментов. Выполнение условия 3.5.2 проверяется для обрамлённых фрагментов постепенно в процессе спуска по АСД. Одновременно со спуском по АСД происходит спуск по деревьям из леса обрамлённых фрагментов: при посещении очередного узла n в *fragments* передаются корни тех поддеревьев леса, узлы которых потенциально могут быть встроены в АСД в поддерево, корнем которого является n. Спуск по некоторому дереву из леса означает, что ранее рассматриваемый узел этого дерева был либо признан полностью удовлетворяющим условию 3.5.2 и встроен в АСД, либо отброшен из-за нарушения условия. В алгоритме 2 используется функция Children(a), обозначающая последовательность всех непосредственных потомков некоторого узла a, принадлежащего АСД либо лесу обрамлённых фрагментов.

Алгоритм 2 Встраивание узлов, соответствующих пользовательским блокам, в АСД

```
1: procedure Visit(n, fragments)
2:     while |fragments| > 0 do
3:         inserted ← ∅
4:         for all f ∈ fragments do
5:             if {c ∈ Children(n) | f !⊓ c ∨ f ⊂ c} = ∅ then
6:                 innerChildren ← [c ∈ Children(n) | c ⊆ f]
7:                 if |innerChildren| > 0 then
8:                     index ← IndexOf(innerChildren[1], Children(n))
9:                 else
10:                    index ← 0
11:                    for i ← 1, | Children(n)| do
12:                        if Children(n)[i].StartOffset > f.EndOffset then
13:                            index ← i
14:                            break
15:                        end if
16:                    end for
17:                    if index = 0 then
18:                        index ← | Children(n)| + 1
19:                    end if
20:                end if
21:                inserted ← inserted ∪ {f};   fCopy ← f
22:                SetChildren(fCopy, innerChildren)
23:                SetChildren(n, children \ innerChildren)
24:                InsertChild(n, fCopy, index)
25:            end if
26:        end for
27:        for all f ∈ inserted do
28:            index ← IndexOf(f, fragments)
29:            RemoveAt(fragments, index)
30:            InsertAt(fragments, Children(f), index)
31:        end for
32:        for all c ∈ Children(n) do
33:            innerFragments ←
   {f ∈ fragments | f !⊓ c ∧ ∀c′ ∈ Children(n) \ {c}, ¬(c′ ⊓ f) ∨ f ⊂ c}
34:            fragments ← fragments \ innerFragments
35:            Visit(c, innerFragments)
36:        end for
37:        fragments ← ⋃_{f∈fragments} Children(f)
38:    end while
39: end procedure
```

На первом этапе (строки 4–26 алгоритма 2) для каждого узла f из *fragments* проверяется, можно ли встроить его в АСД в качестве непосредственного потомка n. В случае, если f строго пересекается с каким-то из непосредственных потомков n или вложен в него, такое встраивание произвести нельзя; иначе фрагмент, соответствующий f, признаётся пользовательским блоком и отбираются все непосредственные потомки n, вложенные в f или совпадающие с ним. В результате получается некоторая подпоследовательность последовательности $\mathrm{Children}(n)$, возможно, пустая (строка 6 алгоритма 2). Её элементы становятся новыми непосредственными потомками f, сам f добавляется вместо этой подпоследовательности к непосредственным потомкам n. Если непосредственных потомков n, вложенных в f, не существует, f вставляется между такими непосредственными потомками n, что текст предшествующего потомка предшествует тексту f, а текст следующего следует за текстом f. Используемая в алгоритме процедура InsertChild принимает в качестве первого аргумента узел, к потомкам которого в позицию, задаваемую третьим аргументом, нужно добавить второй аргумент. По окончании цикла старые непосредственные потомки всех вставленных в АСД узлов f — элементы дерева обрамлённых фрагментов — добавляются к набору *fragments* (строки 27–31 алгоритма 2) вместо соответствующих f.

На втором этапе (строки 32–36 алгоритма 2) отбираются обрамлённые фрагменты, возможно, являющиеся пользовательскими блоками — те, что строго пересекаются ровно с одним непосредственным потомком n или строго вложены в такого потомка (строка 33 алгоритма 2). Соответствующий список передаётся в качестве второго параметра в рекурсивный вызов метода Visit для этого потомка. В итоге в массиве *fragments* остаются узлы, которые точно не удастся встроить в АСД. Однако это не означает, что не удастся встроить узлы, принадлежащие их поддеревьям. Для них проверку нужно произвести отдельно. Все узлы, оставшиеся в *fragments*, заменяются на последовательности своих непосредственных потомков (строка 37 алгоритма 2), после чего процесс повторяется.

Для абстрактного синтаксического дерева, содержащего m узлов, и леса обрамлённых фрагментов, содержащего l узлов, сложность алгоритма в худшем случае составляет $\mathcal{O}(m \cdot l)$.

Очевидно, что алгоритм 2 позволяет добавить в АСД в качестве потомка некоторого узла n любой узел f, соответствующий второму условию встраивания: при прохождении метода Visit через предков n такой узел f попадает (непосредственно либо в составе поддерева,

корень которого находится в *fragments*) в рекурсивные вызовы Visit, а при посещении n сам узел f или некоторый его предок из дерева обрамлённых фрагментов встраивается в АСД как охватывающий одного или нескольких непосредственных потомков n либо как не пересекающийся ни с одним из непосредственных потомков n. В случае, если на текущем шаге встраивается не сам f, а его предок из дерева обрамлённых фрагментов, обход АСД будет продолжен с учётом произошедших в нём изменений, и встраивание f произойдёт при рекурсивном посещении узла, соответствующего этому предку.

На рисунке 5а обрамлённый фрагмент `Некорректный блок` не является пользовательским блоком, поскольку соответствующий ему узел вложен во все узлы АСД в цепочке от корня до `Method1`, однако на уровне потомков узла `Method1` он включает в себя заголовок — узел `header` — и одновременно строго пересекается с телом метода — узлом `body`. В результате `Некорректный блок` не представлен в итоговом АСД, показанном на рисунке 5г.

Инструмент разметки автоматически проверяет условия встраивания, если программист хочет пометить некоторый многострочный фрагмент. Создать обрамлённый фрагмент `Некорректный блок` на рисунке 5а при помощи инструмента разметки невозможно: при попытке привязаться к соответствующему участку кода будет предложена только привязка к объемлющему классу `Class1`.

3.2.3. *Корректность определения пользовательского блока*

Понятие пользовательского блока введено нами таким образом, чтобы при встраивании пользовательских блоков в АСД сохранялась корректность АСД. Под корректностью АСД мы понимаем выполнение его фундаментальных свойств: для любых узлов n_1 и n_2, принадлежащих АСД и не совпадающих друг с другом,

(*1*) n_1 — предок $n_2 \implies n_2 \subseteq n_1$;

(*2*) $n_1 \sqcap n_2 \iff n_1$ — предок n_2 или n_2 — предок n_1.

Утверждение 3.1. 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.

Доказательство. Встраивание в АСД узлов, соответствующих условию 3.5.1, не влияет на связи внутри изначального АСД и не изменяет области текста, соответствующие узлам, присутствовавшим в изначальном АСД. Для узлов, встраиваемых в АСД в соответствии с условием 3.5.2, справедливость всех частей утверждения следует из алгоритма 2. □

Отметим, что узлы n_1 и n_2, связанные отношением $\subset$ до встраивания f в АСД, могут быть связаны отношением $\overset{\mathrm{T}}{=}$ после встраивания, если f стал их потомком. Как было отмечено ранее, области текста, относимые к узлам АСД, являющимся предкам f, могут расширяться в результате встраивания f, так как должны охватывать всех потомков.

Утверждение 3.2. 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'.

Доказательство. Для узла f, встроенного по причине выполнения условия 3.5.1, предки отсутствуют, а старый корень АСД вложен в f и является его потомком, как и все остальные узлы старого АСД. Следовательно, все части утверждения выполняются. Для узлов, встраиваемых в АСД в соответствии с условием 3.5.2, справедливость всех частей утверждения следует из алгоритма 2. □

Утверждение 3.1 означает, что встраивание в АСД узла, соответствующего пользовательскому блоку, не нарушает отношения между узлами этого АСД, существовавшие до встраивания. Утверждение 3.2 означает, что сам узел f корректным образом включается в отношения с исходными узлами АСД.

4. Эксперименты

Поскольку привязка к многострочным фрагментам кода осуществляется так же, как и к синтаксическим сущностям, результаты экспериментов, приведённые в [9], можно распространить на успешность перепривязки пользовательских блоков. Для классов, методов, полей и свойств нами показано, что помеченные и впоследствии отредактированные элементы успешно находятся в 98,93–100% случаев.

Для проверки привязки к однострочным фрагментам нами выполнена массовая привязка к случайно выбранным строкам в коде крупных проектов с открытым исходным кодом: *ASP.NET Core* (*asp*), *PascalABC.NET* (*pabc*), *Roslyn* (*ros*). В таблице 1 приведены более подробные сведения для каждого из них. Основным языком всех проектов является C#.

Для каждого проекта некоторая версия кодовой базы выбирается в качестве исходной, затем производится переход на некоторое количество коммитов вперёд и соответствующая версия рассматривается как актуальная. В исходной версии каждого проекта случайным образом отбираются методы (300 для *asp*, 100 для *pabc* и 300 для *ros*), которые были отредактированы в актуальной версии и успешно перепривязаны нашим инструментом разметки. Дополнительно проверяется, что в теле метода есть две и более строки. К двум случайно выбранным строкам в каждом методе производится привязка. Ожидаемым результатом является успешная перепривязка всех строк, по-прежнему существующих в актуальной версии метода. Требование успешной перепривязки самих методов позволяет исключить влияние алгоритма перепривязки крупных синтаксических сущностей на результаты настоящего эксперимента.

Таблица 1. Анализируемые проекты

	Дата исходной версии	Дата актуальной версии	Коммитов	Изменено файлов
asp	03.02.2020	04.07.2020	3551	1398
pabc	25.05.2018	26.05.2020	1462	335
ros	03.02.2021	30.04.2021	3459	1749

В таблице 2 приведены результаты эксперимента. Для каждого из проектов помеченные строки разделены на две группы: «хорошие» строки — все, кроме пустых строк и строк с одиночными операторными скобками (`{` или `}`), — и «плохие» — собственно строки с одиночными операторными скобками либо пустые. В первом подстолбце для каждого проекта приведена статистика для «хороших» строк, во втором — для «плохих». Пустые строки и строки с операторными скобками выделены в отдельную группу, поскольку они почти всегда не являются уникальными в пределах метода, из-за чего для них успешность поиска может быть значительно меньше, чем для остальных строк (в то же время вероятность того, что в реальной ситуации программист решит осуществить к ним привязку, также ниже).

В строке таблицы «Всего» приведено общее количество помеченных строк, в «Перепривязано, верно» показано количество случаев, когда

ранее помеченная строка по-прежнему присутствует в коде и корректно перепривязывается, в строке «Перепривязано, неверно» указано количество строк, для которых актуальные версии по-прежнему присутствуют в коде, однако перепривязка происходит не к ним. В строке «Найдено место, верно» указано количество случаев, когда искомая строка была удалена и произошла перепривязка к месту, расположенному в пределах 1–2 строк от прежнего места расположения удалённой строки. В «Найдено место, неверно» считаются случаи, когда искомая строка была удалена и произошла перепривязка к строке, расположенной дальше 1–2 строк от прежнего места. В строке «Успешность» посчитан процент успешно перепривязанных строк от общего количества неудалённых помеченных строк.

Таблица 2. Результаты эксперимента

	asp		*pabc*		*ros*	
Всего	433	167	155	45	435	165
Перепривязано, верно	410	151	147	32	414	144
Перепривязано, неверно	2	4	2	11	8	12
Найдено место, верно	18	9	4	1	10	6
Найдено место, неверно	3	3	2	1	3	3
Успешность, %	99,51	97,42	98,66	74,42	98,1	92,31

Существенная разница между результатами перепривязки «хороших» и «плохих» строк действительно наблюдается, в особенности для проекта *pabc*, где успешность перепривязки неудалённых «плохих» строк составляет 74,42%, в то время как для остальных строк успешность составляет 98,66%. Подробный анализ показал, что, поскольку для «плохих» строк ключевую роль в успешной перепривязке играет внешний контекст, рядом расположенные одинаковые «плохие» строки могут стать причиной путаницы даже в случае, когда окружение редактируется незначительно. Также для них оказывается существенным то, что оценка похожести внешнего контекста ухудшается из-за изменения сигнатуры метода: в силу легковесности разбора внешний контекст строки, расположенной в теле метода, захватывает не содержимое тела, а весь текст метода. Более тонкая обработка подобных ситуаций позволила бы повысить процент успешных перепривязок.

В случае «хороших» (содержательных) строк неуспешность перепривязки в первую очередь обусловлена сильным редактированием их самих. На рисунках 8 и 9 продемонстрированы два из восьми ошибочных случаев, имевших место для «хороших» строк в *ros*. На каждом рисунке серым цветом в а) и б) выделены исходная помеченная строка и строка, к которой была осуществлена перепривязка, соответственно.

В рамку с прозрачным фоном взята строка, перепривязка к которой была бы корректной.

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    if (symbol.Locations.Contains(location))
    {
        return symbol;
    }
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(*а*) исходная версия кода

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    // The token may be part of a larger name (for example, `int` in `public static operator int[](Goo g);`.
    // So check if the symbol's location encompasses the span of the token we're asking about.
    if (symbol.Locations.Any(loc => loc.SourceTree == location.SourceTree && loc.SourceSpan.Contains(location.SourceSpan)))
        return symbol;
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(*б*) актуальная версия кода

Рисунок 8. Пример неправильной перепривязки строки в *ros*

```
using (var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync)))
{
    // Let the presented know we're starting a search.
    var context = presenter.StartSearch(
        EditorFeaturesResources.Navigating, supportsReferences: true);
    using (Logger.LogBlock(
        FunctionId.CommandHandler_FindAllReference,
        KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
        context.CancellationToken))
    {
```

(*а*) исходная версия кода

```
using var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync));

// Let the presented know we're starting a search.  We pass in no cancellation token here as this
// operation itself is fire-and-forget and the user won't cancel the operation through us (though
// the window itself can cancel the operation if it is taken over for another find operation.
var (context, cancellationToken) = presenter.StartSearch(EditorFeaturesResources.Navigating, supportsReferences: true);
using (Logger.LogBlock(
    FunctionId.CommandHandler_FindAllReference,
    KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
    cancellationToken))
{
```

(*б*) актуальная версия кода

Рисунок 9. Пример неправильной перепривязки строки в *ros*

Стоит отметить, что иногда удаление строки из метода не означает, что она исчезла совсем: в каждом из проектов обнаружены случаи, когда часть метода выделяется в отдельный метод или создаётся перегруженная версия метода, куда выносится его содержимое, а

старый метод вызывает свою перегруженную версию. На уровне синтаксических сущностей в этом случае происходит перепривязка к методу со старой сигнатурой, однако он не содержит ранее помеченных и впоследствии вынесенных из него строк. Эти случаи учитываются в строках таблицы «Найдено место», прежним местом считается вызов метода, в котором сейчас находится искомая строка. Очевидно, что в подобных ситуациях перепривязку можно выполнить точнее, если отсутствие хороших строк-кандидатов будет влиять на перепривязку объемлющей сущности.

Заключение

В ходе работы над проектом программист нуждается в инструменте, который позволил бы ему помечать важные в рамках текущей задачи участки кода, использовать ранее созданные пометки для быстрого возврата к работе над другими задачами, обмениваться данной информацией с коллегами. Ключевыми проблемами при создании такого инструмента являются привязка к коду — запоминание некоторого места в нём — и перепривязка — поиск ранее запомненного места в актуальной версии кода. Критически важным свойством привязки к коду является устойчивость к редактированию: перепривязка должна происходить успешно независимо от того, насколько сильно отредактирован искомый фрагмент.

В настоящей работе предложено развитие разрабатываемого авторами подхода к привязке к коду, основанного на привязке к узлам легковесного абстрактного синтаксического дерева программы.

Ранее созданные модели и алгоритмы, предназначенные для устойчивой привязки к крупным синтаксическим сущностям программы, расширены и дополнены для обеспечения устойчивой привязки к произвольной строке кода, вложенной в некоторую синтаксическую сущность. Для привязки к многострочному фрагменту предложены необходимая формализация и алгоритм, позволяющий проверить корректность выделения многострочного фрагмента относительно синтаксической структуры программы и осуществить встраивание узла, соответствующего этому фрагменту, в АСД. Приведены результаты экспериментов, демонстрирующие успешность перепривязки для ранее помеченных произвольных участков кода.

В качестве дальнейших направлений исследования рассматриваются улучшение привязки к однострочным участкам путём более сложного учёта описывающих строку контекстов, а также более тесная интеграция

инструмента разметки со средой разработки и учёт фактических сценариев редактирования кода при перепривязке.

Вклад авторов: А. В. Головешкин — 85% (методология исследований, программирование, валидация, формальный анализ, проведение экспериментов, ресурсы, написание черновой версии, доработка и редактирование, визуализация кода); С. С. Михалкович — 15% (постановка задачи, доработка и редактирование, наставничество, администрирование).

Список литературы

[1] Schröer M., Koschke R. *Recording, visualising and understanding developer programming behaviour*, 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (11 May 2021, Honolulu, HI, USA).– 2021.– pp. 561–566. ↑4

[2] Ciborowska A., Chakarov A., Pandita R. *Contemporary COBOL: developers' perspectives on defects and defect location*, 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (27 Sept.–1 Oct. 2021, Luxembourg).– 2021.– pp. 227–238. ↑4

[3] Parnin C., Rugaber S. *Resumption strategies for interrupted programming tasks* // Software Quality Journal.– 2011.– Vol. **19**.– No. 1.– pp. 5–34. ↑4

[4] Badreddin O., Khandoker R., Forward A., Masmali O., Lethbridge T. C. *A decade of software design and modeling: a survey to uncover trends of the practice* // *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS'18 (14–19 October 2018, Copenhagen, Denmark), New York, NY: ACM.– 2018.– ISBN 978-1-4503-4949-9.– pp. 245–255. ↑4

[5] Storey M.-A., Ryall J., Singer J., Myers D., Cheng L.-T., Muller M. *How software developers use tagging to support reminding and refinding* // IEEE Trans. Softw. Eng.– 2009.– Vol. **35**.– No. 4.– pp. 470–483. ↑4

[6] Guzzi A., Hattori L., Lanza M., Pinzger M., A. van Deursen *Collective code bookmarks for program comprehension*, 2011 IEEE 19th International Conference on Program Comprehension (22–24 June 2011, Kingston, ON, Canada).– 2011.– pp. 101–110. ↑4

[7] Goloveshkin A. V., Mikhalkovich S. S. *Tolerant parsing with a special kind of "Any" symbol: the algorithm and practical application* // Proc. ISP RAS.– 2018.– Vol. **30**.– No. 4.– pp. 7–28. ↑4, 9

[8] Goloveshkin A. V. *Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded "Any" symbol* // Proc. ISP RAS.– 2019.– Vol. **31**.– No. 3.– pp. 7–28. ↑4, 9

[9] Goloveshkin A., Mikhalkovich S. *Using improved context-based code description for robust algorithmic binding to changing code* // Procedia Computer Science.– 2021.– Vol. **193**.– pp. 239–249. ↑4, 5, 7, 10, 15, 25

[10] Головешкин А. В., Михалкович С. С. *Привязка к произвольному участку программы в задаче разметки программного кода // Современные информационные технологии: тенденции и перспективы развития*, Материалы XXVI научной конференции (18-–19 апреля 2019 г., Южный федеральный университет, Ростов-на-Дону, Россия), Ростов-на-Дону–Таганрог: Южный федеральный университет.– 2019.– ISBN 978-5-9275-3139-4.– с. 86–89. ↑5, 16

[11] Фуксман А. Л. *Технологические аспекты создания программных систем.*– М.: Статистика.– 1979. ↑6

[12] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. *An overview of AspectJ // ECOOP 2001 — Object-Oriented Programming*, 15th European Conference (18–22 June 2001, Budapest, Hungary), LNCS.– vol. **2072**, Berlin: Springer-Verlag.– 2001.– ISBN 978-3-540-45337-6.– pp. 327–354. doi ↑6

[13] Schaefer I., Bettini L., Bono V., Damiani F., Tanzarella N. *Delta-oriented programming of software product lines // Software Product Lines: Going Beyond*, 14th International Conference SPLC 2010 (13–17 September 2010, Jeju Island, South Korea), LNCS.– vol. **6287**, Berlin: Springer-Verlag.– 2010.– ISBN 978-3-642-15578-9.– pp. 77–91. doi ↑6

[14] Apel S., Kästner C., Lengauer C. *Language-independent and automated software composition: the FeatureHouse experience* // IEEE Trans. Softw. Eng.– 2013.– Vol. **39**.– No. 1.– pp. 63–79. doi ↑6

[15] Kanai S. *Customize code generation using IBM Rational Rhapsody for C++*.– IBM Support. URL ↑7

[16] Калинин С. Ю., Колоколов И. А., Литвиненко А. Н. *Применение концепций АОП в разработке расширяемых приложений* // Известия Южного федерального университета. Технические науки.– 2010.– Т. **103**.– № 2.– с. 58–68. ✻ ↑7

[17] Малеванный М. С., Михалкович С. С. *Поддержка среды программирования для навигации по аспектам программного кода // Современные информационные технологии: тенденции и перспективы развития*, Материалы конференции (17–18 апреля 2014 г., Южный федеральный университет, Ростов-на-Дону, Россия), Ростов-на-Дону: Южный федеральный университет.– 2014.– ISBN 978-5-9275-1227-0.– с. 275–276. ↑

[18] Malevannyy M. S., Mikhalkovich S. S. *Context-based model for concern markup of a source code* // Proc. ISP RAS.– 2016.– Vol. **28**.– No. 2.– pp. 63–78. doi ↑7

[19] Malevannyy M. S., Mikhalkovich S. S. *Robust binding to syntactic elements in a changing code // Proceedings of the 12th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR'16 (28–29 October 2016, Moscow, Russia), New York, NY, USA: ACM.– 2016.– ISBN 978-1-4503-4884-3.– 8 pp. doi ↑7

[20] Roy C. K., Cordy J. R., Koschke R. *Comparison and evaluation of code clone detection techniques and tools: a qualitative approach* // Science of Computer Programming.– 2009.– Vol. **74**.– No. 7.– pp. 470–495. doi ↑7

[21] Rattan D., Bhatia R., Singh M. *Software clone detection: a systematic review* // Information and Software Technology.– 2013.– Vol. **55**.– No. 7.– pp. 1165–1199. ↑7

[22] Walker A., Cerny T., Song E. *Open-source tools and benchmarks for code-clone detection: past, present, and future trends* // SIGAPP Appl. Comput. Rev.– 2019.– Vol. **19**.– No. 4.– pp. 28–39. ↑7

[23] Novak M., Joy M., Kermek D. *Source-code similarity detection and detection tools used in academia: a systematic review* // ACM Trans. Comput. Educ.– 2019.– Vol. **19**.– No. 3.– 27.– 37 pp. ↑7

[24] Cheers H., Lin Y., Smith S. P. *Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications* // Empirical Software Engineering.– 2021.– Vol. **26**.– 83.– 62 pp. ↑7

[25] Luan S., Yang D., Barnaby C., Sen K., Chandra S. *Aroma: code recommendation via structural code search* // Proc. ACM Program. Lang.– 2019.– Vol. **3**.– No. OOPSLA.– 152.– 28 pp. ↑7

[26] Dotzler G., Philippsen M. *Move-optimized source code tree differencing* // *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE'16 (3–7 September 2016, Singapore), New York, NY, USA: ACM.– 2016.– ISBN 978-1-4503-3845-5.– pp. 660–671. ↑7

[27] Frick V., Grassauer T., Beck F., Pinzger M. *Generating accurate and compact edit scripts using tree differencing*, 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (23–29 Sept. 2018, Madrid, Spain).– pp. 264–274. ↑7

[28] Kim M., Notkin D., Grossman D., Wilson G. *Identifying and summarizing systematic code changes via rule inference* // IEEE Transactions on Software Engineering.– 2013.– Vol. **39**.– No. 1.– pp. 45–62. ↑7

[29] Ragkhitwetsagul C., Krinke J., Clark D. *A comparison of code similarity analysers* // Empirical Software Engineering.– 2018.– Vol. **23**.– No. 4.– pp. 2464–2519. ↑7

[30] Brody S., Alon U., Yahav E. *A structural model for contextual code changes* // Proc. ACM Program. Lang.– 2020.– Vol. **4**.– No. OOPSLA.– 215.– 28 pp. ↑7

[31] Asaduzzaman M., Roy C. K., Schneider K. A., Penta M. D. *LHDiff: a language-independent hybrid approach for tracking source code lines* // *2013 IEEE International Conference on Software Maintenance* (22–28 Sept. 2013, Eindhoven, Netherlands).– 2013.– ISBN 978-0-7695-4981-1.– pp. 230–239. ↑8

[32] Moonen L. *Generating robust parsers using island grammars* // *Proceedings of the 8th Working Conference on Reverse Engineering* (2–5 Oct. 2001, Stuttgart, Germany), Washington, DC, USA: IEEE Computer Society.– 2001.– ISBN 0-7695-1303-4.– pp. 13–22. ↑9

[33] Van den Brand M. G. J., Sellink M. P. A., Verhoef C. *Obtaining a COBOL grammar from legacy code for reengineering purposes* // *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications*

(25–26 September 1997, Amsterdam, The Netherlands), eWiC, Swindon, UK: BCS Learning & Development Ltd..– 1997.– ISBN 3-540-76228-0.– pp. 1–16. ↑9

[34] Oliver J., Cheng C., Chen Y. *TLSH – a locality sensitive hash* // *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*, CTC'13 (21–22 Nov. 2013, Sydney, NSW, Australia), Washington, DC, USA: IEEE Computer Society.– 2013.– pp. 7–13. ↑11

[35] Левенштейн В. И. *Двоичные коды с исправлением выпадений, вставок и замещений символов* // Доклады Академии Наук СССР.– 1965.– Т. **163**.– № 4.– с. 845–848. ↑15

Поступила в редакцию 28.12.2021;
одобрена после рецензирования 08.02.2022;
принята к публикации 19.02.2022.

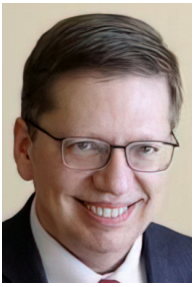
Рекомендовал к публикации *Анд. В. Климов*

Информация об авторах:

Алексей Валерьевич Головешкин

В 2015 году получил степень магистра по направлению "Фундаментальная информатика и информационные технологии в 2019 году завершил программу подготовки научно-педагогических кадров в аспирантуре по направлению "Информатика и вычислительная техника". Область интересов: языки программирования, анализ эволюции программного обеспечения, программная инженерия.

0000-0001-6947-0594
e-mail: alexeyvale@gmail.com



Станислав Станиславович Михалкович

Кандидат физ.-мат. наук, доцент, заведующий кафедрой информатики и вычислительного эксперимента Института математики, механики и компьютерных наук им. И.И. Воровича Южного федерального университета, руководитель проекта PascalABC.NET.

0000-0003-0373-3886
e-mail: miks@sfedu.ru