ИПМ им.М.В.Келдыша РАН

Абрау-2019 • Труды конференции

А.В. Головешкин, С.С. Михалкович

Разметка сквозных функциональностей в коде программы

Рекомендуемая форма библиографической ссылки

Головешкин А.В., Михалкович С.С. Разметка сквозных функциональностей в коде программы // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции (23-28 сентября 2019 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2019. — С. 245-256. — URL: http://keldysh.ru/abrau/2019/theses/83.pdf doi:10.20948/abrau-2019-83

Размещена также презентация к докладу

Разметка сквозных функциональностей в коде программы

А.В. Головешкин, С.С. Михалкович

Южный федеральный университет, Институт математики, механики и компьютерных наук им. И.И. Воровича

Аннотация. Информация о функциональностях — проблемах, решаемых программой, и совокупностях участков кода, реализующих соответствующие решения, — не хранится в исходном коде программного проекта. Если для реализации функциональности служит несколько разрозненных участков кода, каждый раз при её модификации их приходится искать заново. Для упрощения работы с рассредоточенными функциональностями требуется интегрированный в среду разработки инструмент, позволяющий пометить части каждой такой функциональности и осуществлять навигацию между ними. Отдельным требованием к инструменту является устойчивость производимой разметки относительно изменений, вносимых в размеченный код. В настоящей работе описывается алгоритмический подход к решению задачи устойчивой разметки, приводятся примеры применения инструмента, основанного на этом подходе, демонстрирующие превосходство над ближайшими аналогами.

Ключевые слова: рассредоточенная функциональность, прорезающая функциональность, сквозная функциональность, разметка программы, алгоритмическая привязка к коду

Marking up crosscutting concerns in a program code

A.V. Goloveshkin, S.S. Mikhalkovich

Southern Federal University, I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science

Abstract. Problems solved with a program, as well as parts of code implementing that solutions, are usually called “concerns”. The knowledge of program concerns is not stored in a source code. In case several code fragments are intended to solve a single problem, one needs to explore all the code attempting to find all these fragments to modify a concern after a certain time. To simplify development process, a tool is supposed to be embedded into an integrated development environment to add the capability to mark all the code parts involved in concern implementation and navigate between them. An important requirement for such a tool is the markup sustainability. In the paper, an algorithmic approach to the markup task is described, and examples

are given to demonstrate the advantage of the corresponding tool over its closest analogues.

Keywords: scattered concern, crosscutting concern, program markup, algorithmic binding to code

1. Введение

Любая программа реализует одну или несколько *функциональностей*. Словом «функциональность» (concern) обозначается как проблема, решаемая некоторой совокупностью фрагментов программного кода, так и сама эта совокупность. В случае, если функциональность реализуется фрагментами кода, рассредоточенными между элементами декомпозиции программы — файлами, классами, методами, — она называется *сквозной* или *прорезающей* (crosscutting) [1]. Конкретное назначение таких фрагментов и сам факт того, что они решают одну проблему — это семантическая информация, существующая вне кодовой базы. Если по прошествии времени возникает необходимость внести изменения в прорезающую функциональность, её части приходится искать заново, исследуя код. Это существенно замедляет и усложняет разработку.

Первой фундаментальной отечественной работой, посвящённой разработке программ, содержащих прорезающие функциональности, является работа А.Л. Фуксмана [2]. Наряду со многими новаторскими для своего времени идеями, автором обозначен возможный способ упрощения поиска и модификации прорезающих функциональностей — *разметка* кода на их составляющие. В последующем цикле работ М.М. Горбунова-Посадова [3–5] предлагается концептуальная модель организации прорезающего функционала, а также отмечается важность поддержки средой разработки как горизонтального (образованного элементами декомпозиции), так и вертикального (образованного сквозными функциональностями) слоения программы. Настоящее исследование является дальнейшим развитием исследований [2-5].

В соответствии с нашим определением, разметка — семантическая метаинформация о представленных в проекте функциональностях, хранимая вместе с проектом, но отдельно от кода. Функциональность в разметке представлена набором гиперссылок на реализующие её участки кода. В среду разработки должен быть интегрирован инструмент, позволяющий пометить участки кода — *привязаться* к ним — и впоследствии использовать эту разметку для навигации по текущей версии проекта.

В разделе 2 настоящей работы анализируются описанные в литературе способы разметки кода, в разделе 3 описывается применяемая нами модель хранения информации о помеченных участках программы (*точках привязки*), раздел 4 посвящён алгоритмам поиска точки привязки, в разделе 5 приводятся примеры программ, для которых улучшенные модели и алгоритмы позволяют инструменту разметки автоматически найти ранее помеченный участок в изменённом коде. Основным вкладом данной работы являются улучшенная относительно существующей модель хранения точек привязки и алгоритмы

поиска помеченной сущности в актуальной версии программы, использующие преимущества данной модели.

2. Существующие способы разметки кода

В литературе выделяется несколько способов пометки участка кода как принадлежащего некоторой функциональности. Первый способ заключается в обозначении границ участка при помощи комментариев специального вида, однозначно его идентифицирующих [6]. Такой подход устойчив относительно редактирования кода, однако приводит к «замусориванию» исходников информацией, не несущей никакой смысловой нагрузки с точки зрения разработчика. Одновременно с этим случайное удаление или редактирование комментариев-границ гарантированно ведёт к нарушению разметки. Второй способ — указание текстовых координат (номера строки и столбца) начала нужного участка [2]. Он является абсолютно неустойчивым к редактированию файла и потому может быть применён только при условии, что любое редактирование кода происходит в пределах среды разработки, которая отслеживает характер изменений.

Третий способ разметки кода предлагается в работе [7], суть его состоит в том, что в разметке для каждого элемента, к которому осуществлена привязка, сохраняются *контексты* — вспомогательная информация, собираемая по синтаксическому дереву программы. В случае, если запрошен переход к помеченному элементу, этот элемент ищется в актуальной версии кода среди элементов того же типа при помощи специального алгоритма, использующего контексты для оценки похожести. Данный подход, с нашей точки зрения, является наиболее перспективным: реализующий его инструмент не предъявляет специфических требований к среде разработки, в которую он интегрируется, границы устойчивости разметки интуитивно понятны: чем сильнее меняется помеченный участок и его окружение, тем меньше уверенность алгоритма в том, что это тот же самый код, к которому привязались ранее.

3. Привязка к коду: сохраняемая информация

Модели и алгоритмы, лежащие в основе инструмента разметки, описываемого в настоящей работе, базируются на идее контекстов и являются существенно модифицированными по сравнению с [7]. Для осуществления легковесного парсинга программ и построения абстрактных синтаксических деревьев (АСД) используется разработанный нами генератор легковесных парсеров, входящий в программный комплекс LanD [8, 9]. По построенному АСД для каждой помечаемой сущности определяется тип («класс», «метод», «поле» и т. п.) и вычисляются три контекста: *контекст заголовка* C^H, *контекст предков* C^A, *внутренний контекст* C^I.

Контекст заголовка конструируется на основе листовых непосредственных потомков узла АСД, соответствующего помечаемой сущности. В отличие от [7],

где заголовок рассматривается как последовательность строк, в нашей модели заголовок является последовательностью *слотов*. Слот задаётся тройкой *<тип, приоритет, режим сравнения>* и содержит некоторый текст. На рис. 1a приведён фрагмент грамматики языка C# на языке LanD. Правило для нетерминального символа `method` описывает метод класса. Заголовок метода, приведённого на рис. 1b, порождает 5 показанных на этом же рисунке слотов; для каждого слота справа от стрелки указан попадающий в него текст. Тип слота совпадает с соответствующим символом грамматики, приоритет и режим сравнения задаются в грамматике в конфигурационной секции `%mapping`; опция `leaf` в секции `%parsing` задаёт нетерминалы, узлы которых должны, наряду с узлами токенов, быть листовыми. Типизированность слотов позволяет уйти от допустимых в [7] заведомо некорректных межтиповых сравнений похожести. Приоритеты позволяют зафиксировать понимание того, какие элементы заголовка наиболее специфичны для сущности. Режим сравнения определяет, как оценивается похожесть содержимого: путём вычисления расстояния Левенштейна (редакционного расстояния) [10] или проверкой строгого совпадения. Первый вариант (*Similarity*) применяется по умолчанию, использование второго (*ExactMatch*) задаётся опцией.

```
method : MODIFIER* type name arguments block
block  : '{' Any '}'
...
%parsing {
  leaf type name arguments
}
%mapping {
  exactmatch MODIFIER
  priority(0.1) MODIFIER
  priority(0.3) type arguments
  priority(0.8) name
  land namespace class method field property
}
```

(a)

```
public static int TestMethod(...) {...}

<MODIFIER,  0.1, ExactMatch> ← public
<MODIFIER,  0.1, ExactMatch> ← static
<type,      0.3, Similarity> ← int
<name,      0.8, Similarity> ← TestMethod
<arguments, 0.3, Similarity> ← (...)
```

(b)

Рис. 1. Пример разбиения заголовка на слоты: a) фрагмент грамматики в формате LanD, описывающий метод класса и параметры слотов; b) слоты для заголовка метода «TestMethod».

Внутренний контекст в исследовании [7] представляет собой множество, в которое попадает информация о типах и заголовках фиксированного количества как непосредственных, так и опосредованных потомков узла, соответствующего помечаемой сущности. С увеличением количества запоминаемых потомков увеличивается охват содержимого, однако сравнение таких контекстов становится всё более тяжеловесным. Кроме того, некоторые потомки, например, тело метода, не имеют заголовка и совсем выпадают из рассмотрения. Наша реализация является легковесной и при этом обеспечивает полный охват. Внутренний контекст хранит либо конкатенированный текст всех участков программы, соответствующих нелистовым непосредственным потомкам, либо

его нечёткий хеш (fuzzy hash). Для осуществления хеширования был выбран алгоритм TLSH [11]: в отличие от других известных алгоритмов, он позволяет получать содержательные хеши для текстов небольшой длины (начиная с 25 символов в двухбайтовой кодировке).

Контекст предков образован последовательностью пар <*тип, контекст заголовка*>, в нём сохраняются типы и заголовки тех сущностей, объемлющих помечаемую, которые отмечены в грамматике специальной опцией `land`. Пометка сущностей данной опцией означает, что они являются важными в рамках задачи привязки: привязку планируется осуществлять к ним и к их содержимому. Следовательно, правомерно ожидать, что это значимые структурные элементы программы и их учёт как предков помечаемого элемента поможет при его поисках в изменившемся коде.

4. Поиск функциональности в изменённом коде

С точки зрения инструмента разметки, существует несколько моделей программы: исходный текст, АСД и разметка кода на функциональности. При изменении текста остальные модели должны быть синхронизированы с ним и друг с другом. АСД может быть перестроено с нуля; для каждого элемента разметки необходимо осуществить *перепривязку*, сравнив связанные с ним контексты, вычисленные по старому дереву, с новыми, вычисленными для представленных в дереве элементов того же типа, и выбрав наилучшее соответствие. Большая по сравнению с [7] содержательность контекстов позволяет сформулировать более точные оценки похожести.

Пусть даны a и b — два элемента одного типа; a — исходный помеченный элемент программы, b — кандидат для перепривязки. Элементы контекстов заголовка C^{H_a} и C^{H_b} будем обозначать как e^{H_a} и e^{H_b}. Похожесть контекстов $\mathrm{Sim}(C^{H_a}, C^{H_b})$ определяется как $1 - \mathrm{Dist}(C^{H_a}, C^{H_b})$, где $\mathrm{Dist}(C^{H_a}, C^{H_b})$ — редакционное расстояние, вычисляемое для последовательностей слотов при помощи модифицированного алгоритма Вагнера-Фишера [12] и затем нормируемое (приводимое к диапазону от 0 до 1). В модифицированном алгоритме веса операций удаления и добавления полагаются равными приоритетам слотов, стоимость замены e^{H_a} на e^{H_b} равна произведению приоритета на разность $1 - \mathrm{Sim}(e^{H_a}, e^{H_b})$, где

$$\mathrm{Sim}(e^{H_a}, e^{H_b}) = \begin{cases} -\infty, & \mathrm{Slot}(e^{H_a}) \neq \mathrm{Slot}(e^{H_b}) \\ 1, & \mathrm{IsExactMatch}(e^{H_a}), \mathrm{Text}(e^{H_a}) = \mathrm{Text}(e^{H_b}) \\ 0, & \mathrm{IsExactMatch}(e^{H_a}), \mathrm{Text}(e^{H_a}) \neq \mathrm{Text}(e^{H_b}) \\ 1 - \mathrm{Dist}\big(\mathrm{Text}(e^{H_a}), \mathrm{Text}(e^{H_b})\big), & \text{иначе.} \end{cases}$$

Функция Slot возвращает параметризующую слот тройку, IsExactMatch истинна, если для слота требуется точное совпадение, а функция Dist возвращает нормированное расстояние Левенштейна для текстов, содержащихся в слотах, при вычислении этого расстояния веса всех операций полагаются равными 1.

Поскольку для внутренних контекстов C^{I_a} и C^{I_b} содержимым является либо текст, либо хеш текста, их похожесть определяется по формуле

$$\mathrm{Sim}(C^{I_a}, C^{I_b}) = \begin{cases} \theta'\left(1 - \mathrm{Dist}\left(\mathrm{Text}(C^{I_a}), \mathrm{Text}(C^{I_b})\right), \frac{L_1}{L_2}\right), \forall k \in \{a, b\}, \mathrm{Len}\left(\mathrm{Text}(C^{I_k})\right) \leq L_2 \\ \theta'\left(1 - \mathrm{TlshDist}\left(\mathrm{Hash}(C^{I_a}), \mathrm{Hash}(C^{I_b})\right), \frac{L_1}{L_2}\right), \forall k \in \{a, b\}, \mathrm{Len}\left(\mathrm{Text}(C^{I_k})\right) \geq L_1 \\ 0, \quad \text{иначе.} \end{cases}$$

Здесь θ' — пороговая функция, возвращающая первый аргумент, если его значение больше или равно значению второго; TlshDist — функция, возвращающая оценку похожести хешей, полученную от библиотеки TLSH и затем нормированную; L_1 — минимальная длина текста, для которого вычисляется нечёткий хеш; L_2 — максимальная длина текста, непосредственно сохраняемого в контексте. На промежутке от L_1 до L_2 контекст хранит и текст, и его хеш. Длина текста запоминается, даже если сам текст не сохранён. Отметим, что при выбранном способе хранения контексты C^{I_a} и C^{I_b}, для которых $\mathrm{Len}\left(\mathrm{Text}(\mathrm{C}^{I_a})\right) > L_2$, $\mathrm{Len}\left(\mathrm{Text}(\mathrm{C}^{I_b})\right) < L_1$, являются несравнимыми: первый содержит только хеш, второй — только текст. Очевидно, что максимально возможная похожесть таких контекстов не превышает максимально возможного соотношения длин текстов, равного L_1/L_2. Выбрав минимально возможное значение L_1 и достаточно большое значение L_2, мы можем пренебречь всеми похожестями меньше данного соотношения. В текущей реализации $L_1 = 25$, $L_2 = 100$.

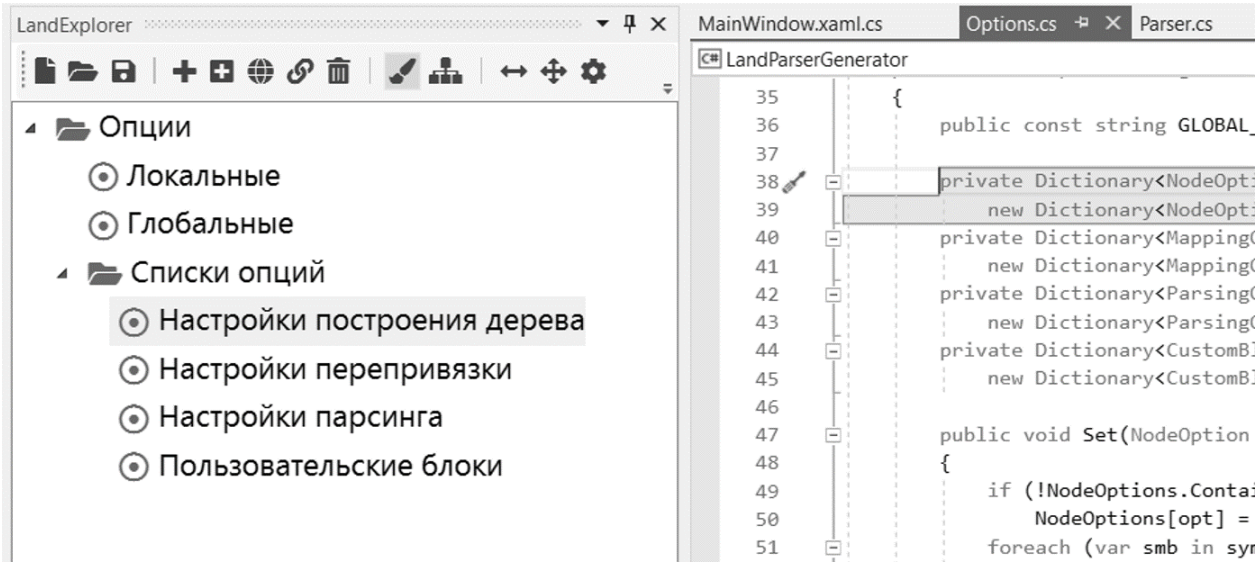

Рис. 2. Фрагмент окна среды VisualStudio с интегрированной панелью разметки.

При сравнении контекстов предков используется стандартная версия алгоритма [12], похожесть двух элементов контекста равна 0 при несовпадении типа, в противном случае она равна похожести их заголовков.

Итоговую оценку похожести a и b в работе [7] предлагается вычислять как скалярное произведение вектора похожестей контекстов на вектор весовых коэффициентов, что эквивалентно линейной комбинации вида

$$\mathrm{Sim}(a,b) = \frac{w^H \cdot \mathrm{Sim}(C^{Ha}, C^{Hb}) + w^A \cdot \mathrm{Sim}(C^{Aa}, C^{Ab}) + w^I \cdot \mathrm{Sim}(C^{Ia}, C^{Ib})}{w^H + w^A + w^I}.$$

Здесь w^H, w^A, w^I — эмпирически подбираемые весовые коэффициенты, отражающие важность каждого из контекстов. В первом приближении предполагается, что имя сущности важнее предков, а предки важнее содержимого. Между тем, как показано в разделе 5, важность контекста для верной перепривязки может варьироваться в зависимости от конкретного набора кандидатов, в связи с чем подбор универсальных «хороших» коэффициентов не представляется выполнимой задачей. Нами применяется комбинированный подход, предполагающий использование весов совместно с эвристическими алгоритмами. До непосредственного вычисления итоговых оценок запускается серия проверок, выявляющих особенности оцениваемых кандидатов. По результатам этих проверок веса корректируются, а для некоторых кандидатов оценка предлагается сразу.

После вычисления итоговых оценок все кандидаты ранжируются по убыванию похожести на a и помещаются в список $Candidates$. Решение о перепривязке к первому элементу списка принимается автоматически, если выполняется условие $\mathrm{Sim}(a, Candidates[0]) \geq 0.6 \wedge \Big(\mathrm{Len}(Candidates) = 1 \vee \big(1 - \mathrm{Sim}(a, Candidates[0])\big) \cdot 1.5 \leq \big(1 - \mathrm{Sim}(a, Candidates[1])\big)\Big)$, где 0.6 — эмпирически подобранная минимальная рассматриваемая похожесть, а 1.5 — коэффициент, означающий, что отличие исходного элемента от второго кандидата должно быть не менее чем в полтора раза больше отличия от первого. Если условие не выполняется, пользователю предоставляется список вероятных кандидатов, в котором подходящий вариант нужно выбрать вручную.

Реализующий описанные модели и алгоритмы инструмент был интегрирован в комплекс LanD и получил название LandExplorer, он также был оформлен в виде расширения для среды разработки VisualStudio (рис. 2).

5. Примеры работы

В рамках эксперимента было смоделировано несколько ситуаций, в которых устойчивая привязка к коду невозможна при использовании не модифицированных нами моделей и алгоритмов. На рис. 3 и 4 в левой части расположен исходный размечаемый код, рамкой очерчены сущности, к которым производится привязка. В правой части помещён отредактированный код, в

рамку взяты вероятные кандидаты для перепривязки, изменённые области выделены серым. В таблице 1 для обоих примеров приведены оценки похожести для различных пар сущностей, получаемые при перепривязке с использованием контекстов, описанных в [7], (столбец «Базовый алгоритм») и с использованием описанной в настоящей работе модификации (столбец «Модиф. алгоритм»). Для базовой версии $w^H = 3$, $w^A = 2$, $w^I = 1$; в столбце «Модиф. алгоритм» первая оценка приведена для тех же весов, вторая — для скорректированных эвристиками. Под суммарной оценкой в квадратных скобках приведены оценки похожести контекстов C^H, C^A и C^I соответственно.

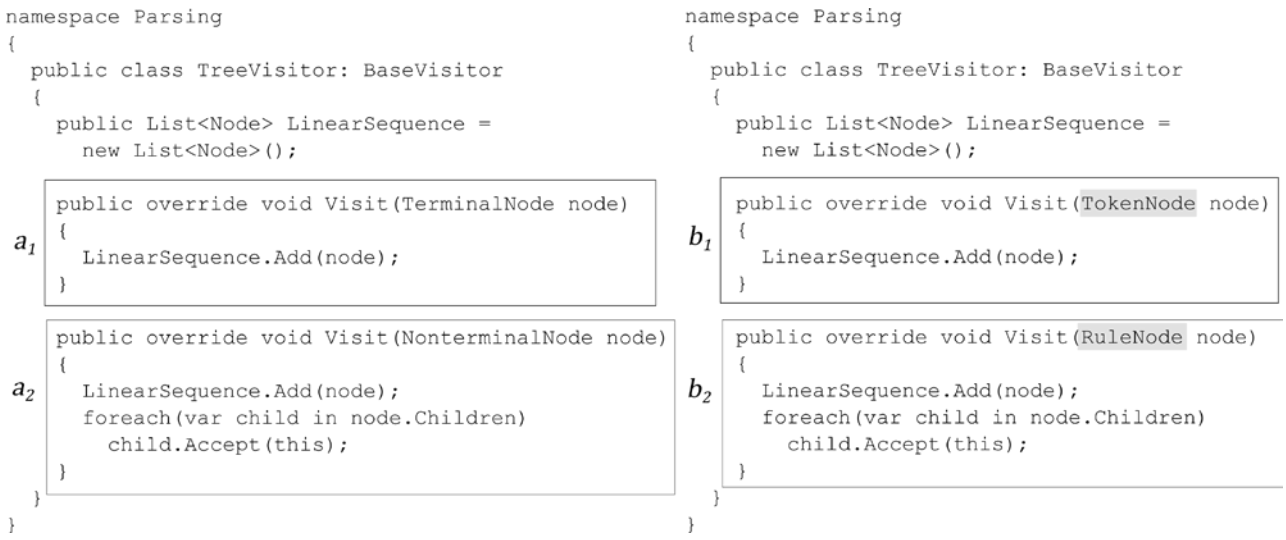

Рис. 3. Пример изменения программы, при котором необходим учёт внутреннего контекста.

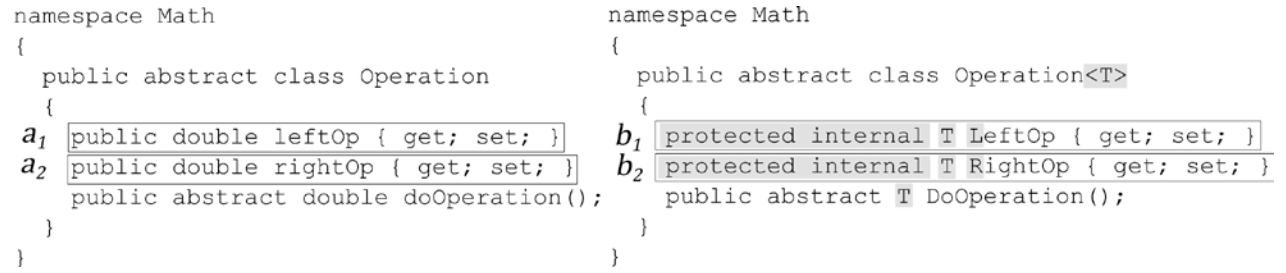

Рис. 4. Пример изменения программы, при котором важен учёт типов и весов элементов заголовка.

На рис. 3 представлен классический пример класса, содержащего набор методов с почти одинаковой сигнатурой — класс визитора по синтаксическому дереву. В данном случае решающим при поиске помеченного элемента в изменённом коде является контекст C^I, который, как уже было отмечено, не учитывается для методов в [7]. Как следствие, оценки похожестей для всех возможных пар в столбце «Базовый алгоритм» близки к 1 и не позволяют автоматически принять решение. Более того, $\mathrm{Sim}(a_2, b_1) > \mathrm{Sim}(a_2, b_2)$, поэтому в списке, выдаваемом пользователю при перепривязке a_2, правильное

соответствие оказывается не на первом месте. Модифицированный алгоритм способен сделать правильный выбор без участия пользователя. Поскольку внутренние контексты кандидатов различаются существеннее всего, эвристика перераспределяет веса: $w^H = 2,\ w^A = 1,\ w^I = 3$. В итоге расстояние между кандидатами увеличивается ещё сильнее.

	Рис. 3		Рис. 4	
	Базовый алгоритм	Модиф. алгоритм	Базовый алгоритм	Модиф. алгоритм
$\mathrm{Sim}(a_1, b_1)$	0.97 [0.93, 1, 1]	0.97 / 0.98 [0.94, 1, 1]	0.62 [0.26, 0.97, 1]	0.73 / 0.69 [0.51, 0.93, 1]
$\mathrm{Sim}(a_1, b_2)$	0.96 [0.91, 1, 1]	0.85 / 0.64 [0.92, 1, 0.34]	0.57 [0.16, 0.97, 1]	0.61 / 0.54 [0.26, 0.93, 1]
$\mathrm{Sim}(a_2, b_1)$	0.96 [0.92, 1, 1]	0.86 / 0.65 [0.93, 1, 0.34]	0.57 [0.16, 0.97, 1]	0.61 / 0.54 [0.26, 0.93, 1]
$\mathrm{Sim}(a_2, b_2)$	0.95 [0.9, 1, 1]	0.96 / 0.97 [0.92, 1, 1]	0.62 [0.26, 0.97, 1]	0.74 / 0.7 [0.53, 0.93, 1]

Таблица 1. Оценки похожестей.

На рис. 4 в заголовках помеченных свойств полностью изменилось всё, кроме имени, при этом именно контекст заголовка является решающим для верной перепривязки: остальные контексты оценены одинаково высоко у обоих кандидатов. В отсутствие типизации и приоритетов все изменившиеся элементы заголовка одинаково важны, в результате оценки, выдаваемые базовым алгоритмом, отстоят друг от друга недостаточно для принятия автоматического решения. В соответствии с рис. 1, для модифицированного алгоритма изменение модификаторов почти не имеет значения, а основное внимание сосредотачивается на именах. Как следствие, увеличивается оценка правильного варианта и уменьшается оценка неподходящего. Эвристически веса корректируются следующим образом: $w^H = 3,\ w^A = 1,\ w^I = 1$; в результате решение о перепривязке снова принимается без участия пользователя.

6. Заключение

В настоящей работе описывается реализация инструмента LandExplorer, предназначенного для разметки сквозных функциональностей в программном коде. Поиск помеченных участков кода осуществляется алгоритмически на основе сохранённой в момент пометки информации — набора вычисляемых по абстрактному синтаксическому дереву контекстов. Модели контекстов и алгоритмы поиска существенно отличаются от ближайшего аналога: элементы контекста заголовка являются типизированными, взвешенными, а также поддерживают режимы сравнения; внутренний контекст охватывает всё содержимое сущности благодаря применению нечёткого хеширования; контекст

предков учитывает только важные объемлющие элементы; алгоритм перепривязки адаптирован к улучшенным контекстам и позволяет получить более точные оценки похожести. В результате расширяется множество устойчиво размечаемых программ.

В качестве основного направления дальнейшего исследования рассматривается реализация на базе инструмента LandExplorer концепции наборных и вариативных гнёзд, предложенной в работах [3–5]. Также существуют задачи, связанные с улучшением инструмента разметки: поиск дополнительных эвристик, возможно, затрагивающих семантику программы, и оптимизация объема сохраняемой в разметке информации.

Литература

1. Kaindl H. What is an Aspect in Aspect-oriented Requirements Engineering // Proceedings of the 13th International Workshop on Exploring Modeling Methods for Systems Analysis and Design, Montpellier, France, June 16–17, 2008. P. 164–170.
2. Фуксман А.Л. Технологические аспекты создания программных систем. М: Статистика, 1979. 184 с.
3. Горбунов-Посадов М.М. Безболезненное развитие программы // Открытые системы. 1996. № 4. С. 65–70.
4. Горбунов-Посадов М.М. Расширяемые программы. М.: Полиптих, 1999. 336 с.
5. Горбунов-Посадов М.М. Как растёт программа. // Открытые системы. 2000. № 10. С. 43–47.
6. 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. Malevannyy M., Mikhalkovich S., Robust binding to syntactic elements in a changing code // Proceedings of the 12th Central & Eastern European Software Engineering Conference in Russia. CEE-SECR ’16. New York, NY, USA: ACM, 2016. P. 13:1–13:8.
8. 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(4). P. 7–28.
9. 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(3). P. 7–28.
10. Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов. // Доклады Академий Наук СССР. 1965. Т. 163, № 4. С. 845-848.
11. Oliver J., Cheng C., Chen Y. TLSH — A Locality Sensitive Hash // Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop. CTC ’13. Washington, DC, USA: IEEE Computer Society, 2013. P. 7–13.

12. Wagner R.A., Fischer M.J. The string-to-string correction problem. // Journal of the ACM. 1974. Vol. 21(1). P. 168–173.

References

1. Kaindl H. What is an Aspect in Aspect-oriented Requirements Engineering // Proceedings of the 13th International Workshop on Exploring Modeling Methods for Systems Analysis and Design, Montpellier, France, June 16–17, 2008. P. 164–170.
2. Fuksman A.L. Tekhnologicheskie aspekty sozdaniya programmnykh sistem. M: Statistika, 1979. P. 184.
3. Gorbunov-Posadov M.M. Bezboleznennoe razvitie programmy // Otkrytye sistemy. 1996. № 4. P. 65–70.
4. Gorbunov-Posadov M.M. Rasshiryaemye programmy. M.: Poliptikh, 1999. P. 336.
5. Gorbunov-Posadov M.M. Kak rastet programma. // Otkrytye sistemy. 2000. № 10. P. 43–47.
6. Kalinin S.Yu., Kolokolov I.A., Litvinenko A.N. Primenenie kontseptsii AOP v razrabotke rasshiryaemykh prilozhenii // Izvestiya Yuzhnogo federal'nogo universiteta. Tekhnicheskie nauki. 2010. Vol. 103(2). P. 58–68.
7. Malevannyy M., Mikhalkovich S., Robust binding to syntactic elements in a changing code // Proceedings of the 12th Central & Eastern European Software Engineering Conference in Russia. CEE-SECR ’16. New York, NY, USA: ACM, 2016. P. 13:1–13:8.
8. 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(4). P. 7–28.
9. 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(3). P. 7–28.
10. Levenshtein V.I. Dvoichnye kody s ispravleniem vypadenii, vstavok i zameshchenii simvolov. // Doklady Akademii Nauk SSSR. 1965. Vol. 163(4). P. 845-848.
11. Oliver J., Cheng C., Chen Y. TLSH — A Locality Sensitive Hash // Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop. CTC ’13. Washington, DC, USA: IEEE Computer Society, 2013. P. 7–13.
12. Wagner R.A., Fischer M.J. The string-to-string correction problem. // Journal of the ACM. 1974. Vol. 21(1). P. 168–173.