: **ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ** :

УДК 004.42

ОБНАРУЖЕНИЕ НЕТОЧНЫХ ПОВТОРОВ В ДОКУМЕНТАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ *

© 2018 г. Д.В. Луцив, Д.В. Кознов, Г.А. Чернышев, А.Н. Терехов, К.Ю. Романовский, Д.А. Григорьев

Санкт-Петербургский государственный университет

199034 Санкт-Петербург, Университетская набережная, д. 7–9

E-mails: d.lutsiv@spbu.ru, d.koznov@spbu.ru, g.chernyshev@spbu.ru, a.terekhov@spbu.ru, k.romanovsky@spbu.ru, d.a.grigoriev@spbu.ru

Документация современного программного обеспечения столь же сложна, как и само программное обеспечение. Велики объёмы документации, имеется значительное количество сложных связей документации внутри себя, а также с другими артефактами разработки, непрост и трудоёмок процесс сопровождения документации. В течение жизненного цикла в документах образуется множество неточных повторов, т.е. фрагментов текста, скопированных из одного источника и впоследствии по-разному модифицированных. Такие дубликаты снижают качество документации и затрудняют её дальнейшее использование. В то же время вручную неточные повторы трудно обнаружить. В статье даётся формальное определение неточных повторов и предлагается алгоритм их обнаружения. Алгоритм основан на поиске клонов в программном обеспечении. Представлено формальное обоснование корректности алгоритма. Описаны результаты апробации алгоритма на документации 19 открытых и коммерческих программных проектов. Выполнен анализ обнаруженных неточных повторов для документации ядра ОС Linux.

Ключевые слова: *документация программного обеспечения, неточные повторы, повторное использование документации, клоны в программном обеспечении.*

DOI: 10.31857/S013234740001215-1

1. ВВЕДЕНИЕ

Программное обеспечение непрерывно проникает в новые сферы жизнедеятельности человека и все больше усложняется. Соответственно, растёт и усложняется его документация. Широко используемый при разработке документации приём copy/paste приводит к накапливанию в документации многочисленных неточных повторов: вначале фрагмент текста копируется, затем каждая копия как-то изменяется. В зависимости от типа и назначения документа [1] повторы могут быть желательны или нет, но в любом случае они усложняют документацию и увеличивают расходы на её поддержку [2].

Проблема повторов в документации программного обеспечения активно изучается [2, 3, 4, 5]. Тем не менее, методов для обнаружения неточных повторов на настоящий момент не представлено. Кроме того, нет формального определения неточных повторов, хотя признано, что данный феномен существует и имеет большую практическую значимость [2, 3, 6]. В наших предыдущих работах [7, 8] была представлена первая версия подхода к обнаружению неточных повторов в документации программного обеспечения. Использовался инструмент Clone Miner [9], предназначенный для поиска клонов в программном обеспечении. С его помощью находились точные повторы, а неточные конструировались как комбинации точных. При этом были рассмотрены лишь неточные повто-

*Работа частично поддержана РФФИ, грант № 16-01-00304.

ры с одним изменяемым фрагментом. Иными словами, описанный подход позволял обнаруживать неточные повторы, состоящие из двух точных и единственного варьируемого фрагмента текста между ними: $exact_1$ $variable_1$ $exact_2$. В данных работах не было также представлено формального определения неточных повторов.

В этой работе мы предлагаем формальное определение неточных повторов с произвольным количеством изменяемых фрагментов: $exact_1$ $variable_1$ $exact_2$ $variable_2$ $exact_3 \ldots variable_{n-1}$ $exact_n$. Предлагаемое определение является формализованным вариантом определения, предложенного в [10]. Также мы предлагаем обобщение алгоритма, описанного в [7, 8]. Предложенный в данной работе алгоритм реализован в инструменте Duplicate Finder [11] и входит в проект DocLine [4]. Также в данной работе представлены результаты апробации предложенного алгоритма на документации 19 коммерческих проектов и проектов с открытым исходным кодом. Наконец, для документации ядра ОС Linux [12] представлены результаты "ручного" анализа найденных неточных повторов.

2. ОБЗОР

Рассмотрим, каким образом в программной инженерии производится поиск и анализ повторов в документации ПО. M. Horie и др. [13] рассматривают повторяющиеся фрагменты текста в документации Java API, используя для обозначения повторов термин "crosscutting concern". Они представляют инструмент CommentWeaver, который расширяет JavaDoc средствами повторного использования. Но неточные повторы в работе не рассматриваются. M. Nosál и J. Porubän [14] расширяют предложенный в [13] подход, рассматривая неточные повторы, называя повтор "documentation phrase". Подобно нашему подходу [4, 5], для определения варьирующихся частей повторов используется параметризация. Но в работе не затрагивается задача обнаружения неточных повторов, а сами неточные повторы определены неформально.

В [3] представлен результат исследования точных повторов во встроенной документации (комментариях в исходном коде) для ряда проектов с открытым исходным кодом. Авторы использовали инструмент для обнаружения "раскопированных" фрагментов (copy-pastes), который исходно был предназначен для анализа кода, и нашли с его помощью значительное количество повторов. Однако неточные повторы в этой работе не рассматривались.

A. Wingkvist и др. использовали инструмент поиска клонов в программном обеспечении для оценки уникальности документов, входящих в программные проекты [6]. Полученную информацию о повторах авторы использовали для оценки качества документации. Неточные повторы в данной работе не рассматривались.

Работа E. Juergens и др. [2] посвящена исследованию избыточности в описаниях требований. С помощью инструмента поиска клонов авторы проанализировали 28 промышленных документов. Найденные повторы были отфильтрованы и классифицированы вручную. Авторы сообщают, что величина среднего *покрытия повторами* документов составила 13,6%. При этом некоторые документы имеют низкий охват (0,9%, 0,7% и даже 0%), на некоторых покрытие достигает 35%, 51,1%, 71,6%. Покрытие документа повторами (reuse amount) означает отношение объёма повторов к объёму всего документа. В статье также обсуждается, что означают найденные повторы, в частности показывается, что некоторые из них соответствуют дублированию в исходном коде систем. Кроме того, изучается влияние повторов на процесс чтения документов экспертами. Также авторы предлагают классификацию как *значимых повторов*, так и *ложноположительных срабатываний* (false positives). В работе не рассмотрены другие виды документации программного обеспечения, а также неточные повторы.

M. Oumaziz с коллегами [15] анализирует API-документацию нескольких известных программных проектов, созданную с применением технологии Javadoc. Поиск повторов производится на основе детектора клонов. Авторы приводят классификацию повторов и исследуют возможность повторного использования документации. Неточные повторы в работе не рассматриваются, хотя и указывается на то, что такие повторы часто встречаются и важны на практике.

A. Rago и др. [16] применяют методы анализа естественных языков для поиска неточных повторов в текстовых описаниях вариантов исполь-

зования (use cases). Следует, однако, заметить, что их работа посвящена весьма специфичному способу описания требований, который в индустрии используется нечасто; неочевидно, как использовать данный метод для других видов документации ПО.

Алгоритмы обнаружения повторов в текстах разрабатываются также в других областях. Сообщество информационного поиска рассматривает ряд задач, связанных с поиском неточных повторов: поиск похожих документов в Интернет [17, 18], обнаружение (локального) повторного использования [19, 20], выделение текстовых шаблонов в коллекциях web-страниц [21, 22]. Сообщество обнаружения плагиата также подробно изучает повторы, и, в частности, неточные повторы в документах [23, 24, 25]. Однако эти подходы ориентированы на обнаружение сходства на уровне целых документов и на достижение высокой производительности на больших коллекциях.

Традиционно, при поиске повторов в документации ПО используются средства обнаружения клонов в программном коде [2, 3, 6, 15] Отметим, что в области поиска программных клонов имеются средства поиска неточных повторов. Инструмент SourcererCC [26] обнаруживает повторяющиеся фрагменты кода, используя стратегию *static bag-of-tokens*, которая устойчива к незначительным различиям этих фрагментов. Инструмент DECKARD [27] вычисляет характеристические векторы текстов программ для аппроксимации структуры абстрактных синтаксических деревьев в евклидовом пространстве. NICAD [28] является средством обнаружения неточных текстовых повторов для функций настраиваемой нормализации кода (pretty-printing) и преобразования/фильтрации исходного кода. Можно также упомянуть другие аналогичные работы [29, 30]. Следует отметить, что подобные методы не подходят для обнаружения дубликатов в текстовых документах, поскольку они применяют синтаксический анализ исходного кода (применительно к документации они используются лишь для поиска точных повторов, и то при условии, что включают в себя поиск на основе токенов, а не только поиск по синтаксическому дереву программы). Тем не менее, является привлекательным использование поиска клонов как базового средства обнаружения повторов в виду доступности соответствующих инструментов, возможности надстроить сверху поиск неточных повторов, а также из-за хорошей корреляции такого синтетического подхода с концепцией архетипа/дельты [10], используемой в нашем исследовании для формализации неточных повторов. Следуя [16, 31] отметим, что для нашей задачи перспективными являются средства анализа текстов на естественных языках (natural language processing). Следующие приёмы могут быть полезны: N-gram модель [31], тематическое моделирование текстов (позволяет отнести фрагменты текстов к той или иной области [32]), техники выделения фактов из текстов в свободной форме [33]; техники нормализации текста. Использование этих методов будет являться предметом наших дальнейших исследований.

3. ОБНАРУЖЕНИЕ ТОЧНЫХ ПОВТОРОВ И CLONE MINER

Не только документация, но и само программное обеспечение зачастую разрабатывается путём копирования исходного кода. Для того, чтобы справляться с повторами в исходном коде, применяются методы обнаружения клонов в исходных кодах [34]. В данной работе для обнаружения точных повторов использован инструмент поиска клонов в исходном коде Clone Miner [9]. Clone Miner — инструмент поиска клонов, основанный на токенах. Под токеном понимается слово, отделённое от соседних некоторым разделителем: “.”, “,”, “ ”, “(”, “)” и т.д. Например, фрагмент текста “FM registers” состоит из двух токенов. Clone Miner рассматривает текст, как упорядоченную коллекцию токенов и находит повторяющиеся фрагменты (клоны) при помощи основанных на суффиксных деревьях алгоритмов [35]. Выбор Clone Miner был обусловлен тем, что он не использует для поиска клонов синтаксического анализа, а также его простотой и возможностью интеграции с другими инструментами при помощи интерфейса командной строки.

4. ОПРЕДЕЛЕНИЕ НЕТОЧНЫХ ПОВТОРОВ

Задача данного раздела — формулировка формальное определение неточного повтора. Рассмотрим документ D как последовательность

символов. Любой символ документа D имеет координату, соответствующую смещению символа от начала документа, и эта координата является числом из интервала $[1, length(D)]$, где $length(D)$ — это количество символов в D.

Определение 1. *Определим* ***текстовый фрагмент*** *документа как вхождение в документ некоторой подстроки символов. Таким образом, каждому текстовому фрагменту документа D соответствует целочисленный интервал $[b, e]$, где b — это координата первого символа фрагмента, e — координата последнего. Для текстового фрагмента g документа D будем писать $g \in D$.*

Введём следующие множества: D^* — множество всех текстовых фрагментов документа D, I^D — множество всех целочисленных интервалов, входящих в интервал $[1, length(D)]$, S^D — множество всех возможных символьных строк длины не больше $length(D)$. Введём следующие обозначения:

- $[g] : D^* \to I^D$ — функция, которая по текстовому фрагменту g выдаёт его интервал;
- $str(g) : D^* \to S^D$ — функция, которая по текстовому фрагменту g выдает его текст;
- $\overline{I} : I^D \to D^*$ — функция, которая по интервалу I выдаёт соответствующий ему текстовый фрагмент;
- $|[b, e]| : I^D \to [0, length(D)]$ — функция, которая по интервалу $[g] = [b, e]$ выдает его длину как $|[g]| = e - b + 1$. Для сокращения записи вместо $|[g]|$ мы будем писать $|g|$;
- Для любых $g^1, g^2 \in D$ их пересечение $g^1 \cap g^2$ будем рассматривать как пересечение соответствующих интервалов $[g^1] \cap [g^2]$;
- Определим двухместный предикат $Before$ на множестве $D^* \times D^*$, являющийся истинным для двух текстовых фрагментов g^1, g^2 документа D, тогда и только тогда, когда $e^1 < b^2$, если $[g^1] = [b^1, e^1], [g^2] = [b^2, e^2]$.

Определение 2. *Пусть G — набор текстовых фрагментов документа D таких, что $\forall g^1, g^2 \in G$ $((str(g^1) = str(g^2)) \wedge (g^1 \cap g^2 = \emptyset))$. Такие фрагменты назовём* ***точными повторами****, а G —* ***группой точных повторов*** *или* ***точной группой****. Посредством $\#G$ обозначим количество элементов в G.*

Определение 3. *Для набора групп точных повторов $G_1, \ldots, G_N$ будем говорить, что этот набор образует* ***вариативную группу*** *$\langle G_1, \ldots G_N \rangle$, если выполняются следующие условия:*

1. $\#G_1 = \ldots = \#G_N$.
2. *Текстовые фрагменты, имеющие в разных группах одни и те же порядковые номера, следуют в исходном тексте в одном и том же порядке: $\forall g_i^k \in G_i \ \forall g_j^k \in G_j \ ((i < j) \Leftrightarrow \Leftrightarrow Before(g_i^k, g_j^k))$, и $\forall\, k \in \{1, \ldots, N-1\} Before(g_N^k, g_1^{k+1})$.*

При этом будем писать, что для любой группы G_k из этого набора справедливо $G_k \in VG$.

Замечание 1. *Из условия 2 определения 3 следует, что $\forall g_i^k \in G_i, \forall g_j^k \in G_j (i \neq j \Rightarrow g_i^k \cap g_j^k = \emptyset)$.*

Замечание 2. *Если $VG = \langle G_1, \ldots G_N \rangle$ и $VG' = \langle G'_1, \ldots G'_M \rangle$ — вариативные группы, то $\langle VG, VG' \rangle = \langle G_1, \ldots G_N, G'_1, \ldots G'_M \rangle$ — тоже вариативная группа, если она удовлетворяет определению 3.*

Определение 4. ***Расстояние между текстовыми фрагментами.*** *Для любых $g^1, g^2 \in \in D$ определим расстояние следующим образом:*

$$dist(g^1, g^2) = \begin{cases} 0, & g^1 \cap g^2 \neq \emptyset, \\ b^2 - e^1 + 1, & Before(g^1, g^2), \\ b^1 - e^2 + 1, & Before(g^2, g^1), \end{cases} \quad (1)$$

где $[g^1] = [b^1, e^1]$ и $[g^2] = [b^2, e^2]$.

Определение 5. ***Расстояние между точными группами*** *G_1 и G_2 при условии, что $\#G_1 = \#G_2$ определим так:*

$$dist(G_1, G_2) = \max_{k \in \{1, \ldots, \#G_1\}} dist(g_1^k, g_2^k). \quad (2)$$

Определение 6. ***Расстояние между вариативными группами*** *VG_1 и VG_2, при условии,*

что существуют $G_1 \in VG_1, G_2 \in VG_2 : \#G_1 = \#G_2$, *определим так:*

$$dist(VG_1, VG_2) = \max_{G_1 \in VG_1, G_2 \in VG_2} dist(G_1, G_2). \quad (3)$$

Определение 7. ***Длина точной группы*** G *определяется следующим образом:* $length(G) = \sum_{k=1}^{\#G} (e^k - b^k + 1)$, *где* $g^k \in G, [g^k] = [b^k, e^k]$.

Определение 8. ***Длина вариативной группы*** $VG = \langle G_1, \ldots, G_N \rangle$ *определяется следующим образом:*

$$length(VG) = \sum_{i=1}^{N} length(G_i). \quad (4)$$

Определение 9. ***Группа неточных повторов*** *— это вариативная группа* $VG = \langle G_1, \ldots, G_N \rangle$ *такая, что* $\forall k \in \{1, \ldots, \#G_1\}$ *справедливо:*

$$\sum_{i=1}^{N-1} dist(g_i^k, g_{i+1}^k) \leq 0,15 * \sum_{i=1}^{N} |g_i^k|. \quad (5)$$

Данное определение является формализацией понятия неточного повтора в [36], где утверждается, что вариативная часть неточных повторов одной и той же информации (*delta*) не должна превышать 15% их неизменной части (*архетипа, archetype*).

Замечание 3. *Точная группа может быть рассмотрена как группа неточных повторов, сформированных единственной группой* $\langle G \rangle$.

Определение 10. *Рассмотрим группу неточных повторов* $VG = \langle G_1, G_2 \rangle$, *где* G_1 *и* G_2 *— точные группы. Будем считать, что* VG *содержит единственную* ***точку расширения****, а фрагменты текста, расположенные в интервалах* $[e_1^k + 1, b_2^k - 1]$, *назовём* ***значениями точки расширения****. В общем случае группа неточных повторов* $\langle G_1, \ldots, G_N \rangle$ *имеет* $N - 1$ точек расширения.

Определение 11. *Рассмотрим две группы неточных повторов* $G = \langle G_1, \ldots, G_n \rangle$ *и* $G' = \langle G'_1, \ldots, G'_m \rangle$. *Предположим, что они могут образовать вариативную группу* $\langle G_1, \ldots, G_n, G'_1, \ldots, G'_m \rangle$ *или* $\langle G_1, \ldots, G_n, G'_1, \ldots, G'_m \rangle$, *которая также является группой неточных повторов. В этом случае будем называть* G *и* G' ***соседними группами****.*

Определение 12. ***Соседние повторы*** *— это текстовые фрагменты, принадлежащие соседним группам.*

Замечание 4. *Следуя замечанию 3, определение 12 применимо как к точным, так и к неточным повторам.*

5. АЛГОРИТМ ОБНАРУЖЕНИЯ ГРУПП НЕТОЧНЫХ ПОВТОРОВ

Ниже представлен алгоритм, который ищет в документе D множество групп неточных повторов (множество $SetVG$), получая на вход множество точных групп этого же документа (множество $SetG$). Алгоритм использует интервальное дерево [37, 38] — структуру данных, которая применяется для того, чтобы для некоторого интервала и множества интервалов быстро найти те, которые с ним пересекаются. Множество $SetG$ создаётся с помощью Clone Miner. Основная идея алгоритма — находить и соединять соседние группы из $SetG$ и добавлять получающиеся группы неточных повторов в $SetVG$. Алгоритм представлен ниже.

При помощи функции $Initiate()$ строится исходное интервальное дерево для множества $SetG$ (строка 2). Основная часть алгоритма является циклом, в котором производится выбор новых групп неточных повторов (строки 3–18). Цикл повторяется, пока на очередном шаге множество новых найденных групп неточных повторов $SetNew$ не окажется пустым (строка 18). Внутри этого цикла алгоритм перебирает все группы из $SetG \cup SetVG$, и для каждой из них функция $NearBy$ возвращает множество близкорасположенных к ней групп $SetCand$ (строки 5, 6), которые затем используются для конструирования неточных групп. Ниже эта функция описана детально, также доказывается её корректность, т.е. то, что она действительно возвращает близкие к G группы. Затем из $SetCand$ выбирается ближайшая к G группа G' (строка 8) и создаётся вариативная группа $\langle G, G' \rangle$ или $\langle G', G \rangle$, которая добавляется в $SetNew$ (строки 10–14).

Алгоритм 1: Конструирование групп неточных повторов

Входные данные: $SetG$
Результат: $SetVG$

```
1  SetVG ← ∅
2  Initiate()
3  repeat
4  |  SetNew ← ∅
5  |  foreach G ∈ SetG ∪ SetVG do
6  |  |  SetCand ← NearBy(G)
7  |  |  if SetCand ≠ ∅ then
8  |  |  |  G' ← GetClosest(G, SetCand)
9  |  |  |  Remove(G, G')
10 |  |  |  if Before(G, G') then
11 |  |  |  |  SetNew ← SetNew ∪ {⟨G, G'⟩}
12 |  |  |  else
13 |  |  |  |  SetNew ← SetNew ∪ {⟨G', G⟩}
14 |  |  |  end if
15 |  |  end if
16 |  end foreach
17 |  Join(SetVG, SetNew)
18 until SetNew = ∅
19 SetVG ← SetVG ∪ SetG
```

Поскольку G' и G объединяются, и, таким образом, прекращают самостоятельное существование, они удаляются из $SetG$ и $SetVG$ при помощи функции $Remove$ (строка 9). Затем функция $Join$ добавляет $SetNew$ к $SetVG$ (строка 17). Функции $Remove$ и $Join$ также выполняют некоторые вспомогательные действия, описанные ниже. В самом конце множество $SetVG$ объединяется с множеством $SetG$ и представляется в виде итога работы алгоритма. Данное объединение выполняется для того, чтобы итоговый результат содержал не только группы неточных повторов, но также и те точные группы, которые не были использованы для комбинации неточных повторов (строка 19).

Опишем функции, которые используются в алгоритме.

Функция $Initiate()$ строит интервальное дерево. Опишем эту структуру данных.

Предположим, что у нас есть n интервалов в натуральных числах, и b_1 — минимум, а e_n — максимум концов всех интервалов. m — середина интервала $[b_1, e_n]$. Интервалы разделяются на три группы: интервалы, целиком расположенные слева от m, интервалы, целиком расположенные справа от m, и интервалы, содержащие m. Текущий узел интервального дерева хранит последнюю группу интервалов и ссылки на дочерние узлы, которые хранят интервалы слева и справа от m соответственно. Для каждого дочернего узла процедура построения повторяется. Дальнейшие детали построения интервального дерева можно найти в [37, 38].

Мы строим интервальное дерево из расширенных интервалов, которые соответствуют точным повторам, обнаруженным Clone Miner. Для каждого интервала сохраняется ссылка на группу точных повторов, которой он принадлежит. Расширенные интервалы получаются так. Мы расширяем на 15% интервалы, соответствующие точным повторам: если $[b, e]$ является исходным интервалом, то расширенный интервал выглядит так: $[b-0,15*(e-b+1), e+0,15*(e-b+1)]$. Расширенный интервал, соответствующий точному повтору g, обозначим $\updownarrow g$.

Функция $Remove$ убирает группы из множеств, а все их интервалы — из интервального дерева. Алгоритм удаления из дерева описан в [37, 38].

Функция $Join$, в дополнение к описанным выше действиям, добавляет в интервальное дерево интервалы для вновь созданной группы неточных повторов $G = \langle G_1, \ldots, G_N \rangle$. При этом использован описанный в [37, 38] стандартный алгоритм вставки. Расширенные интервалы, добавляемые в дерево, для каждого неточного повтора $g = (g_1^k, ..., g_N^k)$ выглядят следующим образом: $[b_1^k - x^k, e_N^k + x^k]$, где $x^k = 0,15 * \sum_{i=1}^{N} |g_i^k| - \sum_{i=1}^{N-1} dist(g_i^k, g_{i+1}^k)$. Такой расширенный интервал для g также будем обозначать $\updownarrow g$.

Функция $NearBy$ для некоторой группы G (своего аргумента) выбирает соседние группы. В связи с этим для каждого текстового фрагмента из G ищется набор интервалов из дерева, пересекающихся с его интервалом. *Текстовые фрагменты, соответствующие этим интервалам, оказываются соседними по отношению к данному*, то есть для них выполнено условие (5). Поиск выполняется при помощи алгоритма поиска по интервальному дереву [37, 38]. Строится мно-

жество групп GL_1, являющихся претендентами на то, чтобы быть соседними с G:

$$GL_1 = \{G' | (G' \in SetG \cup SetVG) \wedge \\ \wedge \exists g \in G,\, g' \in G' : \updownarrow g \cap \updownarrow g' \neq \emptyset. \quad (6)$$

То есть множество GL_1 состоит из тех групп, в которых хотя бы один повтор оказался близко к хотя бы одному повтору из G. Затем из этих групп выбираются и помещаются в множество GL_2 только те, которые могут составить вариативную группу с G:

$$GL_2 = \{G' | G' \in GL_1 \wedge (\langle G, G' \rangle \text{или} \\ \langle G', G \rangle \text{— вариативная группа})\}. \quad (7)$$

Наконец, строится множество GL_3 (результат работы функции $NearBy$), куда берутся только те группы из GL_2, в которых все элементы близки к соответствующим элементам G:

$$GL_3 = \{G' | G' \in GL_2 \wedge \\ \wedge \forall k \in \{1, \ldots, \#G\} : \updownarrow g^k \cap \updownarrow g'^k \neq \emptyset\} \quad (8)$$

Теорема 1. *Предложенный алгоритм обнаруживает группы неточных повторов, которые соответствуют определению 9.*

По построению функции $NearBy$ легко показать, что для некоторой группы G она возвращает множество соседних с ней групп (см. определение 11). То есть каждая из этих групп может вместе с G сформировать неточную группу, и далее алгоритм выбирает ближайшую к G группу из этого множества и конструирует новую неточную группу. Корректность всех промежуточных множеств, а также других используемых функций напрямую следует из способа их построения.

Оценим сложность предложенного алгоритма. Рассмотрим одну итерацию цикла **repeat... until**. Пусть N_i — общее количество повторов в группах из $SetG \cup SetVG$ перед i-й итерацией. Тогда, следуя [38], для одного повтора поиск соседних с ним (который производится при помощи интервального дерева) в среднем имеет сложность $\mathcal{O}(M + \log N_i)$, где M — максимальное количество соседних повторов на всех итерациях алгоритма, найденных для одного из повторов. Значит, сложность каждой итерации цикла в среднем составляет $\mathcal{O}(N_i * (M_i + \log N_i))$, где $M_i = \max m_i$. Цикл **repeat... until** делает не более $E + 1$ итераций, где E — максимальное количество точек расширения у групп из $SetVG$ на выходе алгоритма. В итоге сложность алгоритма можно оценить в среднем как $\mathcal{O}((E+1) * N * (M + \log N))$, где E — максимальное количество точек расширения результирующих групп $SetVG$, $N = N_1$ — количество точных повторов в исходном документе (т.е. в нашем случае это выдача Clone Miner). Для реальных документов N сильно колеблется, и в наших экспериментах оно достигало десятков тысяч, в то время как M не превосходило 10, а E не превосходило 20. Поэтому можно считать, что сложность представленного алгоритма в среднем составляет $\mathcal{O}(N * \log N)$. Реальное время работы алгоритма на различных документах представлено в таблице 1. Из этой же таблице видно, что решающий вклад в сложность вносит именно количество точных повторов, а не размер анализируемого документа.

6. АПРОБАЦИЯ

Предложенный алгоритм реализован в инструменте Duplicate Finder [11]. Для реализации интервального дерева инструмент использует библиотеку `intervaltree` [39].

Мы провели апробацию алгоритма и инструмента на 19 промышленных документах следующих типов: описание требований, руководство разработчика (programming guide), API-документации, руководство пользователя (см. таб. 1). Размер исследуемых документов достигал 3 Мб.

В ходе апробации мы получили следующие результаты. Большинство групп повторов (88,3–96,5% для разных документов) являются точными группами. Групп неточных повторов с одной точкой расширения — 3,3–12,5%, с двумя — 0–1,7%, с тремя — менее 1%, и т.д. В незначительном количестве встречаются группы неточных повторов с 11, 12, 13, и даже 16 точками расширения. Мы проанализировали вручную автоматически найденные группы повторов в документации ядра Linux (LKD) [12]. Было найдено 70 значимых текстовых групп (5,4%), 30 значимых групп с примерами кода (2,3%) и 1191 групп, являющихся ложноположительными срабатываниями (false positives) (92,3%). При этом была найдена 21 неточная группа,

что составило 21% от значимых групп повторов. Таким образом, после исключения ложноположительных срабатываний доля неточных повторов заметно возросла.

Проанализировав результаты апробации, мы пришли к следующим выводам.

1. В ходе наших экспериментов нам не удалось найти неточные повторы в исследуемых документах, которые бы наш алгоритм не нашёл. Хотя следует отметить, что утверждение о полноте алгоритма нуждается в более детальном обосновании.

2. Анализируя документацию ядра Linux мы увидели, что в ней отсутствует единый стиль — документация создавалась бессистемно, разными авторами: практически все повторы расположены локально, т.е. близко друг к другу (например, некоторый автор описывал функциональность какого-нибудь драйвера и применял copy/paste для описания его сходных свойств, но другой драйвер описывался уже другим автором, и тот никак не использовал текст, описывающий предыдущий драйвер). Соответственно, в этой документации практически отсутствуют повторно используемые фрагменты, встречающиеся во всем тексте. Примеры, предупреждения, замечания и пр. элементы документации предваряются различными вводными фразами, также не выдержаны в общем стиле. Таким образом, наш алгоритм может быть использован для анализа уровня унификации документации.

3. Алгоритм показывает хорошие результаты для групп из двух элементов, находя неточные группы с различным количеством точек расширения. Похоже, что неточных групп с бóльшим количеством элементов значительно меньше в принципе. Но, кроме того, иногда алгоритм некорректно выполняет склейку сложно устроенных точных повторов.

4. Многие группы повторов являются подписями к иллюстрациям и таблицам, колонтитулами, элементами оглавления и т.д., — то есть не представляют для нас интереса. Также многие повторы расположены “поверх” структуры документа, например, они могут захватывать часть заголовка и небольшой фрагмент текста после него. Эти проблемы происходят из-за того, что при поиске не учитывается структура документа.

5. Значение 0,15, использованное в определении неточных групп, не позволяет обнаружить некоторые существенные неточные группы (в основном, небольшие, по 10–20 токенов). Возможно, что вместо константы эффективнее использовать какую-либо функцию, зависящую, например, от длины неточного повтора.

6. Кроме того, часто в повтор не включается вариативная информация, находящаяся в начале или в конце, которую было бы целесообразно включить в повтор для его смыслового замыкания. Для решения этой проблемы требуется уточнить и формально определить, что такое смысловое замыкание фрагмента текста: как показывают наши эксперименты, это может быть сделано разными способами (простейший способ — замыкание до начала/конца предложения).

7. Алгоритм показал приемлемое время работы на реальных документах: до 5,4 минут в худшем случае, в среднем — 1,3 минуты.

7. ЗАКЛЮЧЕНИЕ

В данной работе дано формальное определение неточных повторов в документации программного обеспечения и представлен алгоритм обнаружения таких повторов. Проведена также апробация алгоритма на значительном количестве документации коммерческого и открытого программного обеспечения.

Результаты апробации показывают, что документация программного обеспечения содержит существенное количество точных и неточных повторов. Инструмент поиска неточных повторов может помочь в повышении качества документации.

Предложенный алгоритм выявляет значительное количество повторов, но требует дальнейшего улучшения. Основными проблемами являются неудовлетворительное качество найденных

Таблица 1. Результаты апробации

неточных повторов и большое количество ложноположительных срабатываний.

Также необходим детальный анализ видов неточных повторов, свойственных разным видам документации программного обеспечения. Интересными направлениями дальнейшего развития предложенного подхода может быть интеграция повторного использования документации и, в частности, спецификации требований, с автоматизированной разработкой тестов [40, 41], а также визуализация структуры повторов в стиле схем [42].

СПИСОК ЛИТЕРАТУРЫ

1. *Parnas, D.L.* Precise documentation: The key to better software, The Future of Software Engineering, Berlin, Heidelberg: Springer-Verlag, 2011, pp. 125–148.

2. *Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., and Streit, J.* Can clone detection support quality assessments of requirements specifications? Proceedings of the 32 ACM/IEEE International Conference on Software Engineering (ICSE'10), New York, NY, USA: ACM, 2010, vol. 2, pp. 79–88.

3. *Nosál', M.* Preliminary report on empirical study of repeated fragments in internal documentation, Proceedings of Federated Conference on Computer Science and Information Systems, 2016, pp. 1573–1576.

4. *Кознов Д.В., Романовский К.Ю.* DocLine: метод разработки документации семейства программных продуктов // Программирование. 2008. Т. 34. № 4. С. 1–13.

5. *Romanovsky, K., Koznov, D., and Minchin, L.* Refactoring the documentation of software product lines, Lecture Notes in Compute Science, Berlin, Heidelberg: Springer-Verlag, 2011, vol. 4980 of CEE-SET 2008, pp. 158–170.

6. *Wingkvist, A., Lowe, W., Ericsson, M., and Lincke, R.* Analysis and visualization of information quality of technical documentation, Proceedings of the 4th European Conference on Information Management and Evaluation, 2010, pp. 388–396.

7. *Koznov, D., Luciv, D., Basit, H.A., Lieh, O.E., and Smirnov, M.* Clone detection in reuse of software technical documentation, International Andrei Ershov Memorial Conference on Perspectives of System Informatics, 2015, Springer Nature, 2016, vol. 9609 of Lecture Notes in Computer Science, pp. 170–185.

8. *Луцив Д.В.* Задача поиска нечётких повторов при организации повторного использования документации / Д.В. Луцив, Д.В. Кознов, Х.А. Басит, А.Н. Терехов // Программирование. 2016. Т. 42. № 4. С. 39–49.

9. *Basit, H.A.* Efficient Token Based Clone Detection with Flexible Tokenization / H.A. Basit, S.J. Puglisi, W.F. Smyth et al, Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. New York, NY, USA: ACM, 2007, pp. 513–516.

10. *Bassett, P.G.* Framing software reuse: Lessons from the real World, Upper Saddle River, NJ, USA: Prentice-Hall, 1997.

11. Documentation Refactoring Toolkit. http://www.math.spbu.ru/user/kromanovsky/docline/ /index_en. html.

12. *Torvalds, L.* Linux Kernel Documentation, Dec 2013 snapshot. https://github.com/torvalds/linux/ /tree/ master/Documentation/DocBook/.

13. *Horie, M. and Chiba, S.* Tool support for crosscutting concerns of API documentation, Proceedings of the 9th International Conference on Aspect-Oriented Software Development, New York, NY, USA: ACM, 2010, pp. 97–108.

14. *Nosál', M.* Reusable software documentation with phrase annotations, Central Europ. J. Comput. Sci., 2014, vol. 4, no. 4, pp. 242–258.

15. *Oumaziz, M.A., Charpentier, A., Falleri, J.-R., and Blanc, X.* Documentation reuse: Hot or not? An empirical study, Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, 2017, Proceedings, Botterweck, G. and Werner, C., Eds., Cham: Springer-Verlag, 2017, pp. 12–27.

16. *Rago, A., Marcos, C., and Diaz-Pace, J.A.* Identifying duplicate functionality in textual use

cases by aligning semantic actions, Software Syst. Model., 2016, vol. 15, no. 2, pp. 579–603.

17. *Huang, T.-K., Rahman, Md.S., Madhyastha, H.V., Faloutsos, M., and Ribeiro, B.* An analysis of socware cascades in online social networks, Proceedings of the 22Nd International Conference on World Wide Web, New York, NY, USA: ACM, 2013, pp. 619–630.

18. *Williams, K. and Giles, C.L.* Near duplicate detection in an Academic Digital Library, Proceedings of the ACM Symposium on Document Engineering, New York, NY, USA: ACM, 2013, pp. 91–94.

19. *Zhang, Q., Zhang Yu., Yu, H., and Huang, X.* Efficient partial-duplicate detection based on sequence matching, Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA: ACM, 2010, pp. 675–682.

20. *Abdel Hamid, O., Behzadi, B., Christoph, S., and Henzinger, M.* Detecting the origin of text segments efficiently, Proceedings of the 18th International Conference on World Wide Web, New York, NY, USA: ACM, 2009, pp. 61–70.

21. *Ramaswamy, L., Iyengar, A., Liu, L., and Douglis, F.* Automatic detection of fragments in dynamically generated web pages, Proceedings of the 13th International Conference on World Wide Web, New York, NY, USA: ACM, 2004, pp. 443–454.

22. *Gibson, D., Punera, K., and Tomkins, A.* The volume and evolution of web page templates, Special Interest Tracks and Posters of the 14th International Conference on World Wide Web, New York, NY, USA: ACM, 2005, pp. 830–839.

23. *Vallés, E. and Rosso, P.* Detection of near-duplicate user generated contents: The SMS spam collection, Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents, New York, NY, USA: ACM, 2011, pp. 27–34.

24. *Barrón-Cedeño, A., Vila, M., Martí, M., and Rosso, P.* Plagiarism meets paraphrasing: Insights for the next generation in automatic plagiarism detection, Comput. Linguist., 2013, vol. 39, no. 4, pp. 917–947.

25. Antiplagiarism (in Russian). https://www.antiplagiat.ru/. Accessed January 16, 2018.

26. *Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., and Lopes, C.V.* SourcererCC: Scaling code clone detection to big-code, Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA: ACM, 2016, pp. 1157–1168.

27. *Jiang, L., Misherghi, G., Su, Z., and Glondu, S.* DECKARD: Scalable and accurate tree-based detection of code clones, Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA: IEEE Computer Soc., 2007, pp. 96–105.

28. *Cordy, J.R. and Roy, C.K.* The NiCad clone detector, in Proceedings of IEEE 19th International Conference on Program Comprehension, 2011, pp. 219–220.

29. *Akhin, M. and Itsykson, V.* Tree slicing in clone detection: Syntactic analysis made (semi)-semantic (in Russian), Model. Anal. Inform. Syst., 2012, vol. 19, no. 6, pp. 69–78.

30. *Zeltser, N.G.* Automatic clone detection for refactoring, Proc. Inst. Syst. Program., 2013, vol. 25, pp. 39–50.

31. *Wagner, S. and Fernández, D.M.* Analyzing text in software projects, The Art and Science of Analyzing Software Data, Elsevier, 2015, pp. 39–72.

32. *Korshunov, A. and Gomzin, A.* Topic modeling in natural language texts (in Russian), Proc. Inst. Syst. Program., 2012, vol. 23, pp. 21–242.

33. Tomita-parser — Yandex Technologies (in Russian). https://tech.yandex.ru/tomita/. Accessed January 16, 2018.

34. *Rattan, D., Bhatia, R., and Singh, M.* Software clone detection: A systematic review, Inform. Software Technol., 2013, vol. 55, no. 7, pp. 1165–1199.

35. *Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E.* Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms, 2004, vol. 2, no. 1, pp. 53–86.

36. *Bassett, P.G.* The theory and practice of adaptive reuse, SIGSOFT Software Eng. Notes, 1997, vol. 22, no. 3, pp. 2–9.

37. *De Berg, M., Cheong, O., van Kreveld, M., and Overmars, M.* Computational Geometry, Berlin Heidelberg: Springer-Verlag, 2008, pp. 220–226.

38. *Preparata, F.P. and Shamos, M.I.* Computational Geometry: An Introduction, Berlin: Springer-Verlag, 1985, pp. 359–363.

39. PyIntervalTree. https://github.com/chaimleib/intervaltree.

40. *Kolchin, A.V., Kotljarov, V.P., and Drobincev, P.D.* The method of test scenariogeneration in the environment of the insertion modeling, Control Syst. Mach., 2012, no. 6, pp. 43–48.

41. *Pakulin, N.V. and Tugaenko, A.N.* Model-based testing of Internet Mail Protocols, Proc. Inst. Syst. Program., 2011, vol. 20, pp. 125–141.

42. *Kudryavtsev, D. and Gavrilova T.* Diagrammatic knowledge modeling for managers: Ontologybased approach, Proceedings of the International Conference on Knowledge Engineering and Ontology Development, 2011, pp. 386–389.