Модел. и анализ информ. систем. Т. **19**, № **6** (2012) 69–78

УДК 004.05+004.422+004.4’23

Слайсинг над деревьями: метод обнаружения разорванных и переплетенных клонов в исходном коде программного обеспечения

Ахин М.Х., Ицыксон В.М.

Санкт-Петербургский государственный политехнический университет

e-mail: akhin@kspt.ftk.spbstu.ru, vlad@ftk.spbstu.ru

получена 12 сентября 2012

Ключевые слова: обнаружение клонов, слайсинг над деревьями, шаблоны над деревьями, поддержка программного обеспечения

В настоящее время практически любое программное обеспечение (ПО) содержит избыточный дублированный код (клоны), что приводит ко множественным проблемам на этапе поддержки такого ПО. За последние годы для решения проблемы дублирования было предложено множество различных подходов к обнаружению клонов, но большинство из них не рассматривают семантические свойства исходного кода ПО.

В данной работе предлагается усилить классический подход к обнаружению клонов на основе анализа абстрактных синтаксических деревьев (АСД) за счет использования дополнительной информации о слайсах АСД по переменным программы. Это позволяет эффективно обнаруживать разорванные (gapped) и переплетенные (intertwined) клоны – результаты предварительных экспериментов подтверждают применимость предложенного подхода на практике.

1. Введение

Объемы исходного кода ПО постоянно растут – так, кодовая база проектов с открытым исходным кодом удваивается каждые 14 месяцев [1]. Одной из основных причин столь быстрого роста является активное использование программирования в стиле «Копировать-Вставить» (Copy-and-Paste Programming), которое и приводит к чрезмерному дублированию кода, что, в свою очередь, усложняет поддержку такого ПО [2].

Дублирование кода означает, что одна и та же (или близкая) функциональность реализована в нескольких местах ПО, и при внесении изменений в эту функциональность необходимо корректно применить их ко всем копиям-клонам. В случае, если изменения были выполнены некорректно или неконсистентно, это может привести к возникновению трудно обнаруживаемых ошибок в ПО [4].

Для того, чтобы можно было избежать возникновения подобных трудностей, в последние годы большое внимание уделяется обнаружению клонов. Данная область программной инженерии изучает способы обнаружения интересных клонов для того, чтобы позднее их можно было устранить при помощи рефакторинга или других подходов, улучшая тем самым качество и упрощая поддержку ПО.

Большинство из существующих в настоящее время подходов к обнаружению клонов основаны на синтаксическом анализе в той или иной форме и не рассматривают семантические свойства анализируемого ПО. Причиной этого является крайняя сложность осмысленного анализа семантики ПО.

Несмотря на это, возможно частичное использование семантической информации для улучшения традиционных синтаксических подходов. В данной работе предлагается рассматривать задачу обнаружения клонов на уровне программных слайсов, которые строятся на основе слайсинга абстрактного синтаксического дерева программы (АСД) по переменным. Это позволяет обнаруживать дублирование кода с учетом влияния отдельных переменных, то есть эффективно обнаруживать переплетенные или разорванные по определенным переменным клоны[1]. Результаты проведенных экспериментов свидетельствуют о том, что данный подход улучшает результаты обнаружения клонов за счет введения незначительных накладных расходов на слайсинг, которые не снижают применимости подхода к большим программным проектам.

2. Клоны в программном обеспечении

Клон в программном обеспечении – это фрагмент исходного кода, который является «близким» к какому-либо другому фрагменту кода в ПО. В зависимости от того, какое определение близости используется, можно получить различные по своим характеристикам подходы к обнаружению клонов [3]. Выделяют следующие основные типы близости фрагментов исходного кода [2]:

- Синтаксическая близость – дублированные фрагменты кода с возможными модификациями:
  - *Тип I* – фрагменты кода, идентичные с точностью до форматирования.
  - *Тип II* – клоны типа I с некоторыми изменениями в именовании или типах переменных, значениях констант, и тому подобное.
  - *Тип III* – клоны типа II с дополнительными модификациями (например, добавлением или удалением операторов).
- Семантическая близость – фрагменты кода с одной и той же (или близкой) функциональностью:
  - *Тип IV* – семантически эквивалентные фрагменты кода, реализованные с использованием разных синтаксических конструкций.

[1]Разорванные клоны – похожие фрагменты кода с отличающимся друг от друга кодом в середине; переплетенные клоны – похожие фрагменты кода с переплетенными операторами.

Многие работы в области обнаружения клонов концентрируют свое внимание на обнаружении синтаксических клонов и не рассматривают клоны типа IV [5, 6, 7]. Некоторые исследователи предлагают анализировать семантическую близость фрагментов программы при помощи графов программных зависимостей (Program Dependence Graphs, PDG) [8, 9]. Однако, из-за большой вычислительной сложности построения и анализа PDG, все подходы на их основе используют различные аппроксимации для снижения требований по ресурсам.

3. Предлагаемый подход

В данной работе предлагается подход к обнаружению клонов в ПО, комбинирующий отдельные черты синтаксических и семантических подходов. Он основывается на анализе АСД программы, расщепленного по отдельным переменным, за счет чего достигается учет отдельных семантических компонентов работы с переменными. Рассмотрим предлагаемый подход более подробно.

3.1. Слайсинг над деревьями

Под слайсом S^v над АСД T по переменной v будем понимать такое поддерево из T, в котором любой узел T_i удовлетворяет следующему свойству: T_i или любой его потомок содержит использование или определение переменной v. Слайс S^v можно рассматривать как обобщенный шаблон использования переменной v в T – это позволяет использовать слайсы над деревьями для обнаружения клонов с точностью до отдельных переменных.

На рисунке 1 представлен пример слайсинга, в котором мы расщепляем исходное АСД T по переменным *lock* и *guard*, получая два слайса S^{lock} и S^{guard}. Данное преобразование может быть выполнено простым обходом АСД в глубину, после чего АСД T рассматривается как совокупность слайсов S^v по всем его переменным:

$$T = \{S^v | v \in T\}.$$

Таким образом, задача обнаружения клонов над АСД T_1 и T_2 сведена к задаче обнаружения клонов над слайсами между множествами $\{S_1^v\}$ и $\{S_2^v\}$, после чего возможно анализировать слайсы независимо друг от друга с использованием классических подходов для обнаружения клонов в АСД.

3.2. Обнаружение клонов

Классическим способом оценки близости АСД является расстояние редактирования. Для пары деревьев T_1 и T_2 расстояние редактирования определяется как взвешенная по типам сумма числа различных операций редактирования (добавление узла, удаление узла, изменение типа узла, и так далее), которые требуется выполнить над T_1, чтобы превратить его в T_2. Очевидно, что тривиальное решение задачи обнаружения клонов на основе попарного вычисления расстояния редактирования

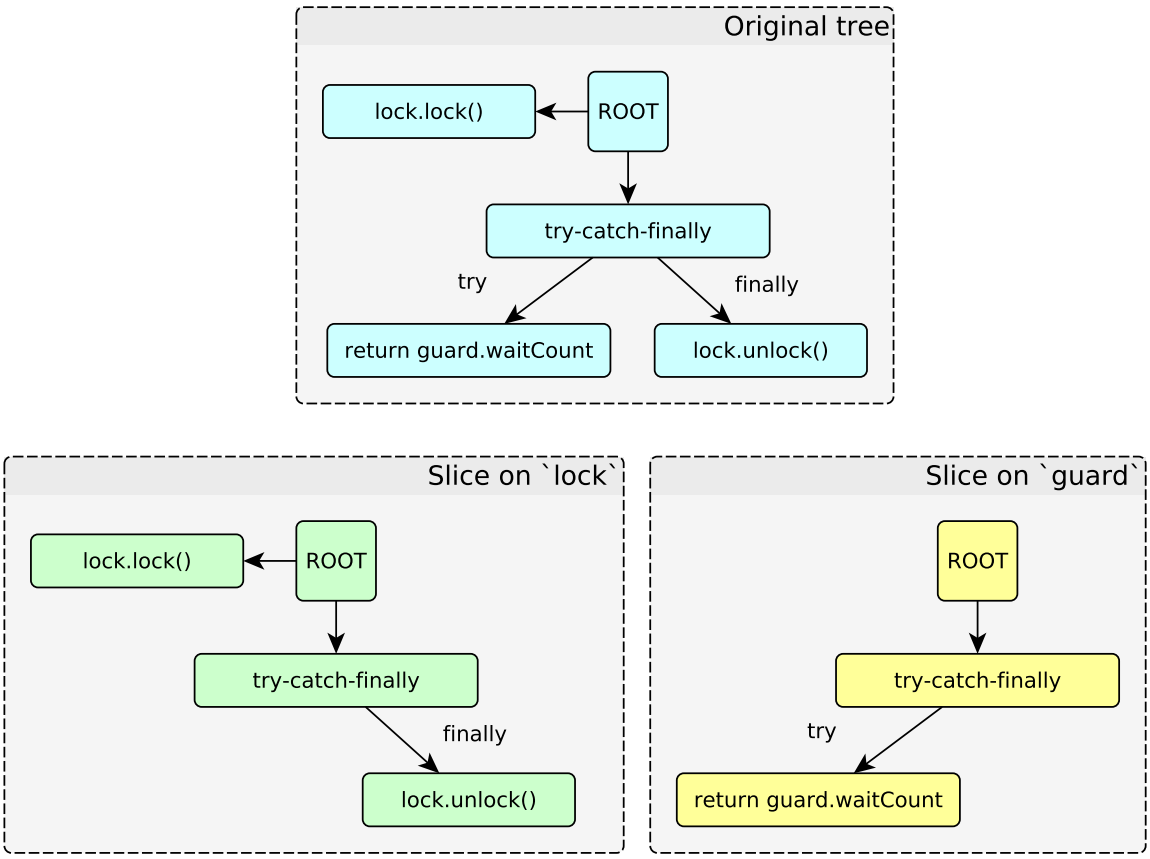

Рис. 1. Пример слайсинга над деревьями

является невозможным на практике из-за чрезмерной вычислительной сложности такого подхода.

Для обнаружения похожих слайсов в данной работе предлагается использовать подход на основе DECKARD [10]. Основная идея DECKARD заключается в сведении проблемы анализа близости АСД на основе расстояния редактирования к задаче многомерной кластеризации над характеристическими векторами.

Для заданного АСД T характеристический вектор задается через число вхождений в T шаблонов определенного вида. Более подробно, под q-шаблоном P^q понимают дерево высоты q, в котором узлы соответствуют возможным типам узлов из исходного АСД для заданного языка программирования. Число таких шаблонов для множества возможных типов узлов L (в которое также входит пустой тип ϵ) ограничено $C = |L|^{2^q-1}$.

Характеристический вектор $v^q(T)$ для АСД T задается следующим образом:

$$v^q(T) = (n_1, n_2, \ldots, n_C)\,,$$

где n_i – число вхождений i-го q-шаблона в T. В самом простом случае, который и рассматривается в данной работе, $q = 1$, и характеристические векторы соответствуют числу вхождений узлов определенного типа в заданное АСД T.

Замечательным свойством характеристических векторов является то, что расстояние между ними является оценкой снизу для расстояния редактирования между исходными АСД, то есть близким АСД соответствуют близкие характеристические векторы и наоборот. Можно показать, что расстояние редактирования $\delta(T_1, T_2)$ можно ограничить снизу как:

$$\delta(T_1, T_2) \geq D(v^q(T_1), v^q(T_2)),$$

где D – это l^2 норма для разности соответствующих векторов. Это позволяет свести задачу обнаружения похожих АСД к задаче поиска близких характеристических векторов в метрическом пространстве.

Данную задачу поиска можно рассматривать как задачу многомерной кластеризации, для решения которой существует множество эффективных алгоритмов. В DECKARD кластеризация выполняется при помощи Locality Sensitive Hashing (LSH) [11]. Данный метод кластеризации является представителем семейства подходов на основе снижения размерности и основывается на очень простой идее: если две точки p, q в заданном многомерном пространстве M близки, то любые их проекции $h(p), h(q)$ в пространство меньшей размерности N также будут близки.

Формально, функция h является locality sensitive, если для любых двух векторов v, w выполняется[2]:

$$\begin{cases} Pr[h(v) = h(w)] > P_1, & \text{если } D(v, w) < r \\ Pr[h(v) = h(w)] < P_2, & \text{если } D(v, w) > c \cdot r \end{cases}$$

Вероятности P_1 и P_2 обычно сопоставимы по величине, и для увеличения разницы между ними одновременно используют не одну, а k LSH-функций $g = h_1 \circ h_2 \circ \cdots \circ h_k$, тем самым переходя к вероятностям P_1^k и P_2^k. Очевидно, что при этом происходит снижение вероятности обнаружения близких векторов (так как $P_1^k < P_1$).

Для борьбы с этим последовательно применяют L групп LSH-функций $H = (g_1, g_2, \ldots, g_L)$, благодаря чему вероятность ошибки LSH ζ (вероятность того, что пара близких векторов не будет обнаружена) может быть сделана сколь угодно малой на основе следующего соотношения:

$$L = \left\lceil \frac{|log\ \zeta|}{log(1 - P_1^k)} \right\rceil$$

путем задания соответствующего значения L.

После проведения LSH для поиска близких характеристических векторов достаточно рассмотреть группы векторов с одинаковым значением хотя бы одной функции g_j из H. В случае, если выбранные параметры k и L показывают плохие результаты, они могут быть автоматически скорректированы в процессе анализа для улучшения качества обнаружения близких векторов.

Результаты, полученные после обнаружения клонов над слайсами, подвергаются дополнительной обработке:

[2]Также должно выполняться $P_1 > P_2$, чтобы применение данной функции имело смысл.

- удаление небольших клонов (если размер слайса меньше заданного порогового значения);

- объединение клонов для слайсов, относящихся к одной и той же локации в исходном коде (с сохранением информации о переменных, по которым проводился слайсинг).

В итоге для каждого обнаруженного клона также имеется дополнительная информация в виде того, слайсы по каким переменным в нем присутствуют. Для пары АСД T_1 и T_2 и соответствующих множеств слайсов $\mathbb{S}_1 = \{S_1^v\}$ и $\mathbb{S}_2 = \{S_2^w\}$ возможны следующие соотношения:

- $T_1 \sim T_2$ – T_1 и T_2 являются обычными клонами;

- $T_1 \not\sim T_2$,
  $\forall S_1 \in \mathbb{S}_1, \exists S_2 \in \mathbb{S}_2 : S_1 \sim S_2$,
  $\forall S_2 \in \mathbb{S}_2, \exists S_1 \in \mathbb{S}_1 : S_1 \sim S_2$ – T_1 и T_2 являются переплетенными клонами;

- $T_1 \not\sim T_2$,
  $\forall S_1 \in \mathbb{S}_1, \exists S_2 \in \mathbb{S}_2 : S_1 \sim S_2$,
  $\exists S_2 \in \mathbb{S}_2 : \nexists S_1 \in \mathbb{S}_1 : S_1 \sim S_2$ – T_1 и T_2 являются разорванными клонами,

где $A \sim B$ обозначает, что элемент A похож на элемент B. Таким образом, видно, что слайсинг позволяет обнаруживать не только обычные клоны, но и разорванные и переплетенные клоны по различным переменным в программе.

4. Экспериментальные результаты

На основе предложенного подхода был реализован прототип системы обнаружения клонов для языка программирования Java. Работа с АСД (построение, анализ и генерация слайсов) выполняются при помощи Compiler Tree API; кластеризация на основе LSH была реализована самостоятельно.

В качестве цели для экспериментальных исследований был выбран проект Guava, который является набором библиотек Java различного назначения от компании Google и содержит чуть более 100 тысяч строк исходного кода. Обнаружение клонов выполнялось в двух режимах, – с и без слайсинга, – после чего результаты сравнивались по общему времени анализа[3] и числу обнаруженных клонов (см. таблицу 1).

Таблица 1. Результаты экспериментов над проектом Guava

	Без слайсинга	**Со слайсингом**
Время анализа	10.5 с	12.2 с
Число найденных групп клонов	78	154
Средний размер группы клонов	2.65	4.23

[3]Эксперименты проводились на рабочей станции с процессором Intel Core i7-620M и 8 Гб оперативной памяти.

Как видно из результатов, хотя слайсинг и несколько увеличивает общее время анализа (из-за появления дополнительных вычислительных затрат на его проведение), он также увеличивает и количество обнаруженных клонов. На рисунке 2 приведены два примера клонов, обнаруженных с использованием слайсинга по переменным (разница между клонами выделена серым цветом).

В результате можно сделать вывод о том, что использование подхода на основе слайсинга позволяет находить сложные клоны, отличающиеся только слайсами по некоторым переменным. В данном прототипе информация об отличающихся слайсах не используется – ее применение для дальнейшего улучшения обнаружения клонов остается одним из направлений дальнейших исследований.

Следующим шагом является более подробное исследование предложенного подхода: планируется провести эксперименты на большем числе различных проектов на языке Java, исследовать вопрос масштабируемости слайсинга и рассмотреть то, насколько интересными[4] являются обнаруживаемые с его помощью клоны.

5. Другие работы в этой области

Наиболее близкими к предлагаемому подходу являются работы Джианг и Кринке, в которых для обнаружения клонов используется анализ на основе PDG, что позволяет обнаруживать некоторые типы семантических клонов [8, 9]. Принципиальным отличием предложенного в данной статье подхода является использование АСД вместо PDG, тем самым достигается значительное снижение вычислительной сложности и, вместе с тем, за счет использования слайсинга – сохранение возможности обнаруживать разорванные и переплетенные клоны.

Описанный подход также близок к подходам на основе анализа шаблонов над АСД [12, 13]. Данная группа подходов основана на выводе обобщенных шаблонов из АСД при помощи структурной абстракции или анти-унификации, после чего поиск клонов осуществляется над полученными шаблонами. В отличие от этого, в предлагаемом подходе анализ выполняется над шаблонами использования переменных в АСД в виде слайсов над деревьями, после чего результаты анализа отдельных шаблонов объединяются в итоговые результаты обнаружения клонов в исходном коде ПО.

Еще одной группой подходов к обнаружению клонов являются подходы на основе анализа исходного кода как текста и/или последовательности токенов [7, 14, 4]. Несмотря на то, что такие подходы являются крайне эффективными при обнаружении клонов типа I и II, они не могут использоваться для обнаружения клонов типа III и IV, что ограничивает сферу их возможного применения.

6. Заключение

В данной статье представлен новый подход к обнаружению клонов, основанный на анализе слайсов над деревьями, позволяющий находить переплетенные и разорван-

[4]Под интересным клоном понимается клон, рефакторинг которого приводит к улучшению сопровождаемости ПО.

```
// Clone on 'lock' and 'guard.monitor' variables
public boolean hasWaiters(Guard guard) {
  if (guard.monitor != this) {
    throw new IllegalMonitorStateException();
  }
  lock.lock();
  try {
     return guard.waiterCount > 0;
  } finally {
    lock.unlock();
  }
}
public int getWaitQueueLength(Guard guard) {
  if (guard.monitor != this) {
    throw new IllegalMonitorStateException();
  }
  lock.lock();
  try {
     return guard.waiterCount;
  } finally {
    lock.unlock();
  }
}
```

```
// Clone on 'element' and 'iterator' variables
public static boolean contains(Iterator<?> iterator, @Nullable Object element)
{
  if (element == null) {
    while (iterator.hasNext()) {
      if (iterator.next() == null) {
        return true;
      }
    }
  } else {
    while (iterator.hasNext()) {
      if (element.equals(iterator.next())) {
        return true;
      }
    }
  }
  return false;
}
public static int frequency(Iterator<?> iterator, @Nullable Object element) {
  int result = 0;
  if (element == null) {
    while (iterator.hasNext()) {
      if (iterator.next() == null) {
        result++;
      }
    }
  } else {
    while (iterator.hasNext()) {
      if (element.equals(iterator.next())) {
        result++;
      }
    }
  }
  return result;
}
```

Рис. 2. Примеры обнаруженных разорванных клонов

ные клоны путем анализа шаблонов использования переменных в программе и их дальнейшего объединения.

Проведенные экспериментальные исследования показывают, что предложенный подход способен улучшить результаты обнаружения клонов по сравнению с традиционными подходами на основе анализа АСД – в результате анализа проекта Guava было обнаружено большое число переплетенных и разорванных клонов. Вместе с тем, дополнительные расходы на проведение слайсинга являются незначительными и практически не влияют на общую производительность анализа.

В дальнейшем планируется более подробно изучить предложенный подход на большем числе тестовых проектов. Еще одним направлением дальнейшего развития данной работы является использование информации о слайсах над переменными для автоматизированного или автоматического принятия решений о возможных способах устранения клонов.

Список литературы

1. Deshpande A., Riehle D. The Total Growth of Open Source // Conference on Open Source Systems. Springer Verlag, 2008.

2. Roy C. K., Cordy J. R. A Survey on Software Clone Detection Research. TR 2007-541, School of Computing, Queen's University, 2007.

3. Ахин М.Х., Ицыксон В.М. Обнаружение клонов исходного кода: теория и практика // Системное программирование. 2010. 5. С. 145–163.

4. Li Z., Lu S., Myagmar S., Zhou Y. CP-Miner: a Tool for Finding Copy-Paste and Related Bugs in Operating System Code // Conference on Symposium on Opearting Systems Design & Implementation, USENIX Association, Berkeley, 2004. P. 20–20.

5. Kontogiannis K. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics // Working Conference on Reverse Engineering. IEEE Computer Society. Washington, DC, 1997. P. 44–44.

6. Koschke R., Falke R., Frenzel P. Clone Detection Using Abstract Syntax Suffix Trees // Working Conference on Reverse Engineering. IEEE Computer Society. Washington, DC, 2006. P. 253–262.

7. Hummel B., Juergens E., Heinemann L., Conradt M. Index-based Code Clone Detection: Incremental, Distributed, Scalable // International Conference on Software Maintenance. IEEE Computer Society. Washington, DC, 2010. P. 1–9.

8. Gabel M., Jiang L., Su S. Scalable Detection of Semantic Clones // International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 2008. P. 321–330.

9. Krinke J. Identifying Similar Code with Program Dependence Graphs // Working Conference on Reverse Engineering. IEEE Computer Society. Washington, DC, 2001.

10. Jiang L., Misherghi G., Su S., Glondu S. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones // International Conference on Software Engineering. IEEE Computer Society. Washington, DC, 2007. P. 96–105.

11. Andoni A., Indyk P. Near-optimal Hashing Algorithms for Approximate Nearest Neighbour in High Dimensions. Commun. // ACM. 2008. 51, 1. P. 117–122.

12. Evans W., Fraser C., Ma F. Clone Detection via Structural Abstraction // Software Quality Journal. 2009. 17, 4. P. 309–330.

13. Bulychev P., Minea M. Duplicate Code Detection Using Anti-unification // Spring Young Researchers Colloquium on Software Engineering, 2008. P. 51–54.

14. Kamiya T., Kusumoto S., Inoue K. CCFinder: Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code // IEEE Trans. Softw. Eng. 2002. 28, 7. P. 654–670.

Tree Slicing in Clone Detection: Syntactic Analysis Made (Semi)-Semantic

Marat Akhin, Vladimir Itsykson

Keywords: Clone Detection, Tree Slicing, Tree Patterns, Software Maintenance

Nowadays most of software contains code duplication that leads to serious problems in software maintenance. A lot of different clone detection approaches have been proposed over the years to deal with this problem, but almost all of them do not consider semantic properties of the source code. We propose to reinforce traditional tree-based clone detection algorithms by using additional information about variable slices. This allows to find intertwined/gapped clones on variables; preliminary evaluation confirms applicability of our approach to real-world software.

Сведения об авторах:
Ахин Марат Халимович, СПбГПУ, аспирант.
Ицыксон Владимир Михайлович, СПбГПУ, доцент.