МЕТОДЫ ПРОВЕДЕНИЯ СТАТИЧЕСКОГО АНАЛИЗА ПРОГРАММНОГО КОДА

Беликов Дмитрий Владиславович

студент, МИРЭА — Российский технологический университет, РФ, г. Москва

Болбаков Роман Геннадьевич

научный руководитель,

Аннотация. Статья посвящена методам проведения статического анализа программного кода. Разобраны 3 метода, приведены примеры их работы, способ использования, представлены примеры ошибок, находимых статическим анализом

Ключевые слова: статический анализ, дефект, тестирование ПО

Статический анализ кода – это анализ программного обеспечения, производимый без выполнения исследуемой программы с помощью специального ПО. Обычно он производится над исходным кодом исследуемого приложения.

Исправление плохо построенного ПО обходится дорого – чем больше работы уже сделано, тем выше цена ошибки. Так, согласно отчёту Консорциума по качеству информации и программного обеспечения (США) в 2020 году ошибки ПО стоили компаниям 2,07 трлн $ [9]. Эта цена складывается, как из затрат времени разработчиков на исправление ошибок, так и из потерь клиентов и заказов, и в целом недоделанных частей ПО. Рекомендуется тратить не менее 20% времени разработки на поиск и исправление ошибок ПО [8], что уже обеспечивает большую стоимость багов для компании.

Основное преимущество статического анализа кода заключается в том, что он обеспечивает важную информацию об исходном коде до его выполнения. Он позволяет на раннем этапе обнаружить недочёты в исходном коде программы – неочевидные дефекты, плохое оформление [6].

Статический анализатор принципиально состоит из 2 частей: лексического анализатора, который приводит исходный код в нужный для анализа вид, и анализатора синтаксического дерева, работающего на результатах синтаксического анализатора.

Первое, что делает анализатор, пытаясь понять фрагмент кода, – это разбивает его на более мелкие фрагменты, также называемые токены. Токен может состоять либо из одного символа, например открывающейся скобки, либо из литералов (таких как целые числа, строки), или из зарезервированных ключевых слов этого языка. Символы, не относящиеся к семантике языка, такие как пробелы, комментарии и т. д., часто отбрасываются сканером.

Далее происходит парсинг этих токенов. На данном этапе у нас есть только словарный запас языка, но сами по себе токены ничего не говорят о грамматике языка. Синтаксический анализатор берет эти токены, проверяет, соответствует ли последовательность, в которой они появляются, синтаксису языка, и организует их в древовидную структуру. Её называют абстрактным синтаксическим деревом (AST).

Беликов Д.В. МЕТОДЫ ПРОВЕДЕНИЯ СТАТИЧЕСКОГО АНАЛИЗА ПРОГРАММНОГО КОДА // Студенческий форум: электрон. научн. журн. 2022. № 13(192). URL: https://nauchforum.ru/journal/stud/192/108071 (дата обращения: 30.08.2025).

Рассмотрим основные методы статического анализа [4].

Некоторые ошибки легко найти, даже без использования сложных технологий. Например, в C/C++ распространенной ошибкой является ввод = вместо == при проверке условий. Мы можем легко обнаружить этот «шаблон ошибки», проверив, используется ли оператор = для проверки условий. Обычно это выполняется путем сопоставления шаблона кода в программе с ожидаемым шаблоном ошибки на уровне абстрактного синтаксического дерева. [6]

Одним из основных преимуществ сопоставления с образцом является то, что способ может выполняться довольно быстро даже с большой базой кода и даже с частично написанными программами.

Рассмотрим ряд возможных шаблонов для языка Java [3]:

- Сравнение с NaN вместо использования метода isNaN. По стандарту языка Java, сравнение оператором == с NaN всегда возвращает ложное значение (даже при сравнении NaN с NaN), использование обычных операторов == не имеет смысла.

return a == Double.NaN; // данное сравнение всегда вернёт false

- Сравнение ссылок вместо значения по этим ссылкам. Если типы переопределяют метод equals, то обычно следует вызывать его для сравнения двух объектов по значению.

```
String a = "Abc";
String b = "ggg";
return a == b; // Следует использовать a.equals(b)
```

Рисунок 1. Сравнение строк по ссылке

- Переопределённый метод equals не учитывает возможность что один из параметров может быть null.

При анализе потока данных (DFA) собирается информация о данных в программе во время выполнения. Этот анализ обычно выполняется путем обхода графа потока управления (CFG) программы [5]. Граф потока управления можно представить как абстрактное представление функций в программе на графе. Каждый узел в графе представляет базовый блок, а направленные ребра используются для представления переходов в потоке управления (рисунок 2).

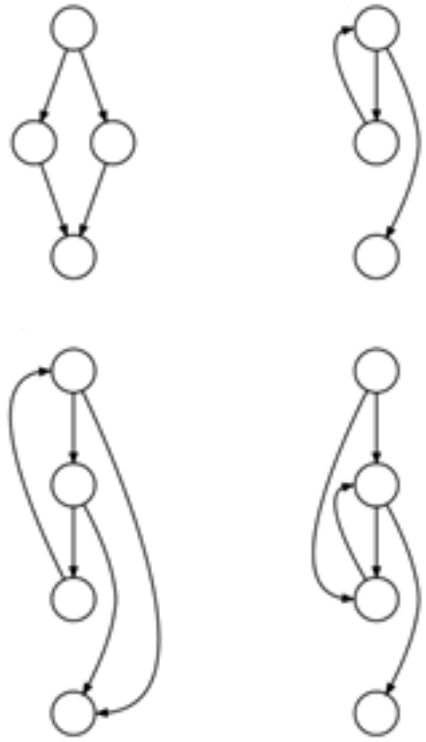

Рисунок 2. Граф потока управления

Рассмотрим основы анализа потока данных [2]:

Для каждого блока можно выделить наборы переменных, на примере выражения a = b + c:

- используемые переменные (b, c);
- уничтожаемые определения (a – старое значение переменной изменяется);
- новые определения (a – так как её значение меняется, её можно считать переопределённой);

Определением переменной называется присвоение переменой значения.

Считается, что определение d достигает точку p, если существует путь от d к p, в котором выражение d не уничтожается каким-то другим выражением [1].

Рассмотрим пример кода на рисунке 3:

```
d1: y := 3
d2: x := y
d3: y := 4
d4: x := y
```

Рисунок 3. Код для достигающих определений

В данном примере определение d1 является достигающим для выражений d2 и d3. Но оно не является достигающим для выражения d4, т.к. определение d3 переопределяет переменную y (то есть «уничтожает» её).

Для каждого блока b можно определить трансферную функцию f_b – функция, переводящая множество входящих определений In в множество выходящих определений Out (Out[b] = f_b(In[b]). Для этого определяются следующие множества:

- генерируемые определения Gen[b] – определения, создаваемые в блоке b;
- уничтожаемые определения Kill[b] – определения, переопределяемые в блоке b;
- распространяемые определения, равные In[b] - Kill[b] – набор определений, которые переживают блок b, и не уничтожаются в нём.

В результате выход из блока b является множеством Out[b] = Gen[b] U (In(b) - Kill[b]), то есть набором определений, созданных в блоке, плюс определениями, которые сохранились с входа в блок и не были переопределены в нём.

Третий метод – символьное выполнение

Важная задача символьного выполнения – определять диапазон значений, приводящих к выполнению того или иного участка кода.

Во время символьного выполнения входные данные программы заменяются свободными переменными. Если выполнение встречает выражение ветвления, анализатор разветвляется и проходит по обеим ветвям с ограничениями, определёнными для данной ветви.

Рассмотрим пример кода на рисунке 4.

```
y = read ();
y = 2 * y;
if (y == 12) {
    //..
}
...
```

Рисунок 4. Пример кода для символьного исполнения

Анализатор создаёт временную переменную s, присваивает ей значение read(). Когда код доходит до ветвления, переменной s присваивается 6, потому что это то значение конечно сделает условие истинным, и выполняет эту ветвь, заменив y на s. В то же время программа выполняет и другую ветку, учитывая, что y не равно 12.

Когда пути завершаются, символическое выполнение вычисляет конкретное значение для s путем решения накопленных ограничений для каждого пути. Эти конкретные значения можно рассматривать как тестовые примеры, которые могут помочь разработчикам воспроизводить ошибки. В этом примере решатель ограничений определит, что для достижения блока if y должно быть равно 6.

В данной статье был приведён обзор трёх методов статического анализа программного кода – сопоставления с шаблоном, анализа потока данных и символьного выполнения.

Список литературы:

1. Альфред Ахо, Моника Лам, Рави Сети, Джеффри Ульман. Компиляторы: принципы, технологии и инструментарий — 2-е издание. — М.: «Вильямс», 2008. — 1184 с. — 1500 экз. —

ISBN 978-5-8459-1349-4

2. Carnegie Mellon University [Электронный ресурс] — режим доступа: https://www.cs.cmu.edu/afs/cs/academic/class/15745-s03/public/lectures/L4_handouts.pdf (дата обращения: 26.03.2022)

3. ErrorProne [Электронный ресурс] — режим доступа: https://errorprone.info/bugpatterns (дата обращения: 26.03.2022)

4. Fan G. Practical static code analysis: challenges, methods, and solutions: дис. – 2020.

5. GeeksForGeeks [Электронный ресурс] — режим доступа: https://www.geeksforgeeks.org/data-flow-analysis-compiler/ (дата обращения: 26.03.2022)

6. Gosain A., Sharma G. Static analysis: A survey of techniques and tools // Intelligent Computing and Applications. – Springer, New Delhi, 2015. – С. 581-591.

7. OpenSource For U [Электронный ресурс] — режим доступа: https://www.opensourceforu.com/2011/09/joy-of-programming-technology-behind-static-analysis-tools/ (дата обращения: 26.03.2022)

8. Raygun [Электронный ресурс] — режим доступа: https://raygun.com/blog/cost-of-software-errors/ (дата обращения: 26.03.2022)

9. Synopsys [Электронный ресурс] — режим доступа: https://www.synopsys.com/blogs/software-security/poor-software-quality-costs-us/ (дата обращения: 26.03.2022)