ИСПОЛЬЗОВАНИЕ СТАТИЧЕСКОГО АНАЛИЗА ИСХОДНОГО КОДА В РАЗРАБОТКЕ И ТЕСТИРОВАНИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Беликов Дмитрий Владиславович

студент, Российский технический университет МИРЭА, РФ, г. Москва

Болбаков Роман Геннадьевич

научный руководитель,

Аннотация. Статья посвящена использованию статического анализа исходного кода в разработке ПО. Выделены преимущества использования статических анализаторов кода при тестировании приложений, приведены конкретные примеры их пользы, описаны особенности и ограничения их использования.

Ключевые слова: статический анализ, дефект, тестирование ПО.

Тестирование играет жизненно важную роль в разработке программного обеспечения и является неотъемлемой частью жизненного цикла его разработки. Одной из его задач является поиск и устранение дефектов в программе. Хорошим помощником в этом деле является статический анализ кода – в этой статье рассматривается необходимость использования этого инструмента в разработке 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 [3].

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. При этом не требуется запускать всю программу – достаточно исходного кода приложения, или даже его части. Статические анализаторы проверяют все ситуации, независимо от частоты их выполнения. Таким образом, статический анализ хорошо дополняет юнит-тесты и другие методы контроля качества кода. [6]

Использование статического анализа для поиска ошибок особенно интересно разработчикам из-за того, что его можно применять на ранних этапах жизненного цикла разработки ПО, и он способен находить недочёты, которые сложно обнаружить обычным тестированием. [5]

Приведём примеры ошибок, которые обнаруживаются статическими анализаторами.

Более слабые анализаторы (например, встроенные в IDE или компиляторы) могут выдавать предупреждения по таким недочётам как:

- неинициализированные и неиспользуемые переменные
- использование неинициализированной переменной;
- очевидный выход за пределы массива;

- недостижимые участки кода;

Продвинутые анализаторы, поставляемые в виде отдельного ПО, способны обнаруживать более сложные просчёты – приведём примеры из реальных проектов [2]:

- ошибки при копировании однообразных строк кода – повторы в присваивании:

```
public DataStoreEvent(DBIDs inserts, DBIDs removals, DBIDs updates) {

    super();

    this.inserts = inserts;

    this.removals = inserts;

    this.updates = inserts;

}
```

Здесь в конструкторе одна и та же переменная присваивается нескольким полям, хотя очевидно, что каждому поля соответствует свой параметр.

- Повторы в условиях:

```
void text_editor::set_highlight(

  const std::string& name,

  const ::nana::color& fgcolor,

  const ::nana::color& bgcolor)

{

  if (fgcolor.invisible() && fgcolor.invisible())

  {

    ...
```

Здесь дважды проверяется одно и тоже условие по обе стороны оператора «И». Судя по параметрам функции, вместо fgcolor во втором подвыражении хотели использовать аргумент bgcolor, но не изменили имя переменной при копировании.

- Использование указателя до его проверки на null:

```
owner->children.push_back(wd);

if (owner

      && owner->other.category == category::frame_tag::value)

    insert_frame(owner, wd);
```

Здесь сначала происходит обращение к памяти по указателю owner, и только потом он проверяется на неравенство нулю.

- Несоответствие использования результата функции со списком возможных значений, которые она возвращает:

```
month = fromShortMonthName(parts.at(1));

  if (month)

    day = parts.at(2).toInt(&ok);

  // If failed, try day then month

  if (!ok || !month || !day) {

    ...
```

Здесь функция fromShortMonthName, определённая в другом месте кода, возвращает значения от 1 до 12, и -1. Значение -1 должно означать, что в функцию передан некорректное название месяца. Однако в выделенном участке кода программист рассчитывает, что в качестве статуса ошибки ему будет возвращено нулевое значение, и сравнивает переменную month с 0, что не имеет смысла – эта переменная никогда не будет ему равна.

- Неправильный приоритет операций:

```
if (int icID = containingType.lookupInlineComponentIdByName (typeStr) != -1)
```

Здесь сначала происходит сравнение результата функции с -1, и далее icID получит значение 0 или 1, хотя скорее всего разработчик хотел сравнить именно icID с -1. Статический анализатор способен обращать снимание на такие ошибки.

- Многие другие виды ошибок.

При использовании статических анализаторов кода нужно учитывать, что они имеют некоторые ограничения и недостатки, уменьшающую точность анализа и усложняющую работу с его результатами [4].

1. Для повышения точности необходим полный исходный код. В зависимости от свойств недоступного кода, некоторые операции могут приводить или не приводить к ошибке. Анализатор не может определить, корректны ли все параметры, передаваемые в функцию, если её код недоступен. Это затрудняет использование сторонних библиотек.
2. Заметное количество ложных предупреждений (оценивается на уровне 10-20% [1]). Во многих случаях нельзя установить, возможен ли путь программы и данные, приводящие к ошибке. Пометка подозрительных участков предупреждениями приводит к большой доле ложных срабатываний, что снижает полезность статического анализа и усложняет работу с ним.
3. Важную роль играет длительность проведения анализа. Для больших проектов невозможно в приемлемое время провести все проверки, изучить все возможные интервалы значений переменных. Это также снижает точность анализа ради его ускорения и осуществимости.

Большим достоинством статического анализа является то, что его можно проводить на очень ранних этапах жизненного цикла – не стоит забывать об этой возможности.

Важно понимать, что статический анализатор не способен предложить значимую оптимизацию кода, найти проблемное с точки зрения производительности и используемой памяти место в программе. Он помогает найти достаточно простые ошибки в логике и коде программы – такие недочёты зачастую сложно обнаружить тестированием, так как они могут редко себя проявлять. Но в проекте таких недочётов может быть много, и они приводят к сбоям в работе, источник которых нелегко найти.

Отличия статических анализаторов кода

Помимо различий в качестве проводимого анализа, статические анализаторы имеют набор характеристик, который отличает их друг от друга [7, 9].

При выборе анализатора, в первую очередь нужно учитывать только те, которые поддерживают языки программирования, используемые в проверяемом проекте, и вашу платформу. Синтаксис и особенности у языков различны, поэтому каждый анализатор работает с конкретным языком или несколькими языками.

Также нужно определиться со стоимостью программы. Некоторые анализаторы распространяются бесплатно или имеют открытый исходный код, некоторые – платные.

Далее нужно обращать внимание на такие параметры, как:

- эффективность поиска разных типов ошибок;
- количество ложных срабатываний;
- удобство использования;
- сложность установки и конфигурации;
- возможность интеграции в среды разработки;
- количество настроек анализа;
- работа технической поддержки;
- возможность работы с исходным кодом и/или бинарными файлами.

Проанализируем статический анализатор PVS-Studio по данным параметрам [8].

Поддерживает платформы: Linux, macOS, Windows.

Поддерживает языки: C, C++, C#, Java.

Лицензия платная, есть испытательный период.

Отмечается хороший уровень технической поддержки.

Возможность выбрать только нужные виды ошибок.

Способна интегрироваться в большое количество IDE, такие как Visual Studio, Intelli IDEA, Eclipse и другие.

Отчёты предоставляются в разных форматах.

Работа только с исходным кодом.

Заключение:

В данной статье рассмотрено использование статического анализа кода в процессе разработки и тестирования приложений. Оно позволяет сократить количество дефектов и уязвимостей в программном обеспечении, а его преимуществами перед другими видами тестирования является то, что оно может выполняться уже на ранних этапах жизненного цикла продукта, и способно обнаруживать сложно находимые ошибки в исходном коде.

Список литературы:

1. Герасимов А.Ю. Обзор подходов к улучшению качества результатов статического анализа программ // Труды ИСП РАН. 2017. №3.

2. Ошибки, найденные с помощью PVS-Studio в открытых проектах [Электронный ресурс]. — режим доступа: https://pvs-studio.com/ru/blog/inspections/ (дата обращения: 28.11.2021).

3. Статический анализ кода – Википедия [Электронный ресурс]. — режим доступа: https://ru.wikipedia.org/wiki/Статический_анализ_кода (дата обращения: 25.11.2021).

4. Чукляев Евгений Игоревич Современные технологии статического и динамического

анализа программного обеспечения // Наукоемкие технологии в космических исследованиях Земли. 2016. №S2.

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

6. Formal concept analysis model for static code analysis // Carpathian J. Math. 37 (3), 49–58 (2021).

7. Hofer T. Evaluating static source code analysis tools: дис. – 2010

8. PVS-Studio – статический анализатор [Электронный ресурс]. — режим доступа: https://pvs-studio.com/ru/pvs-studio/ (дата обращения: 05.12.2021).

9. Source Code Analysis Tools [Электронный ресурс] — режим доступа: https://owasp.org/www-community/Source_Code_Analysis_Tools (дата обращения: 02.12.2021).