электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 12, декабрь 2015

УДК 004.42

Методы и перспективы статического анализа программного кода

Лебедев В.И., *студент*
105005, МГТУ им. Н.Э. Баумана, г. Москва, Россия,
кафедры «Компьютерные системы и сети»

Научный руководитель: Хохлов С.А., ст. преподаватель
105005, МГТУ им. Н.Э. Баумана, г. Москва, Россия,
кафедры «Компьютерные системы и сети»
bauman@bmstu.ru

Статическим анализом кода называют процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода.

Обзор кода (code review) является весьма надежным методом выявления дефектов программного кода, не смотря на то, что он же – один из самых старых [1]. Суть данного метода состоит в совместном внимательном чтении исходного кода и высказывании рекомендаций по его улучшению группой лиц, не участвовавших в написании данного кода. В процессе чтения кода выявляются ошибки или участи кода, которые могут стать ошибочными в будущем. При этом автор кода в ходе обзора не должен давать объяснений или комментариев о работе тех или иных участков кода и программы в целом. Алгоритм работы должен быть понятен непосредственно из текста программы и комментариев. Если это условие не выполняется, то код должен быть доработан. Как правило, обзор кода хорошо работает, так как программисты намного легче замечают ошибки в чужом коде.

Серьезным недостатком методологии обзора кода является высокая стоимость такого тестирования. Это связано с высокими требованиями к организации обзора. Необходимо регулярно собирать нескольких программистов для обзора нового кода или повторного обзора кода после внесения рекомендаций. При этом программисты должны регулярно делать перерывы для отдыха во избежание притупления внимания. Если пытаться просматривать сразу большие фрагменты кода, то внимание быстро притупляется и польза от обзора кода быстро сходит на нет. По той же причине нежелательно выполнение программистами какой-то другой работы параллельно с

просмотром кода, так как это опять-таки ведёт к потере внимания. Данные факты существенно ограничивают возможности совместного обзора для массивных программных комплексов.

Получается, что с одной стороны необходимо регулярно осуществлять обзор кода, а другой - это слишком дорого [2]. Компромиссным решением являются инструменты статического анализа кода. Они обрабатывают исходные тексты программ и выдают программисту рекомендации обратить повышенное внимание на определенные участки кода. Конечно, программа не заменит полноценного обзора кода, выполняемого коллективом программистов, однако соотношение польза/цена делает использование статического анализа весьма полезной практикой, применяемой многими компаниями. Данный метод позволяет не заменить обзор кода программистом, но существенно повысить КПД подобного обзора.

Задачи, решаемые программами статического анализа кода можно разделить на 3 категории:

1. выявление ошибок в программах;
2. рекомендации по оформлению кода. Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и тому подобные стилистические рекомендации;
3. подсчет метрик. Метрика программного обеспечения - это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Существует множество различных метрик, которые можно подсчитать, используя те ли иные инструменты.

Есть и другие, более специфичные сферы использования статического анализа кода. Например, статический анализ можно использовать для обучения и контроля деятельности новых сотрудников, еще недостаточно знакомых с принятыми в компании правилами и культурой кода.

Существует большое количество как коммерческих, так и свободно-распространяемых статических анализаторов кода. Статические анализаторы существуют для достаточно большого числа распространенных языков программирования, таких как C, C++, Java, Pascal, Python и другие [3].

Как и все прочие методы тестирования и поиска ошибок, статический анализ не является панацеей и имеет свои сильные и слабые стороны. Важно понимать, что

идеального способа тестирования ПО не существует. Для разных классов программного обеспечения разные методики будут давать разные результаты. Поэтому, как правило, обеспечение стабильно высокого качества ПО подразумевает использование сочетаний различных методов проверки кода.

Важное преимущество статического анализ состоит в возможности максимально быстрого нахождения и устранения ошибки в коде и минимизации, таким образом, цены исправления данной ошибки. Чем раньше ошибка выявлена, тем меньше стоимость ее исправления. Так согласно данным, приведенным в книге Макконнелла «Совершенный Код», исправление ошибки на этапе тестирования обойдется примерно в десять раз дороже, чем на этапе конструирования (написания кода) [4].

Инструменты статического анализа позволяют выявить большое количество ошибок этапа конструирования, что существенно снижает стоимость разработки всего проекта. Существуют, к примеру, статические анализаторы, работающие в фоновом режиме сразу после компиляции и, в случае нахождения потенциальной ошибки, уведомляющие об этом программиста (т.н. инкрементальный анализ).

Другие преимущества статического анализа кода:

1. полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки кода, как правило, не удается протестировать другими методами [4]. Это позволяет находить дефекты в обработчиках редких ситуаций, в обработчиках ошибок или в системе логирования;

2. статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа [4]. Это позволяет находить скрытые ошибки, которые могут проявить себя только спустя длительное время после разработки, например ошибки неопределенного поведения. Такие ошибки могут проявить себя при смене версии компилятора или при использовании других ключей для оптимизации кода;

3. можно легко и быстро обнаруживать опечатки и ошибки, связанные с использованием копирования кусков кода. Как правило, нахождение таких ошибок другими способами является неэффективной тратой времени и сил [3]. Обсуждая типовые ошибки, про подобные ляпы, как правило, не упоминают, но на практике на их выявление тратится существенное время.

Недостатки статического анализа кода:

1. статический анализ, как правило, слаб в диагностике утечек памяти и параллельных ошибок. Чтобы выявлять подобные ошибки, фактически необходимо виртуально выполнить часть программы. Это крайне сложно реализовать. Также подобные алгоритмы требуют очень много памяти и процессорного времени. Как правило, статические анализаторы ограничиваются диагностикой простых случаев. Более эффективным способом выявления утечек памяти и параллельных ошибок является использование инструментов динамического анализа [5];

2. программа статического анализа предупреждает о потенциально опасных местах. Это означает, что код может быть вполне корректен. Подобное свойство называется ложно-позитивными срабатываниями. Понять, указывает анализатор на ошибку или выдал ложное срабатывание, может только человек-программист. Необходимость просматривать и распознавать ложные срабатывания отнимает время и ослабляет внимание к тем участкам кода, где действительно содержатся ошибки.

Ошибки, которые обнаруживаются статическими анализаторами, весьма разнообразны. Некоторые анализаторы могут специализироваться на определенной области или типах дефектов, к примеру, проблемах с переносом кода между 32-х и 64-х битными машинами, или ошибках в параллельных процессах. Другие, поддерживают определенные стандарты кодирование, например MISRA-C, Sutter-Alexandrescu Rules, Meyers-Klaus Rules и т.п. Область статического анализа активно развивается, появляются новые диагностические правила и стандарты, некоторые правила напротив устаревают. Поэтому нет смысла пытаться сравнить анализаторы, на основании списков обнаруживаемых дефектов [6].

Дефекты программного кода, обнаруживаемые при помощи статического анализатора можно разделить на функциональные ошибки и нефункциональные. Функциональные ошибки являются следствием нарушения спецификаций

Нефункциональные ошибки не являются нарушением явных требований спецификаций и являются:

1. ошибками кодирования
2. нарушениями правил языка программирования
3. следствием некорректной работы с библиотеками
4. и т. д.

Важнейшим этапом анализа является формирование модели, с которой фактически и работает статический анализатор. Модель для обеспечения корректной работы должна соответствовать следующим требованиям:

1. Полнота представления объектов программы
2. Восстанавливаемость:

2.1. Идеальный вариант — возможность полного восстановления исходного кода

2.2. Желательный вариант — связь с исходным кодом

3. Эффективность поиска объектов

Типы моделей весьма разнообразны и применения какого-то типа из них зависит от целей и задач данного анализатора:

1.1. структурные

1.1.1. синтаксическое дерево разбора

1.1.2. абстрактное синтаксическое дерево (AST – abstract syntax tree)

1.2. Поведенческие

1.2.1. Граф потока управления (CFG – control flow graph)

1.2.2. Граф зависимостей по данным (DDG – data dependence graph)

1.2.3. Граф зависимостей программы (PDG – program dependency graph)

1.2.4. Представление в виде однократного статического присваивания (SSA - static single assignment)

1.3. Гибридные

1.3.1. Абстрактный семантический граф (ASG — abstract semantic graph)

Различные модели по-разному представляют программный код и применяются для решения весьма различных задач в рамках статического анализа. Именно целями анализа определяется используемый тип (или типы) модели в данном анализаторе [3]. Так, можно указать следующие зависимости типа модели от стоящей перед анализатором задачи:

1. для простейшего статического анализа достаточно структурных моделей (AST);
2. для анализа потока данных необходим учет семантики, т. е. применение CFG, SSA, PDG или ASG;
3. для анализа зависимостей используются DDG, PDG, SSA или ASG.

Говоря о показателях тех или иных систем статического анализа, следует остановиться на критериях оценивания. Наиболее общими для всех систем СА,

позволяющими получить общее представление о системе и сравнивать системы между собой, являются следующие оцениваемые параметры:

1. полнота, т. е. доля найденных системой дефектов по отношению к общему их количеству;
2. точность, т. е. доля истинных дефектов относительно общего числа срабатываний системы;
3. ресурсоёмкость.

После обзора существующих моделей необходимо осветить следующий этап статического анализа, который, наравне с построением модели можно назвать важнейшим — непосредственно анализ модели с помощью специальных алгоритмов. Встречаются следующие подходы к организации анализа:

1. на основе систем уравнений. Суть данного метода состоит в представление состояния программы как совокупности состояний всех её объектов в некой точке. Каждому оператору ставится в соответствие уравнение, связывающее состояние до выполнения и после. Данные уравнения оперируют состояниями программы, а решение системы уравнений является множеством состояний программы во всех точках. Целью данной работы является вычисление всех возможных состояний программы во всех точках. Обнаружение дефектов происходит на основе проверки вычисленных состояний. Проиллюстрируем на простом примере:

```
int a;
a = C;
a = b;
φ(…)
if(cond) Strue else Sfalse
```

Соответствующее уравнение:

$Sl = Sl\text{-}1 \cup \langle a, noninit\rangle$

$Sl = Sl\text{-}1 \setminus Ui\langle a, i\rangle \cup \langle a, C\rangle$

$Sl = Sl\text{-}1 \setminus Ui\langle a, i\rangle \cup Uj\langle a, bj\rangle$

$Sl = Ui\ Si$

$Strue = Sl\text{-}1 \cap cond$

$Sfalse = Sl\text{-}1 \cap \neg cond$

Уравнения системы могут решаться в любой последовательности, а завершением процесса решения определяется монотонностью уравнения и конечностью системы уравнений.

2. на основе абстрактной интерпретации

При использовании данного метода выполняется Аппроксимация конкретной семантики программы приближенной (или абстрактной) семантикой. Интерпретация программы – выполнение ее над конкретным доменом данных происходит как абстрактная интерпретация т.е. Интерпретация программы над выбранным абстрактным доменом. Абстрактная интерпретация позволяет снизить размерность задачи, не снижая существенно качество решения.

3. так же стоит упомянуть о более редких методах — на основе ограничений и на основе систем типов и эффектов.

Рассмотрев общие методы организации анализа, разумно перейти к обзору конкретных алгоритмов статического анализа [5]. Таковые включают в себя:

1. интервальный анализ;
2. анализ указателей;
3. ресурсный анализ;
4. строковый анализ;
5. анализ зависимостей;
6. а так же ряд других.

Суть интервального анализа состоит в представление состояния переменных интервальными значениями, например,

a = [10;20]

b = (-40;20) U (30;100)

c = [0;∞).

Затем, вычисление состояния после выполнения оператора языка производится на основе интервальной арифметики. Разбор кода простой программы с позиции интервального анализа приведен в следующей таблице.

Код программы	Представление кода при интервальном анализе

int a;
a=10;
int b = f(…);
if (b < 100){
…
a = a + b;
}
else{
…
a = b / 2; }
	<a, noninit>
	<a, [10:10]>
	<a, [10:10]>, <b, (-∞; ∞)>
	
	<a, [10:10]>, <b, (-∞; 100)>
	<a, [-∞+10:110]>, <b, (-∞; 100)>
	
	<a, [10:10]>, <b, [100; ∞)>
	<a, [50: ∞/2]>, <b, [100; ∞)>

Анализ указателей в целом схож с интервальным, но в отличие от него в процессе анализа хранятся не интервалы значений, а списки связей “points-to” и таким образом, значения одних объектов могут влиять на другие.

При ресурсном анализе основным объектом рассмотрения являются ресурсы т. е.

объекты программы, имеющие свой жизненный цикл, например выделенная динамическая память, файлы, сокеты, потоки и др. Обычно ресурсы характеризуются состоянием и атрибутами, при этом каждый тип ресурса изменяет свое состояние по определенным правилам. Допустим, возьмем динамический элемент. Правила его изменения довольно просты:

Создается (p = malloc(…))

Используется (*p)

Освобождается (free(p))

При ресурсном анализе данного объекта этом можно детектировать такие дефекты как: использование несозданного объекта, повторное освобождение, утечка памяти. Аналогично, для файла можно обнаружить попытку чтения из неоткрытый файл, попытку записи в файл для чтения и т. д. Отличие ресурсов от других объектов программы заключаются в том, что ни:

1. управляются операционной системой;

2. управление сменой состояний и атрибутов происходит через функции API.

Так же, статический анализ классифицируется ещё по двум важным показателям — по масштабу и по консервативность.

По масштабу разделяют внутрипроцедурный анализ и межпроцедурный. Внутрипроцедурный анализ характеризуется следующими условиями:

1. каждая функция анализируется отдельно;
2. информация не передается между функциями.

К достоинствам данного метода можно отнести относительно низкую ресурсоёмкость, что делает его весьма популярным (реализован в большинстве существующих статических анализаторов), а к недостаткам - неизбежное при применение данного метода снижение точности и полноты анализа.

Межпроцедурный анализ напротив состоит в том, что:

1. все функция анализируется совместно;
2. информация передается между функциями.

Точность и полнота такого анализа существенно выше, но и ресурсоёмкость очень высока, что делает такой подход среди анализаторов общего назначения редким, но даёт ему применение в сферах, где цена ошибки может быть непомерно высока, в частности, в сфере исследований космоса и проектирования космических аппаратов [3].

По степени консервативности анализа (по правилам объединения ветвей дерева) выделяют:

1. оптимистичный анализ (при объединении ветвей пересекает интервалы)
2. пессимистичный анализ (при объединении ветвей объединяет интервалы)

При оптимистичном анализе страдает полтона анализа (большее число ошибок может быть пропущено), но повышается точность (меньше ложных срабатываний), поэтому выбор сценария объединения ветвей зависит от стоящих перед системой требований и задач, давать в данном случае общие рекомендации затруднительно.

Теперь, обозрев методы и виды анализа, а так же способы организации представления кода в анализаторе, произведём обзор существующих статических анализаторов для языка C++ на предмет реализации тех или иных функций (прежде всего функции оценки кода по признаку кроссплатформенности).

Рассмотрев методы статического анализа и его виды, обозначим перспективные направления развития статического анализа. К таковым можно отнести прежде всего анализ кроссплатформенности (т.е. переноса кода между различными системами) и анализ возможности переноса кода между платформами различной разрядности (в наиболее

классическом случае – перенос между 32- и 64-битными системами). Данные области пока слабо представлены на рынке статических анализаторов и практически не представлены в области open-source продуктов, однако разработка и широкое внедрение таких анализаторов поможет существенно упростить проверку совместимости для кроссплатформенных приложений, а так же облегчит портирование программ на платформы другой разрядности. При этом, в данном случае статический анализатор не может быть полноценно заменен, например, компилятором или другим классом программ.

В рамках данной работы рассмотрим необходимые шаги для проектирования анализатора или модуля к уже существующему анализатору для проверки переносимости кода между платформами разной разрядности.

Важным этапом разработки такого анализатора или модуля является определение ошибок, которые он должен находить [7]. Целесообразно сформулировать наиболее общие правила, которым должен соответствовать код для того, что бы в нем не было возможно возникновение ошибок определенного типа (например, ошибки переполнения). Эти правила должны описывать различные виды ошибок. Для нашей области к таким ошибкам относятся следующие группы:

1. ошибки, связанные с переменной размерностью некоторых типов данных (т.н. memsize-типов), например ptr_diff и int64
2. ошибки, связанные с различным видом констант в разноразрядных системах
3. ошибки адресации памяти, связанные с разным видом и размером адресов

Данные виды ошибок являются наиболее частыми «скрытыми» проблемами, мешающими безаварийной работе программы на платформе другой разрядности [3]. После того, как правила для недопущения подобных ошибок сформулированы, необходимо представить их в математическом виде, в виде систем уравнений вхождений в множества. Это позволит реализовать эти правила программно, в виде функций проверки кода.

Заключение

В целом, можно сделать вывод, что применение данного многообразия средств даёт возможность решать весьма различные задачи, связанные с проверкой кода программы по тем или иным критериям.

В частности, весьма перспективным кажется внедрение функции проверки кода на возможность переноса на другие платформы и разделение на переносимый и непереносимый по итогам данной проверки. Подобное может быть реализовано через

проверку моделей на соответствие (т. е. стандартными методами статического анализа). При этом в ходе исследования не было обнаружено статических анализаторов для языка C++ (во всяком случае находящихся в открытом доступе), имеющих подобную функцию.

Использование подобного анализатора поможет существенно упростить разработку кроссплатформенного ПО, позволяя ещё на этапе проектирования избегать непереносимых решений, библиотек и т. д.

В ходе исследования были не только выделены перспективные направления развития такого класса программ, как статические анализаторы, но и обозначены шаги для проектирования перспективного анализатора переносимости кода между разноразрядными платформами. Классифицированы наиболее частые ошибки, препятствующие свободному переносу программы. На основании данной работы может быть осуществлено математическое представление правил для кода программы, а затем начато проектирование анализатора, реализующего функцию проверки кода программы на соответствие этим правилам

На основании данной результатов данной работы в настоящее время автором ведётся выполнения квалификационной работы бакалавра по разработке модуля статического анализа переносимости кода с 32- на 64-битные системы для статического анализатора CppCheck.

Список литературы

1. Ахо А. Компиляторы: принципы, технологии и инструменты. М.: Издательский дом «Вильямс», 2003. 421 с.
2. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. М.: Издательство «Мир», 1975. 295 с.
3. Карпов А.Н. 20 ловушек переноса C++-кода на 64-битную платформу // RSDN-Magazine. 2007. № 1. С. 67-75.
4. Карпов А.Н. Статический анализатор PVS-Studio и его аналоги // IT-Tech. 2013. № 6. С. 43-58.
5. Макконнелл С. Совершенный код. СПб.: Питер, 2007. 252 с.
6. Рыжков Е.А. Подходы к верификации и тестированию 64-битных приложений // Информационные технологии. 2008. № 7. С. 154-162.
7. Себеста Р. Основные концепции языков программирования. М.: Издательский дом «Вильямс», 2001. 370 с.

8. Ларус Д., Болл Т., Дас М. О Безошибочных программах // Открытые системы. 2007. №8. С. 120-128.