Федеральное государственное автономное образовательное учреждение
высшего образования «Длинное название образовательного учреждения
«АББРЕВИАТУРА»

На правах рукописи

Фамилия Имя Отчество

Длинное название диссертационной работы, состоящее из достаточно большого количества слов, совсем длинное длинное длинное длинное название, из которого простому обывателю знакомы, в лучшем случае, лишь отдельные слова

Специальность XX.XX.XX —
«Технология обработки, хранения и переработки злаковых, бобовых культур, крупяных продуктов, плодоовощной продукции и виноградарства»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
уч. степень, уч. звание
Фамилия Имя Отчество

Город — 2021

Оглавление

Стр.

Введение

В прошлом статический анализ в основном означал ручную проверку программ. Сегодня автоматические инструменты статического анализа приобрели популярность в индустрии разработки программного обеспечения, они значительно увеличивают продуктивность за счет автоматической проверки кода по широку диапазону критериев. Многие проекты по разработке программного обеспечения разрабатываются в соответствии с рекомендациями по кодированию, таких как MISRA C[1], стремясь к стилю программирования, который улучшает ясность и снижает риск появления ошибок. Для проектов с повышенными требованиями к безопасности соответствие подобным правилам написания кода настоятельно рекомендуется всеми текущими стандартами безопасности. Проверка соответствия с помощью инструментов статического анализа является обычной практикой.

Однако, для того чтобы предотвратить ошибку одного соотетствия правилам кодирования не достаточно. Это признано нормой MISRA C через особое правило, которое рекомендует более глубокий анализ: ”Минимизация сбоев во время выполнения должна быть обеспечивается использованием хотя бы одного из следующих подходов: (a) инструментом статического анализа; (b) инструментами динамического анализа; (c) явное написание проверок для обработки во время выполнения программы.”(MISRA C:2004, rule 21.1). Действующие стандарты безопасности требуют демонстрации отсутствия ошибок времени выполнения и состояния гонки, например: DO-178B/C, ISO-26262, EN-50128, IEC-61508.

Неудивительно, что существует множество различных инструментов статического анализа. Их сравнение и выбор наиболее подходящего инструмента - непростая задача. Первая проблема заключается в том, что термин статический анализ используется для широкого диапазона техник, которые концептуально сильно различаются. Их всех объединяет то, что они анализируют код не выполняя его. Их можно разделить на три большие группы: синтаксические чекеры, не чувствительные к путям (unsound) семантические анализаторы, чувствительные к путям (sound) семантические анализаторы.

Синтаксические чекеры. Ограничиваются исследованием синтаксиса программы. Большинство алгоритмически проверяемых правил MISRA C могут быть проверены на синтаксическом уровне. В MISRA C:2012, 116 из 143 правил

классифицированы как разрешимые, что в данном контексте подразумевает возможность проверки на синтаксическом уровне. Примеры: Lint? (1978).

Не чувствительные к путям семантические анализаторы. Сообщают о семантических ошибках в программе, таких как ошибки времени выполнения (деления на ноль, арифметические переполнения или переполнения буфера) и состояния гонки. Могут выдавать false positives (ложные срабатывания где нет настоящих дефектов) и false negatives (настоящий дефект, на котором анализатор не сработал). Примеры: Klocwork[2], CodeSonar[3], Polyspace Bug Finder[4] и Coverity[5].

Чувствительные к путям семантические анализаторы. В основном основаны на абстрактной интерпритации, формальном методе анализа программ, который обеспечивает математически строгий способ доказательства отсутствие дефектов без false negatives: ни один дефект не пропущен (из рассматриваемого типа дефектов). Могут сообщать о семантических ошибках в программе, включая ошибки времени выполнения и состояния гонки, также могут использоваться для доказательства функциональных утверждений, например что выходные значения всегда будут в ожидаемом диапазоне. False positives могут иметь место. Примеры: Astree[6], Frama-C[7], Polyspace Code Prover[8], AdaCore CodePeer[9].

Разницу между этими подходами можно проиллюстрировать на примере деления на 0. В выражении ”x/0”деление на ноль возможно обнаружить синтаксически, но не в выражении ”a/b”. ”Unsound”анализатор может не сообщить о делении на ноль для ”a/b”хотя деление на ноль может иметь место в сценариях, не рассмотренных анализатором. Когда ”sound”анализатор не вадает предупреждение о делении на ноль в ”a/b”это доказывает, что b никогда не будет 0.

Бенчмарки для инструментов статического анализа предоставляет основу для сравнения различных инструментов. Они должны (как минимум) точно определять, какие дефекты исследуются, взвешивать серьезность различных дефектов брать в расчет глубину анализа (syntactic vs. unsound semantic vs. sound semantic), определять false positives и false negatives оценки.

В

Целью данной работы является разработать подход к проверке качества автоматических инструментов анализа программ.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Изучить современные инструменты анализа программ : типы, характеристики, техники сравнения.
2. Сформулировать требования, критерии и методы, которые позволят ранжировать анализаторы кода согласно их сильным и слабым сторонам.
3. Разработать фреймворк оценки качества анализаторов кода.
4. Протестировать полученный фреймворк.

Научная новизна:

1. Впервые …
2. Впервые …
3. Было выполнено оригинальное исследование …

Практическая значимость …

Методология и методы исследования. …

Основные положения, выносимые на защиту:

1. Первое положение
2. Второе положение
3. Третье положение
4. Четвертое положение

В папке Documents можно ознакомиться в решением совета из Томского ГУ в файле `Def_positions.pdf`, где обоснованно даются рекомендации по формулировкам защищаемых положений.

Достоверность полученных результатов обеспечивается … Результаты находятся в соответствии с результатами, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на: перечисление основных конференций, симпозиумов и т. п.

Личный вклад. Автор принимал активное участие …

Объем и структура работы. Диссертация состоит из введения, 3 глав, заключения и 1 приложения. Полный объём диссертации составляет 19 страниц, включая 0 рисунков и 0 таблиц. Список литературы содержит 9 наименований.

Глава 1. Обзор

1.1 История автоматического анализа программ

В 1947 году появились термины «ошибка» (bug) и «отладка» (debugging). Грейс Мюррей, ученая из Гарвардского университета, работавшая с компьютером Mark II, обнаружила, что мотылек застрял в реле, из-за чего оно не вступало в контакт. Она подробно описала инцидент в рабочем журнале, приклеив мотылька лентой в качестве доказательства и назвав мотылька «ошибкой», вызывающей ошибку, а действие по устранению ошибки - «отладкой».

В то время тесты были сосредоточены на оборудовании, потому что оно было не так развито, как сегодня, и его надежность была важна для правильного функционирования программного обеспечения. Термин отладка был связан с применением исправлений для конкретной ошибки как одна из фаз в стадии разработки программного обеспечения. Проводимые тесты имели коррекционный характер и выполнялись для устранения ошибок, не дававших программе работать.

В 1957 году Чарльз Бейкер объясняет необходимость разработки тестов, чтобы гарантировать, что программное обеспечение соответствует заранее разработанным требованиям (тестирование), а также функциональным возможностям программы (отладка). Разработка тестов стала более важной по мере того, как разрабатывались более дорогие и сложные приложения, и стоимость устранения всех этих недостатков оказывала явный риск для прибыльности проекта. Особое внимание было уделено увеличению количества и качества тестов, и впервые качество продукта стало связано с фазой тестирования. Цель заключалась в том, чтобы продемонстрировать, что программа выполняет то, что от нее требовалось, с использованием ожидаемых и выдаваемых параметров.

В 1979 году Гленфорд Дж. Майерс радикально меняет процедуру обнаружения ошибок в программе: ”Тестирование программного обеспечения - это процесс запуска программы с целью поиска ошибок.”Обеспокоенность Майерса заключалась в том, что, преследуя цель продемонстрировать, что программа безупречна, можно подсознательно выбрать тестовые данные, которые имеют низкую вероятность вызвать сбои программы, тогда как если цель состоит в том, чтобы

продемонстрировать, что программа ошибочна, тестовые данные будут имеют большую вероятность их обнаружения, и мы будем более успешными в тестировании и, следовательно, в качестве программного обеспечения. С этого момента тесты будут пытаться продемонстрировать, что программа не работает должным образом, в отличие от того, как это делалось ранее, что приведет к новым методам тестирования и анализа.

В 1983 году была предложена методология, которая объединяет действия по анализу, пересмотру (revision) и тестированию в течение жизненного цикла программного обеспечения, чтобы получить оценку продукта в процессе разработки. Этап тестирования признан неотъемлемым этапом в разработке продукта, приобретая особое значение в связи с появлением инструментов для разработки автоматизированных тестов, которые заметно повысили эффективность.

В 1988 году Уильям Хетцель опубликовал «Рост тестирования программного обеспечения», в котором он переопределил концепцию тестирования как планирование, проектирование, создание, обслуживание и выполнение тестов и тестовых сред. Это в основном отразилось на появлении фазы тестирования на самом раннем этапе разработки продукта, этапе планирования. Если мы представим весь процесс разработки в виде конечной линии, где начало - это планирование, а конец - мониторинг проданного продукта, мы увидим, как фаза тестирования переместилась влево. Она появилась как этап пост- продакшнен, позже это был этап предпродакшн, а сейчас она находится на стадии завершения. Эта практика известна как Shift-Left.

Э. В. Дейкстра в лекции «О надёжности программ» утверждает, что тесты могут показать наличие ошибок в программе, но не могут доказать их отсутствие Dijkstra.

1.1.1 Уровни тестирования ПО

Модульное тестирование (Unit testing)
Интеграционное тестирование (Integration testing)
Приемочное тестирование (Acceptance testing)

1.1.2 Правила кодирования

На ряду с тестированием одним из подходов к написанию безопасного и переносимого ПО является использование в разработке стандартов кодирования.

Одним из подобных стандартов является MISRA. Проект MISRA (Motor Industry Software Reliability Association) был основан для создания руководства по разработке ПО для микроконтроллеров в наземных средствах по заказу правительства Британии. Первое руководство вышло в 1994 году, не было привязано к какому-либо языку. Первый связанный с языком C стандрат MISRA C стал общепринятым.

В MISRA C правила деляться на три основных категории: Mandatory, Required и Advisory. Mandatory - наиболее строгая категория, требующая постоянного выполнения. Required - менее строгая: возможны отклонения при условии документирования и обоснования. Advisory - правила, которым следовать не обязательно.

В MISRA-C:1998 перечислено 127 правил (93 обязательных и 34 рекомендательных).

В MISRA-C:2004 141 правило (121 обязательное и 20 рекомендательных). Правила разделены на 21 категорию.

В MISRA-C:2012 143 правила (каждое из которых может быть проверено статическим анализатором кода) и 16 директив (правил, соответствие которым открыто для интерпретаций или связано с процессами и процедурами). Правила делятся на обязательные, требуемые и рекомендательные; могут распространяться на отдельные единицы трансляции или на всю систему.

В MISRA C++

1.1.3 Первое поколение статических анализаторов. Lint

Первый инструмент статического анализа Lint появился в конце 1970-х. Впервые разработчики получили возможность автоматизировать обнаружение дефектов программного обеспечения на самых ранних этапах жизненного цикла приложения, когда их легче всего исправить. Кроме того, это давало разработчи-

кам уверенность в качестве своего кода перед релизом. Технология, лежащая в основе Lint, была революционной, она использовала компиляторы для проверки дефектов.

Однако Lint не разработатывался с целью выявления дефектов, вызывающих проблемы во время выполнения программы. Скорее, его цель заключалась в том, чтобы выделить подозрительные или непереносимые конструкции в коде и помочь разработчикам соблюдать общий формат при кодировании. Под ”подозрительным кодом”имееся в виду код, который, будучи технически правильным с точки зрения языка исходного кода (например C, C++), может быть структурирован так, чтобы он выполнялся способами, которые разработчик не предполагал. Lint являлся дополнением к компилятору. В то время как компилятор концетрировался на быстром и точном превращении программы в послдовательность бит, Lint концетрировался на ошибках в переносимости, стиле и эффективности. Из-за ограниченных возможностей анализа Lint, уровень шума был чрезвычайно высоким, часто превышая соотношение между шумом и реальными дефектами в соотношении 10 к 1.

Следовательно, обнаружение настоящих дефектов требовало от разработчиков проведения трудоемкой ручной проверки результатов Lint, что усложняло именно ту проблему, которую должен был устранить статический анализ. По этой причине Lint так и не получил широкого распространения в качестве инструмента обнаружения дефектов, хотя имел ограниченный успех в нескольких организациях. Фактически, как свидетельство качества технологии, лежащей в основе Lint, множество различных версий продукта по-прежнему доступны сегодня.

1.1.4 Второе поколение статических анализаторов

Почти два десятилетия статический анализ оставался скорее фикцией, чем коммерчески жизнеспособным производственным инструментом для выявления дефектов. В начале 2000 года появилось второе поколение инструментов (Stanford Checker). Используя новые технологии, оно расширяло возможности инструментов первого поколения от простого выявления нежелательных паттернов до покрытия путей выполнения. Эти инструменты могли анализировать целые базы кода, а не только один файл.

Сместив фокус с ”подозрительных конструкций”на ”дефекты времени выполнения разработчики статического анализа осознали необходимость в большем понимания внутреннего устройста программ. Это означало объединение сложного анализа путей с межпроцедурным анализом, чтобы понять, что происходит, когда поток управления переходит от одной функции к другой в рамках данной программы.

Несмотря на принятие и использование организациями, статический анализ 2-го поколения все еще не мог найти золотую середину между точностью и масштабируемостью. Некоторые решения были точными для небольшого набора типов дефектов, но не могли масштабироваться для анализа миллионов строк кода. Другие могли работать за короткое время, но имели показатели точности, аналогичные Lint, представляя похожие проблемы с ложными срабатываниеми и шумом. После внедрения в процесс разработки эти инструменты могут сообщать о дефектах в приемлемом соотношении, но только с ограниченными параметрами анализа.

Инструментам второго поколения требовались однородные среды сборки и разработки. Из-за этого внедрить их в какой то проект было сложной задачей, требовавшей больших усилий.

Борьба между точностью и маштабируемостью вылилась в проблему с ложными срабатываниями. Подобно шуму, от которого страдали инструменты первого поколения, ложные срабатывания тормозили распространению инструментов нового поколения.

1.1.5 Третье поколение статических анализаторов

Третье поколение инструментов статической проверки программ превосходит своих предшественников по всем параметрам и является неотъемлемой частью процессов и сред разработки.

Глава 2. Длинное название главы, в которой мы смотрим на примеры того, как будут верстаться изображения и списки

2.1 Одиночное изображение

2.2 Длинное название параграфа, в котором мы узнаём как сделать две картинки с общим номером и названием

2.3 Пример вёрстки списков

Глава 3. Вёрстка таблиц

3.1 Таблица обыкновенная

Заключение

Основные результаты работы заключаются в следующем.

1. На основе анализа …
2. Численные исследования показали, что …
3. Математическое моделирование показало …
4. Для выполнения поставленных задач был создан …

И какая-нибудь заключающая фраза.

Последний параграф может включать благодарности. В заключение автор выражает благодарность и большую признательность научному руководителю Иванову И. И. за поддержку, помощь, обсуждение результатов и научное руководство. Также автор благодарит Сидорова А. А. и Петрова Б. Б. за помощь в работе с образцами, Рабиновича В. В. за предоставленные образцы и обсуждение результатов, Занудятину Г. Г. и авторов шаблона *Russian-Phd-LaTeX-Dissertation-Template* за помощь в оформлении диссертации. Автор также благодарит много разных людей и всех, кто сделал настоящую работу автора возможной.

Список сокращений и условных обозначений

Словарь терминов

TeX : Система компьютерной вёрстки, разработанная американским профессором информатики Дональдом Кнутом

панграмма : Короткий текст, использующий все или почти все буквы алфавита

Список литературы

1. *HORIBA-MIRA-Ltd*. MISRA C [Electronic Resource] / HORIBA-MIRA-Ltd. — URL: https://www.misra.org.uk (visited on 01/03/2021).
2. *Perforce-Software-Inc*. Rogue Wave Software, Klocwork [Electronic Resource] / Perforce-Software-Inc. — URL: https://www.perforce.com/products/klocwork (visited on 01/03/2021).
3. *GrammaTech-Inc*. GrammaTech, CodeSonar [Electronic Resource] / GrammaTech-Inc. — URL: https://www.grammatech.com/codesonar-cc (visited on 01/03/2021).
4. *The-MathWorks-Inc*. Mathworks, Polyspace Bug Finder [Electronic Resource] / The-MathWorks-Inc. — URL: https://www.mathworks.com/products/polyspace-bug-finder.html (visited on 01/03/2021).
5. *Synopsys-Inc*. Synopsys, Coverity [Electronic Resource] / Synopsys-Inc. — URL: https://scan.coverity.com (visited on 01/03/2021).
6. *AbsInt*. AbsInt Angewandte Informatik GmbH, Astree [Electronic Resource] / AbsInt. — URL: https://www.absint.com/astree/index.htm (visited on 01/03/2021).
7. *FRAMA-C*. The Frama-C platform web site, Frama-C [Electronic Resource] / FRAMA-C. — URL: https://frama-c.com (visited on 01/03/2021).
8. *The-MathWorks-Inc*. Mathworks, Polyspace Code Prover [Electronic Resource] / The-MathWorks-Inc. — URL: https://www.mathworks.com/products/polyspace-code-prover.html (visited on 01/03/2021).
9. *AdaCore*. CodePeer [Electronic Resource] / AdaCore. — URL: https://www.adacore.com/codepeer (visited on 01/03/2021).

Список рисунков

Список таблиц

Приложение А

Примеры вставки листингов программного кода