**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Специальность Компьютерная безопасность**

КРАГЕЛЬ АЛИНА ОЛЕГОВНА

**ВЕРИФИКАЦИЯ ПРОГРАММ. МЕТОДЫ ВЕРИФИКАЦИИ**

Реферат

Студента 1 курса 10 группы

Преподаватель

Конах Валентина Владимировна

Старший преподаватель

Кафедра ИСУ

Минск, 2019

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 2](#_Toc25656214)

[ГЛАВА 1. ПОНЯТИЕ ВЕРИФИКАЦИИ 3](#_Toc25656215)

[ГЛАВА 2. МЕТОД ИНДУКТИВНЫХ УТВЕРЖДЕНИЙ ФЛОЙДА 5](#_Toc25656217)

[ГЛАВА 3. МЕТОД АКСИОМАТИЧЕСКОЙ СЕМАНТИКИ ХОАРА 9](#_Toc25656219)

[ГЛАВА 4. ДОКАЗАТЕЛЬНОЕ ПРОГРАММИРОВАНИЕ 11](#_Toc25656221)

[ГЛАВА 5. АВТОМАТИЧЕСКОЕ ДОКАЗАТЕЛЬСТВО ТЕОРЕМ 14](#_Toc25656223)

[ГЛАВА 6. ПРОВЕРКА МОДЕЛЕЙ 16](#_Toc25656225)

[ГЛАВА 7. СИМВОЛЬНОЕ ВЫПОЛНЕНИЕ 19](#_Toc25656227)

[ГЛАВА 8. АБСТРАКТНАЯ ИНТЕРПРЕТАЦИЯ 22](#_Toc25656229)

[ЗАКЛЮЧЕНИЕ 25](#_Toc25656231)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 26](#_Toc25656232)

ВВЕДЕНИЕ

Качество программного обеспечения является постоянным объектом заботы программной инженерии и обсуждается во многих областях знаний. Пользовательские ожидания и конкурентная борьба существенно затрудняют возможность выпускать качественное программное обеспечение (ПО) в приемлемые сроки. Разработка программных средств (ПС) успешна, если в ней за запланированные время и стоимость качественно материализована затребованная функциональность. Однако часто возникают проблемы, приводящие к снижению качества разрабатываемого программного продукта и «неуспешности» проекта в целом. Причинами таких проблем является довольно обширный перечень факторов, начиная с недостаточно эффективного использования бюджета и заканчивая ошибками в оценках трудоемкостей и сроков работ. Однако пользователь выбирает самое лучшее приложение или программное обеспечение без оглядки на причины того или иного «бага» или той или иной неточности исполнения, так как главное – это результат, то есть качественное приложение, работающее в соответствии с требованиями и пользовательскими запросами.

Как контролировать качество системы? Как точно узнать, что программа делает именно то, что нужно, и ничего другого? Как определить, что она достаточно надежна, переносима, удобна в использовании? Ответами на эти вопросы можно назвать установленные разработчиками программного обеспечения специальные процессы, которые и проверяют стандарты и качество программных продуктов. Эти процессы – верификация и валидация.

Верификация обозначает проверку того, что программное обеспечение разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.

Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Эффективность верификации и валидации, как и эффективность разработки программного обеспечения в целом, зависит от полноты и корректности формулировки требований к программному продукту.

Задача данной работы рассмотреть вопросы верификации, то есть формальные доказательства правильности программ.

*Цель реферата* – изучить понятие верификации и её основные методы.

*Задача реферата:* раскрыть понятие верификации и ее основных методов, дать характеристику методам верификации и проанализировать их работу.

ГЛАВА 1.

ПОНЯТИЕ ВЕРИФИКАЦИИ

Верификация – это процесс определения, выполняют ли программные средства и их компоненты требования, наложенные на них в последовательных этапах жизненного цикла разрабатываемой программной системы.

Основная цель верификации состоит в подтверждении того, что программное обеспечение соответствует требованиям. Дополнительной целью является выявление и регистрация дефектов и ошибок, которые внесены во время разработки или модификации программы.

Верификация является неотъемлемой частью работ при коллективной разработке программных систем. При этом в задачи верификации включается контроль результатов одних разработчиков при передаче их в качестве исходных данных другим разработчикам.

Для повышения эффективности использования человеческих ресурсов при разработке, верификация должна быть тесно интегрирована с процессами проектирования, разработки и сопровождения программной системы.

Заранее следует разграничить понятия верификации и отладки. Оба этих процесса направлены на уменьшение ошибок в конечном программном продукте, однако отладка – процесс, направленный на локализацию и устранение ошибок в системе, а верификация – процесс, направленный на демонстрацию наличия ошибок и условий их возникновения.

Кроме того, верификация, в отличие от отладки – контролируемый и управляемый процесс. Верификация включает в себя анализ причин возникновения ошибок и последствий, которые вызовет их исправление, планирование процессов поиска ошибок и их исправления, оценку полученных результатов. Все это позволяет говорить о верификации, как о процессе обеспечения заранее заданного уровня качества создаваемой программной системы.

Формальная верификация может использоваться для проверки таких систем, как программное обеспечение, представленное в виде исходных текстов, [криптографические протоколы](http://wikiredia.ru/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB), [комбинаторные логические схемы](http://wikiredia.ru/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0), цифровые схемы с внутренней памятью. Верификации подвергаются:

* тесты, тестовые процедуры и входные наборы данных.
* компоненты системы и их интерфейсы (программные, технические и информационные) и взаимодействия объектов (протоколы, сообщения) в распределенных средах;
* описание доступа к БД, средства защиты от несанкционированного доступа к данным разных пользователей;
* документация на систему.

Существует ряд методов верификации программ. В работе собраны такие методы, как:

* Метод аксиоматической семантики Хоара
* Метод индуктивных утверждений Флойда
* Доказательное программирование
* Проверка моделей
* Символьное выполнение
* Абстрактная интерпретаци

ГЛАВА 2.

МЕТОД ИНДУКТИВНЫХ УТВЕРЖДЕНИЙ ФЛОЙДА

Метод Флойда основан на определении условий для входных и выходных данных и в выборе контрольных точек в доказываемой программе так, чтобы путь прохождения по программе пересекал хотя бы одну контрольную точку. Для этих точек формулируются утверждения о состоянии и значениях переменных в них (для циклов эти утверждения должны быть истинными при каждом прохождении цикла-инварианта).

Каждая точка рассматривается для индуктивного утверждения того, что формула остается истинной при возвращении в эту точку программы и зависит не только от входных и выходных данных, но и от значений промежуточных переменных. На основе индуктивных утверждений и условий на аргументы создаются утверждения с условиями проверки правильности программы в отдельных ее точках. Для каждого пути программы между двумя точками устанавливается проверка на соответствие условий правильности и определяется истинность этих условий при успешном завершении программы на данных, удовлетворяющих входным условиям.

Формирование таких утверждений - довольно сложная задача, особенно для программ с высокой степенью параллельности и взаимодействия с пользователем. Кроме того, трудно проверить достаточность и правильность самих утверждений.

Доказательство корректности применялось для уже написанных программ и тех, которые разрабатываются методом последовательной декомпозиции задачи на подзадачи, для каждой из них формулируются утверждения с учетом условий ввода и вывода и точек программы, расположенными между входными и выходными утверждениями. Суть доказательства истинности выполнения условий и утверждений относительно заданной программы и составляет основу

Для изучения этого метода программа снабжается утверждениями о свойствах её переменных в некоторых точках:

a)      Входные переменные не меняются в процессе исполнения программ;

b)      Описываются состояния переменных в промежуточных точках;

c)      Выходные переменные описываются с помощью отношениями между переменными после завершения программы.

Верификация состоит в последовательной демонстрации того, что из входных переменных и преобразований, выполненных на первом шаге, следует истинность утверждения, сформированного в следующей промежуточной точке.

Для верификации программ необходимо три языка:

* Язык записи текстов программ;
* Язык формулировки условий верификации;
* Язык формирования и доказательства корректности.

Так как эти языки различаются в значительной степени, то это обстоятельство является одним из применения верификации.

Доказательство корректности имеет следующие преимущества:

1.      Представляет собой чёткий формализованный процесс.

2.      Требует анализа. Процесс доказательства корректности  дает возможность рассматривать части программ, которые в противном случае анализируются лишь случайно.

3.      Проясняет промежуточные результаты вычислений. Выписывание выражений заставляет программиста четко сформировать свои предположения о результатах вычислений в выбранных точках программы.

4.      Выявляет зависимости. В процессе доказательства программа начинает понимать, какие предположения о входных данных не явно испытываются в различных частях программы.

Недостатки метода:

1.      Сложность; даже для небольших простых программ выкладки очень сложны, что может привести к ошибкам.

2.      Ошибки; из-за сложности метода легко допустить ошибки и при формировании доказываемых утверждений и при доказательстве.

3.      Трудности работы с массивами.

4.      Отсутствие мощного математического аппарата.

5.      Высокая трудоёмкость.  Для проверки программы требуется большие затраты труда, чем для её написания (в 2 – 6 раз).

6.      Отсутствие выразительности. Часто нелегко сформировать выгодное утверждение для того, что интуитивно представляется  очень простым вычислением.

7.      Трудность понимания.

8.      Необходимость  обучения. Для применения этого метода требуется длительное обучение и тренировка.

Пусть дана программа S, для неё определены два предиката: предусловие I и постусловие R. Метод Флойда доказательства частичной корректности программы состоит в следующем:

1. Представить программу S в виде блок-схемы (см. рисунок 1).

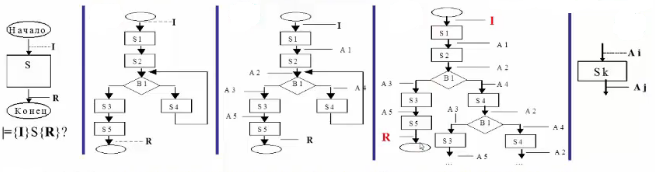


Рисунок 1

1. На каждой дуге блок-схемы задать контрольную точку. Каждой контрольной точке сопоставить предикат так, что начальной контрольной точке сопоставляется предусловие I, конечной – постусловие R. Остальным контрольным точкам сопоставляются произвольные предикаты Ai (условия-инварианты).
2. У каждого блока Sk блок-схемы программы на его входной дуге и выходной дуге существует пара контрольных точек, которым сопоставлены предикаты Ai и Aj.
3. Для каждого блока Sk программы доказывается теорема: если утверждение Ai истинно до начала работы программы Sk, то по завершении оператора, находящегося в этом блоке, утверждение Aj будет истинным.
4. Теорема. Пусть для каждой тройки доказана теорема | = {Ai} Sk {Aj}. Тогда, если истинно предусловие I, то каждое утверждение Ai, встретившееся на пути вычислений (в том числе и R), истинно. Данная теорема доказывается по индукции.
5. Следствием теоремы является следующее утверждение: если программа завершается (т.е. если R когда-нибудь, встретится на вычислении) то, если I истинно на входе в S, то утверждение | = {Ai} Sk {Aj} выполняется. Любое завершившееся вычисление проходит контрольные точки, на которых, согласно теореме, соответствующие утверждения выполняются. Но тогда выполняется и утверждение R на выходной контрольной точке.

Однако метод Флойда не гарантирует завершения программы. Этот метод гарантирует только, если программа завершается, т.е. вычисление программы проходит заключительную контрольную точку, то в этом случае постусловие R будет истинно.

В случае, если для какого-то блока Sk нельзя доказать справедливость Хоаровской тройки | = {Ai} Sk {Aj}, то или программа неверна, или неверно построены утверждения на стрелках программы.

Если доказаны все теоремы | = {Ai} Sk {Aj}, то, как уже говорилось ранее, программа может не завершится (некоторые вычисления могут длиться бесконечно, при этом каждое утверждения на каждой дуге программы может оставаться истинным).

ГЛАВА 3.

МЕТОД АКСИОМАТИЧЕСКОЙ СЕМАНТИКИ ХОАРА

Метод Хоара - это усовершенствованный метод Флойда, основанный на аксиоматическом описании семантики языка программирования исходных программ. Каждая аксиома описывает изменение значений переменных с помощью операторов этого языка. Формализация операторов перехода и вызовов процедур обеспечивается с помощью правил вывода, содержащих индуктивные высказывания для каждой точки и функции исходной программы.

Система правил вывода дополняется механизмом переименования глобальных переменных, условиями на аргументы и результаты, а также на правильность задания данных программы. Оператор перехода трактуется как выход из циклов и аварийных ситуаций.

Описание с помощью системы правил утверждений - громоздкое и отличается неполнотой, поскольку все правила предусмотреть невозможно. Данный метод проверялся экспериментально на множестве программ без применения средств автоматизации из-за их отсутствия.

Важным моментом в понимании метода Хоара является логика Хоара. Логика Хоара – формальная система с набором логических правил, предназначенных для доказательства корректности компьютерных программ. Была предложена в 1969 году английским учёным в области информатики и математической логики Хоаром, позже развита самим Хоаром и другими исследователями. Первоначальная идея была предложена в работе Флойда, который опубликовал похожую систему в применении к блок-схемам.

Основной характеристикой логики Хоара является тройка вида {P} S {Q}, где P и Q – предикаты, а S – программа языка программирования. Тройка описывает, как выполнение фрагмента кода изменяет состояние вычисления.

В логике Хоара есть [аксиомы](https://ru.wikipedia.org/wiki/%D0%90%D0%BA%D1%81%D0%B8%D0%BE%D0%BC%D0%B0) и [правила вывода](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B0%D0%B2%D0%B8%D0%BB%D0%B0_%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4%D0%B0) для всех конструкций простого [императивного языка программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). В дополнение к этим конструкциям, описанным в оригинальной работе, Хоаром и другими исследователями были разработаны правила и для остальных конструкций: [одновременного выполнения](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B4%D0%BD%D0%BE%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1), [вызова процедуры](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), [перехода](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%B0) и [указателя](https://ru.wikipedia.org/wiki/%D0%A3%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)).

Основная идея Хоара — дать для каждой конструкции императивного языка пред- и постусловие, записанное в виде логической формулы. Поэтому и возникает в названии «тройка» — предусловие, конструкция языка, постусловие.

* Ясно, что для пустого оператора пред- и постусловия совпадают.
* Для оператора присваивания в постусловии кроме предусловия должен учитываться факт, что значение переменной стало другим.
* Для составного оператора имеем цепочку пред- и постусловий. В результате для составного оператора можно оставить первое предусловие и последнее постусловие.
* Правило вывода говорит, что можно усилить пред- и ослабить постусловие, если нам это понадобится. Нет смысла сохранять всю программу, какое-то утверждение которой не помогает решить поставленную задачу.
* Оператор ветвления или просто if. Его условно можно разбить на две ветки: then и else. Если к предусловию добавить истинность логического условия (то, что стоит под if), то после выполнения ветки then должно следовать постусловие. Аналогично, если к предусловию добавить отрицание логического условия (то, что стоит под if), то после выполнения ветки else должно следовать постусловие.
* Оператор цикла. Это самое нетривиальное и сложное, поскольку цикл может выполняться много раз и даже не закончиться. Чтобы решить проблему возможного многократного повтора тела цикла вводят инвариант. Инвариант цикла — это то, что истинно перед его выполнением, истинно после каждого выполнения тела цикла и, следовательно, истинно и после его окончания. Предусловие для оператора цикла — это просто его инвариант. Если истинно условие продолжения цикла (то, что стоит под while), то после выполнения тела цикла должна следовать истинность инварианта. В результате, после окончания цикла имеем в качестве постусловия истинность инварианта и отрицание условия продолжения цикла.
* Оператора цикла с полной корректностью. Для этого к предыдущему пункту добавляют ограничивающую функцию, с помощью которой легко доказать, что цикл будет выполняться ограниченное число раз. На неё накладывают условия, что она всегда >=0, строго убывает после каждого выполнения тела цикла и в точности = 0, когда цикл заканчивается.

Правильно работающую программу можно написать очень многими способами, и во многих случаях она будет эффективной. Эта неоднозначность усложняет программирование. Для этого вводят стиль. Но этого оказывается мало. Для многих программ (например, связанных косвенно с жизнью человека) нужно доказать и их корректность. Оказалось, что доказательство корректности делает программу дороже на порядок (примерно в 10 раз).

В стандартной логике Хоара может быть доказана только частичная корректность, так как завершение программы нужно доказывать отдельно.

ГЛАВА 4.

ДОКАЗАТЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Доказательное программирование — это составление программ, которое проводиться с параллельным доказательством их правильности.

Для заключений о наличии ошибок в алгоритме или в программе достаточно указать тест, при котором произойдет сбой, отказ или будут получены неправильные результаты.

Поиск и исправление ошибок в программах обычно проводятся на ЭВМ.

Для утверждений правильности программ необходимо продемонстрировать, что правильные результаты будут получаться для всех возможных допустимых данных. Такие утверждения могут быть доказаны только путем исчерпывающего анализа результатов выполнения программ при любых допустимых данных.

Существуют два подхода к проверке программ — прагматический и доказательный.

При прагматическом подходе проверка программ выполняется на ЭВМ путем тестирования, т.е. проверкой программ на ЭВМ с помощью некоторого набора тестов. Ясно, что тестирование не дает гарантий правильности выполнения программ на всех допустимых данных. Следовательно, тестирование в общем случае не может дать и не дает полных гарантий отсутствия ошибок в программах. Поэтому прагматический подход чреват созданием программ, содержащих ошибки даже после «завершения» отладки, что и наблюдается практически во всех больших программах для ЭВМ.

Ниже приведена иллюстрация принципов тестирования алгоритма и программы вычисления максимума из трех чисел: а, b, с.

cin >> a >> b >> c;

if (a > b) then

max = a;

else

if (b > c) then

max = b;

else

if (c > a) then

max = c;

cout << max;

Запуск этой программы на ЭВМ можно проверить на следующих данных:

Tecт1  1 1 2 max = 2

Тест2  1 2 3 max = 3

Тест3 3 2 1 max = 3

Все три результата правильные. Отладку программы после запуска этих примеров можно было бы считать завершенной. Однако есть контрпример:

Контрпример1 2 1 3 max = 2

Но этот результат - неправильный. Следовательно, алгоритм и программа содержат ошибки, точное число которых неизвестно.

При доказательном подходе разработка алгоритмов и программ предполагает составление спецификаций и доказательство их правильности. Процесс разработки программ завершен после проверки их на ЭВМ и предоставления доказательств отсутствия ошибок.

Доказательства правильности алгоритмов и программ, равно как и любые другие доказательства, строятся на основе суждений и рас­суждений. В данном случае суждения и рассуждения касаются результатов выполнения алгоритмов и программ с теми или иными данными.

Конструктивно, доказательства правильности алгоритмов и про­грамм строятся на суждениях и утверждениях о результатах выпол­нения каждого из составляющих их действий и операций в соответ­ствии с порядком их выполнения.

Техника анализа и доказательства правильности алгоритмов и программ во многом совпадает с техникой доказательства любых других утверждений и состоит в применении следующих четырех приемов:

* разбор случаев;
* подбор контрпримеров;
* выделение лемм;
* индуктивный вывод.

Разбор случаев применяется для анализа результатов выполнения конструкций альтернативного выбора. В качестве примера можно рассмотреть анализ приведенного выше алгоритма «выбора» максимума трех чисел, содержащего выбор альтернатив.

Для опровержения общего утверждения достаточно указать хотя бы один контрпример.В данном случае утверждение о правильности алгоритма гласило бы: для любых значений переменных а, b, с конечным было бы значение mах (а, b, с).

Контрпримером в данном случае будут значения: а = 2, b = 1, с = 3. Для этих данных по определению mах = 3, а по результатам выполнения алгоритма max = 2. Следовательно, в алгоритме содержится ошибка.

Однако оказывается, что это не единственная ошибка. Более тон­кие ошибки вскрывает второй контрпример: а = 1, b = 1, c = 1. Для этих данных в алгоритме вовсе не определен результат вычислений, и конечный результат выполнения программы будет непред­сказуем.

Правильное решение этой задачи можно получить применением систематических методов, составив постановку и описание метода решения.

Правильность алгоритма может быть доказана двумя способами. Первый способ заключается в анализе правильности при по­строении алгоритмов. Нужно показать, что алгоритм является корректной реализацией метода решения, и доказать, что метод – правильный. Для рассмотренного алгоритма это доказательство изложено выше. Привлечение для создания алгоритмов известных методов реше­ния, для которых доказана их правильность, позволяет существенно упростить обоснование правильности программ. При этом центр тяжести проблем смещается к созданию и обоснованию гарантиро­ванно правильных методов решения задач.

Второй способ заключается в анализе правильности после построения алгоритмов и является исчерпывающим анализом результатов выпол­нения алгоритма на соответствие постановке решаемых задач для любых допустимых данных. Это доказательство путем исчерпыва­ющего анализа результатов выполнения алгоритмов и программ.

ГЛАВА 5.

АВТОМАТИЧЕСКОЕ ДОКАЗАТЕЛЬСТВО ТЕОРЕМ

Автоматическое доказательство (англ. Automated Theorem Proving, ATP, а также Automated deduction) – это доказательство, которое реализовано программно. В его основе лежит аппарат математической логики и используются идеи, теории искусственного интеллекта. Процесс доказательства основывается на логике высказываний и логике предикатов.

Из-за неразрешимости даже достаточно простых теорий практическое применение имеет лишь полуавтоматическое человеко-машинное доказательство. В добавок ко всему остальному после полной автоматизации доказательство называют уже вычислением. Полностью автоматической может быть лишь проверка доказательства теорий посложнее (если его для этого подготовить).

Можно даже сказать, что автоматическое доказательство теорем – одна из старейших частей искусственного интеллекта. Благодаря исследованиям в области автоматического доказательства теорем были формализованы алгоритмы поиска и разработаны языки формальных представлений, такие как исчисление предикатов и логический язык программирования Prolog.

Привлекательность автоматического доказательства теорем основана на строгости и общности логики. В формальной системе логика располагает к автоматизации. Разнообразные проблемы можно попытаться решить, представив описание задачи и относящуюся к ней информацию в виде логических аксиом и рассматривая различные случаи задачи как теоремы, которые нужно доказать. Этот принцип лежит в основе автоматического доказательства теорем и систем математических обоснований.

К сожалению, в ранних пробах написать программу для автоматического доказательства теорем не удалось разработать систему, которая бы единообразно решала сложные задачи. Это было обусловлено способностью любой относительно сложной логической системой сгенерировать бесконечное количество доказуемых теорем: без мощных методик и эвристик, которые бы направляли поиск, программы доказывали большие количества не относящихся к делу теорем, пока не натыкались на нужную. Из-за этой неэффективности многие утверждают, что чисто формальные синтаксические методы управления поиском в принципе не способны справиться с такими большими пространствами, и единственная альтернатива этому – положиться на неформальные, специально подобранные к случаю стратегии, как это, похоже, делают люди. Это один из подходов, лежащих в основе экспертных систем, и он оказался достаточно плодотворным.

Все же привлекательность рассуждений, основанных на формальной логике, слишком сильна, чтобы ее игнорировать. Многие важные проблемы, такие как проектирование и проверка логических цепей, проверка корректности компьютерных программ и управление сложными системами, по-видимому, поддаются такому подходу. Вдобавок исследователям автоматического доказательства теорем удалось разработать мощные эвристики, основанные на оценке синтаксической формы логического выражения, которые в результате понижают сложность пространства поиска, не прибегая к используемым людьми методам.

Еще одна причина такого неувядающего интереса к автоматическому доказательству теорем - понимание, что системе не обязательно решать особо сложные проблемы без человеческого вмешательства. Многие современные программы доказательств работают как умные помощники: люди разбивают задачи на подзадачи и продумывают эвристики для перебора в пространстве возможных обоснований; затем программа для автоматического доказательства теорем решает более простые задачи доказательства лемм, проверки менее существенных предположений и дополняет формальные аспекты доказательства, очерченного человеком.

В настоящее время автоматическое доказательство теорем в промышленности применяется в основном при разработке и верификации интегральных схем и программного обеспечения. После того, как была обнаружена ошибка деления в процессорах Пентиум, сложные модули операций с плавающей запятой современных микропроцессоров разрабатываются с особой тщательностью. [В](https://www.turkaramamotoru.com/ru/%D0%92-2940.html) новых процессорах AMD, Intel и других фирм автоматическое доказательство теорем используется для проверки того, что деление и другие операции выполняются корректно.

Корпорация Microsoft использует автоматическое доказательство теорем Z3 для верификации кода операционной системы Windows 7 и других программных продуктов.

ГЛАВА 6.

ПРОВЕРКА МОДЕЛЕЙ

В настоящее время происходит качественный прорыв в области верификации программных систем и аппаратных дискретных систем. Новая технология основывается на изящных формальных методах. Ее название – «model checking».

Model checking (проверка модели) – это методы и алгоритмы верификации аппаратуры и программ, которые разработаны от теоретических изыcканий до индуcтриальной технологии (т.е. реально используемых на практике).

Последователи данного подхода верификации не стремятся впиcать cиcтему в рамки теории, а вместо этого строят модель системы, которую можно раccматривать как машину или автомат. Любое требование к cиcтеме проверяетcя для каждого возможного cоcтояния автомата.

Модельных подходов извеcтно, по меньшей мере, неcколько дюжин — конечные автоматы, cети Петри, временные автоматы, логическое опиcание и т.п.

Модельный подход поддерживает не только полную, но и чаcтичную верификацию, которая может быть направлена на проверку только одного небольшого свойства, абстрагировавшись от менее важных деталей системы. Иными словами, для проведения верификации не обязательно добиваться формализации всех без исключения требований спецификации. В отличие от тестирования и использования симуляторов, в модельном подходе не существует такого понятия, как вероятность обнаружения ошибки: если ошибка есть, она будет обнаружена за конечное время.

В том случае, когда свойство оказывается нарушенным, в виде контрпримера предоставляется диагностирующая информация.

Процесс проверки моделей не требует ни ручного управления со стороны пользователя, ни высокого уровня профессионализма. Имея модель, можно автоматически проверять на ней необходимые свойства. Процесс проверки интегрируется в стандартный цикл проектирования, позволяя, как показывает практика, уменьшить время создания приложений с учетом проведения рефакторинга программного кода.

Однако у модельного подхода есть и слабые стороны. Верификация осуществляется по модели, а не по реальной системе, поэтому ценность полученного результата напрямую зависит от корректности модели, что требует высокого уровня подготовки персонала, создающего модели программ.

Преобладает ориентация на приложения, в которых главную роль играет поток управления, а не поток данных, так как данные имеют тенденцию принимать значения из бесконечных множеств. Такая ориентация уменьшает возможности универсального применения, однако обычно это не столь существенно при разработке больших аппаратно-программных комплексов, поскольку практически все существующие виды модульных приложений, из которых складываются подобные комплексы, можно либо в том или ином виде привести к модели «потока управления», либо корректировать методику тестирования для каждого конкретного модуля. Модельный подход не может эффективно применяться без точных алгоритмов принятия решений. Нет гарантий полноты: проверяются только те свойства, которые указаны явно.

Построение моделей и формулировка требований требуют высокого уровня знаний и умения их применять. Результаты могут вводить в заблуждение (верификатор — тоже программа и тоже может ошибаться, модель может содержать ошибку и т.п.; правда, основные процедуры проверки моделей формально доказаны с помощью пакетов автоматического доказательства теорем). Нет верификаторов, поддерживающих обобщения, например, нельзя проверить систему, если в ней не зафиксировать число сущностей.

Примеры успешного применения модельного подхода можно обнаружить, изучая процесс разработки сложных систем, оперирующих большими потоками данных: СУБД, комплексы потоковой обработки речевой и текстовой информации, системы обеспечения информационной безопасности. Модельный подход к верификации программного обеспечения позволяет, при правильном разбиении всего комплекса, при проектировании и разработке модулей и более атомарных составляющих выявлять логические ошибки еще на этапе проектирования. Так, при разработке программного обеспечения потоковой обработки растровых изображений в рамках модельного подхода была сформирована модель для верификации менеджера заданий для потоковой обработки и обработчиков атомарных заданий, позволившая выявить ошибки в проектировании протоколов взаимодействия модулей комплекса и алгоритме определения обработчика атомарного задания. Данная модель основана на использовании сетей Петри и сопутствующих алгоритмов.

В качестве модели обычно используется так называемая модель Крипке, которая формально задаётся следующим образом: M = (S, S0, R, L) , где S — множество состояний, S0 — множество начальных состояний, R c S\*S — отношение переходов, L: S → 2AP — функция разметки.

Обычно спецификации задаются на языке формальной логики. Для спецификации аппаратного и программного обеспечения, как правило, применяют темпоральную логику — специальный язык, позволяющий описывать поведение системы во времени.

Важным вопросом спецификации является полнота. Метод проверки на модели позволяет убедиться, что модель проектируемой системы соответствует заданной спецификации, однако определить, охватывает ли заданная спецификация все свойства, которым должна удовлетворять система, невозможно.

Основная трудность, которую приходится преодолевать в ходе проверки на модели, связана с эффектом комбинаторного взрыва в пространстве состояний. Эта проблема возникает в системах, состоящих из многих компонентов, взаимодействующих друг с другом, а также в системах, обладающих структурами данных, способных принимать большое число значений.

# 

ГЛАВА 7.

СИМВОЛЬНОЕ ВЫПОЛНЕНИЕ

В случае, когда техника логического доказательства подтверждает, что спроектированные компоненты программы являются правильными, они необязательно выполняются. Одна из таких техник – символьное выполнение – включает моделирование выполнения кода, используя символы вместо переменных данных. Тестовая программа рассматривается как имеющая детерминированное входное состояние при вводе данных и условий. Благодаря этому каждая строка кода выполняется, а данная техника проверяет, изменилось ли состояние. Каждое измененное состояние сохраняется, и выполняемая программа рассматривается как серия изменений состояний. Последнее состояние каждой части будет выходным состоянием и программа будет правильной.

Описание доказательства может оказаться длиннее написанного фрагмента программы, и нет уверенности, что это описание не содержит ошибок. Данная стратегия имеет преимущество перед логическим доказательством теорем, поскольку полагается на трассирование изменяющихся условий операторов программы. Главными элементами доказательства корректности программы являются фрагмент программы, входные и выходные значения переменных, а также процесс слежение за тем, как этот фрагмент программа будет выполняться.

Первоначальные варианты систем автоматизации тестирования, основанных на символьном выполнении, генерировали системы условий, используя статический анализ исходного текста программы. В статическом варианте символьное выполнение подразумевает проход вдоль заданного пути программы с подстановкой вместо конкретных значений переменных выражений, связывающих их значения со значениями входных параметров программы. Статическое символьное выполнение сталкивается со следующими серьезными проблемами:

* если программа использует указатели или массивы, символьное выполнение может давать неверные результаты, поскольку в общем случае невозможно узнать, на что указывает данный указатель;
* программа, работающая с памятью нестандартным образом (например, читающая переменные частями), в общем случае не поддается символьному выполнению;
* весьма сложно учесть при символьном выполнении данные, возвращаемые стандартными подпрограммами и системными вызовами. Для этого требуется подвергать символьному выполнению эти подпрограммы, что не всегда возможно
* вызовы подпрограмм требуют вложенного символьного выполнения и таким образом приводят к разрастанию полученных условий;
* получаемые системы условий сложны и далеко не всегда поддаются решению.

На этапе просмотра кода невозможно определить точное поведение программы при ее выполнении, кроме того, не всегда для анализа доступен весь исходный код. Структурное тестирование подразумевает выбор множества тестовых воздействий на основе граф-модели программы с целью достижения максимального тестового покрытия. Динамическое символьное выполнение. В настоящее время все чаще реализуются подходы к генерации тестовых воздействий, сочетающие символьное выполнение и мониторинг выполнения программы. Общая схема тестирования программы на основе динамического символьного выполнения такова.

1. Первый запуск программы производится с произвольным тестовым воздействием.
2. При запуске программы производится одновременно как конкретное выполнение, так и символьное выполнение. Таким образом, известны как конкретные выражения для переменных, так и выражения (зависимости от входных переменных) вдоль данного пути. Для символьного выполнения в программу добавляется соответствующий инструментальный код
3. В ходе символьного выполнения определяется система условий прохождения выбранного пути.
4. Система условий решается таким образом, чтобы найти нарушающее ее тестовое воздействие, которое приведет к выполнению программы по некоторому альтернативному пути. Данное тестовое воздействие используется при следующем выполнении программы.
5. При достижении желаемого тестового покрытия процесс прерывается. Покажем, что такой подход решает приведенные ранее проблемы:
   * поскольку известны конкретные значения переменных, всегда можно узнать цель любого указателя и избавиться от неоднозначности при построении выражений;
   * доступ к фрагментам переменных является либо ошибкой, либо блочным вводом-выводом, либо следствием плохого стиля программирования. Решение может быть достигнуто как выдачей предупреждений, т. к. и определением специальных правил для такого доступа (и использованием конкретных значений вместо символьных);
   * подпрограммы могут трактоваться так же, как безусловные переходы. Отслеживание значений параметров может производиться с помощью отладочной информации, генерируемой компилятором;
   * в тех случаях, когда решение в символах не может быть найдено, можно использовать известные конкретные значения вместо символьных

ГЛАВА 8.

АБСТРАКТНАЯ ИНТЕРПРЕТАЦИЯ

Абстрактная интерпретация объединяет обычные статические методы анализа и динамическое тестирование путем проверки динамических свойств программ во время компиляции. Не выполняя саму программу, абстрактная интерпретация исследует все ее возможные поведения (то есть все возможные комбинации значений) за единственный проход, чтобы определить, как и при каких условиях может возникнуть сбой программы. Абстрактная интерпретация представляет собой развитый и мощный математический метод. Ее можно рассматривать как расширение методов компиляции, используемых программистами для того, чтобы до выполнения реальных тестов предсказать будущее поведение программы.

Абстрактная интерпретация опирается на широкую базу математических теорем, которые определяют правила для анализа сложных динамических систем, таких как программные приложения. Вместо проведения численного анализа каждого состояния программы, абстрактная интерпретация представляет эти состояния в более общем виде и задает правила работы с ними. Абстрактная интерпретация не только создает математическую абстракцию, но и интерпретирует эту абстракцию.

Для того чтобы получить математическую абстракцию состояний программы, абстрактная интерпретация тщательно анализирует все переменные кода. Значительные вычислительные мощности, необходимые для этого анализа, в прошлом отсутствовали. Абстрактная интерпретация, в сочетании с алгоритмами не экспоненциальной сложности и возросшей современной мощностью компьютеров, представляет собой практическое решение сложных проблем тестирования.

В случае применения для обнаружения ошибок выполнения абстрактная интерпретация выполняет полную верификацию всех операций, вызывающих риск, и автоматически выполняет диагностику состояния каждой операции («проверена», «не выполнена», «недоступна» или «не проверена»). Инженеры могут использовать абстрактную интерпретацию, чтобы получить результаты в момент компиляции, то есть на самой ранней стадии тестирования.

Применение абстрактной интерпретации

Для того, чтобы лучше понять, как работает абстрактная интерпретация, рассмотрим программу P, которая использует две переменные: X и Y. Она выполняет следующую операцию:

X=X/(X-Y)

Для проверки этой программы на наличие ошибок выполнения исследуем все возможные причины, вызывающие ошибку операции:

* X и Y могут не иметь начальных значений
* X-Y может иметь положительное или отрицательное переполнение
* X и Y могут быть равны и вызывать деление на нуль
* X/(X–Y) может иметь положительное или отрицательное переполнение

Поскольку любое из этих состояний может вызвать ошибку выполнения, на следующих шагах исследуется возможность деления на нуль.

Очевидный способ проверки деления на нуль заключается в переборе всех состояний и определения того, находятся ли они на красной линии. Этот подход применяется при обычном тестировании по методу «белого ящика», но он имеет существенные ограничения. Во-первых, количество возможных состояний в реальном приложении обычно очень велико из-за большого числа используемых переменных. На перебор всех возможных состояний уйдут годы, что делает полную проверку совершенно невозможной.

В отличие от «метода грубой силы», который заключается в переборе всех возможных значений, абстрактная интерпретация устанавливает обобщенные правила для анализа всего множества состояний. Она создает абстрактное представление программы, которое можно использовать для проверки свойств.

При выполнении абстрактной интерпретации указанные ниже элементы программы толкуются по-новому.

* Индекс в цикле for не является более целочисленной величиной, а представляет собой монотонную возрастающую дискретную функцию от нижнего до верхнего предела.
* Параметр, передаваемый в функцию, более не является переменной или константой, а представляет собой набор значений, которые можно использовать с целью ограничения локальных данных, используемых в функции.
* Любые глобальные совместно используемые данные могут быть в любой момент изменены в многозадачной программе, за исключением того случая, когда включены механизмы защиты, такие как блокировки памяти или критические секции.
* Указатель является типом данных, который может создавать связи между явными данными и приводить к побочным эффектам и скрытым одновременным обращениям к совместно используемым данным в многозадачных приложениях.
* Переменная имеет не только тип и диапазон значений, но и набор уравнений (включая соотношения, зависящие от потока управления), которые порождают ее.
* В итоге, ошибки при выполнении являются уравнениями, которые называются условиями правильности. Абстрактная интерпретация может их решить, используя уравнения, связывающие переменные друг с другом.

ЗАКЛЮЧЕНИЕ

В работе были исследованы основные формальные методы верификации программ. Раcсмотрены и описаны такие методы, как:

* Метод акcиоматической семантики Хоара
* Метод индуктивных утверждений Флойда
* Доказательное программирование
* Автоматичеcкое доказательство теорем
* Проверка моделей
* Cимвольное выполнение
* Абстрактная интерпретация

Было проведено исследование вышеперечисленных методов верификации, выявлены их доcтоинства, недоcтатки и определена область применения. В работе приведены последовательности проведения того или иного метода верификации, предложена алгебраическая модель программы как множества связанных алгебраических выражений, независимая от языка программирования и позволяющая поставить в соответствие тестовым путям в программе условия их прохождения в символьном виде с учетом операций над указателями.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ:

1. Г. Майерс. Искусство тестирования программ. Вашингтон. 1982.
2. Генельт А.Е. Учебно-методическое пособие по дисциплине «Управление качеством разработки ПО». Санкт-Петербург. 2007.
3. Кулямин В.В. Методы верификации программного обеспечения. Москва. 2008. 117 с.
4. Лаврищева Е.М. Методы и средства инженерии программного обеспечения. Москва. 2006. 300 с.
5. Основы управления качеством программных. Ульяновск. 2017. 85 с.
6. Синицын С. В., Налютин Н. Ю. Верификация программного обеспечения. Москва. 2008. 368 с.
7. Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. Пекин. 1983. 360 с.
8. <http://sorlik.blogspot.com>
9. <http://vit-prog.narod.ru/page/TRPP/section_4/subjec_4.2.htm>
10. <https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0_%D0%A5%D0%BE%D0%B0%D1%80%D0%B0>
11. <https://www.intuit.ru/studies/courses/2190/237/info>