**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ГОРНЫЙ УНИВЕРСИТЕТ**

**кафедра вм**

**Курсовик**

**“Программа сложной структуры с использованием меню”**

**ВЫПОЛНИЛ: Пикулин Е. Г.**

**принял: Солодовников А. Д.**

**© мОСКВА 1996 год**

**ОГЛАВЛЕНИЕ.**

**1. ВИДЫ КОНТРОЛЯ ПРОГРАММ**

**2. ЦЕЛИ, ПРИНЦИПЫ И ЭТАПЫ ТЕСТИРОВАНИЯ**

**3. СТРУКТУРНОЕ ТЕСТИРОВАНИЕ**

**4. СОВМЕСТНОЕ ТЕСТИРОВАНИЕ МОДУЛЕЙ**

**5. ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ**

**6. ТЕСТИРОВАНИЕ ПРОГРАММНОГО КОМПЛЕКСА В ЦЕЛОМ**

**7. ОТЛАДКА ПРОГРАММ**

ВИДЫ КОНТРОЛЯ ПРОГРАММ

Программный комплекс - это совокупность программных модулей, предназначенных для решения одной задачи и составляющих одно целое.

Основными разновидностями контроля программного обеспечения являются визуальный, статический и динамический.

Визуальный контроль - это проверка программ “ за столом “ , без использования компьютера. На первом этапе визуального контроля осуществляется чтение программы, причем особое внимание уделяется следующим ее элементам:

1. комментариям и их соответствию тексту программы ;
2. условиям в операторах условного выбора ( IF, CASE ) и цикла;
3. сложным логическим выражениям;
4. возможности незавершения итерационных циклов ( WHILE, REPEAT, LOOP ).

Второй этап визуального контроля - сквозной контроль программы

( ее ручная прокрутка на нескольких заранее подобранных простых тестах). Распространенное мнение , что более выгодным является перекладывание большей части работы по контролю программных средств на компьютере, ошибочно. Основной довод в пользу этого таков : при работе на компьютере главным образом совершенствуются навыки в использовании клавиатуры, в то время как программистская квалификация приобретается прежде всего за столом.

Статический контроль- это проверка программы по ее тексту (без выполнения) с помощью инструментальных средств. Наиболее известной формой статического контроля является синтаксический контроль программы с помощью компилятора , при котором проверяется соответствие текста программы синтаксическим правилам языка программирования.

Сообщения компилятора обычно делятся на несколько групп в зависимости от уровня тяжести нарушения синтаксиса языка программирования :

1. информационные сообщения и предупреждения , при обнаружении которых компилятор, как правило, строит корректный объектный код и дальнейшая работа с программой (компоновка, выполнение) возможна (тем не менее сообщения этой группы также должны тщательно анализироваться, так как их появление также может свидетельствовать об ошибке в программе - например, из-за неверного понимания синтаксиса языка);
2. сообщения об ошибках, при обнаружении которых компилятор пытается их исправить и строит объектный код, но его корректность маловероятна и дальнейшая работа с ним скорее всего не возможна;
3. сообщения о серьезных ошибках , при наличии которых построенный компилятором объектный код заведомо некорректен и его дальнейшее использование невозможно;
4. сообщения об ошибках , обнаружение которых привело к прекращению синтаксического контроля и построения объектного кода .

Однако, практически любой компилятор пропускает некоторые виды синтаксических ошибок. Место обнаружения ошибки может находиться далеко по тексту программы от места истинной ошибки, а текст сообщения компилятора может не указывать на истинную причину ошибки. Одна синтаксическая ошибка может повлечь за собой генерацию компилятором нескольких сообщений об ошибках (например, ошибка в описании переменной приводит к появлению сообщения об ошибке в каждом операторе программы, использующем эту переменную).

Второй формой синтаксического контроля может быть контроль структурированности программ, то есть проверка выполнения соглашений и ограничений структурного программирования. Примером подобной проверки может быть выявление в тексте программы ситуаций, когда цикл образуется с помощью оператора безусловного перехода (использования оператора GOTO для перехода вверх по тексту программы ). Для проведения контроля структурированности могут быть созданы специальные инструментальные средства, а при их отсутствии эта форма статического контроля может совмещаться с визуальным контролем .

Третья форма статического контроля - контроль правдоподобия программы, то есть выявление в ее тексте конструкций, которые хотя и синтаксически корректны, но скорее всего содержат ошибку или свидетельствуют о ней. Основные неправдоподобные ситуации :

1. использование в программе неинициализированных переменных (то есть переменных, не получивших начального значения) ;
2. наличие в программе описаний элементов, переменных, процедур, меток, файлов, в дальнейшем не используемых в ее тексте;
3. наличие в тексте программы фрагментов, никогда не выполняющихся;
4. наличие в тексте программы переменных, ни разу не используемых для чтения после присваивая им значений;
5. наличие в тексте программы заведомо бесконечных циклов ;

Даже если присутствие в тексте программы неправдоподобных конструкций не приводит к ее неправильной работе, исправление этого фрагмента повысит ясность и эффективность программы, т. е. благотворно скажется на ее качестве.

Для возможности проведения контроля правдоподобия в полном объеме также должны быть созданы специальные инструментальные средства, хотя ряд возможностей по контролю правдоподобия имеется в существующих отладочных и обычных компиляторах.

Следует отметить, что создание инструментальных средств контроля структурированности и правдоподобия программ может быть существенно

упрощено при применении следующих принципов:

1. проведение этих дополнительных форм статического контроля после завершения компиляции и только для синтаксически корректных программ ;
2. максимальное использование результатов компиляции программы и, в частности, информации, включаемой в листинг компилятора;
3. вместо полного синтаксического разбора текста проверяемой программы построение для нее списка идентификаторов и списка операторов с указанием всех их необходимых признаков.

При отсутствии инструментальных средств контроля правдоподобия эта фаза статического контроля также может объединяться с визуальным контролем.

Четвертой формой статического контроля программ является их верификация, то есть аналитическое доказательство их корректности.

В интуитивном смысле под корректностью понимают свойства программы, свидетельствующие об отсутствии в ней ошибок, допущенных разработчиком на различных этапах проектирования ( спецификации, проектирование алгоритма и структур данных, кодирование ). Корректность самой программы по отношению к целям, поставленным перед ее разработкой ( то есть это относительное свойство ). Отличие понятия корректности и надежности программ в следующем :

* надежность характеризует как программу, так и ее “окружение” ( качество аппаратуры, квалификацию пользователя и т.п. );
* говоря о надежности программы, обычно допускают определенную, хотя и малую, долю ошибок в ней и оценивают вероятность их появления.

Надежность можно представить совокупностью следующих характеристик :

1. целостность программного средства (способность его к защите от отказов);
2. живучесть (способность к входному контролю данных и их проверки в ходе работы) ;
3. завершенность (бездеффектность готового программного средства, характеристика качества его тестирования);
4. работоспособность (способность программного средства к восстановлению своих возможностей поле сбоев).

Очевидно, что не всякая синтаксически правильная программа является корректной в указанном выше смысле, т. е. корректность характеризует семантические свойства программ.

С учетом специфики появления ошибок в программах можно выделить две стороны понятия корректности :

* корректность как точное соответствие целям разработки программы (которые отражены в спецификации) при условии ее завершения или частичная корректность ;
* завершение программы , то есть достижение программой в процессе ее выполнения своей конечной точки.

В зависимости от выполнения или невыполнения каждого из двух названных свойств программы различают шесть задач анализа корректности :

1. доказательство частичной корректности ;
2. доказательство частичной некорректности ;
3. доказательство завершения программы ;
4. доказательство незавершения программы ;
5. доказательство тотальной (полной ) корректности (то есть одновременное решение первой и третьей задач);
6. доказательство некорректности (решение второй или четвертой задачи).

Методы доказательства частичной корректности программ как правило опираются на аксиоматический подход к формализации семантики языков программирования. В настоящее время известны аксиоматические семантики Паскаля, подмножества ПЛ/1 и некоторых других языков.

Аксиоматическая семантика языка программирования представляет собой совокупность аксиом и правил вывода. С помощью аксиом задается семантика простых операторов языка (присваивания, ввода - вывода, вызова процедур). С помощью правил вывода описывается семантика составных операторов или управляющих структур (последовательности, условного выбора, циклов). Среди этих правил вывода надо отметить правило вывода для операторов цикла так как оно требует знания инварианта цикла (формулы, истинности которой не изменяется при любом прохождении цикла).

Построение инварианта для оператора цикла по его тексту является алгоритмически не разрешимой задачи, поэтому для описания семантики циклов требуется своего рода ”подсказка” от разработчика программы.

Наиболее известным из методов доказательства частичной корректности программ является метод индуктивных утверждений предложенный Флойдом и усовершенствованный Хоаром. Метод состоит из трех этапов.

Первый этап - получение аннотированной программы. На этом этапе для синтаксически правильной программы должны быть заданы утверждения на языке логики предикатов первого порядка :

* входной предикат ;
* выходной предикат ;
* по одному утверждению для каждого цикла (эти утверждения задаются для входной точки цикла и должны характеризовать семантику вычислений в цикле).

Доказательство неистинности условий корректности свидетельствует о неправильности программы, или ее спецификации, или программы и спецификации.

Несмотря на достаточную сложность процесса верификации программы и на то, что даже успешно завершенная верификация не дает гарантий качества программы ( т.к. ошибка может содержаться и в верификации ), применение методов аналитического доказательства правильности очень полезно для уточнения смысла разрабатываемой программы, а знание этих методов благотворно сказывается на квалификации программиста.

Наконец, динамический контроль программы - это проверка правильности программы при ее выполнении на компьютере, т.е. тестирование.

**ЦЕЛИ , ПРИНЦИПЫ И ЭТАПЫ ТЕСТИРОВАНИЯ .**

Каждому программисту известно, сколько времени и сил уходит на отладку и тестирование программ. На этот этап приходится около 50% общей стоимости разработки программного обеспечения.

Но не каждый из разработчиков программных средств может верно определить цель тестирования. Нередко можно услышать, что тестирование - это процесс выполнения программы с целью обнаружения в ней ошибок. Но эта цель недостижима : ни какое самое тщательное тестирование не дает гарантии, что программа не содержит ошибок.

Другое определение тестирования ( у Г. Майерса ) тестирование - это процесс выполнения программы с целью обнаружения в ней ошибок. Такое определение цели стимулирует поиск ошибок в программах. Отсюда также ясно, что “удачным” тестом является такой, на котором выполнение программы завершилось с ошибкой. Напротив, “неудачным можно назвать тест, не позволивший выявить ошибку в программе.

Определение Г. Майерса указывает на объективную трудность тестирования : это деструктивный ( т.е. обратный созидательному ) процесс. Поскольку программирование - процесс конструктивный, ясно, что большинству разработчиков программных средств сложно “переключиться” при тестировании созданной ими продукции.

У Майерса сформулированы также основные принципы организации тестирования :

1. необходимой частью каждого теста должно являться описание ожидаемых результатов работы программы, чтобы можно было быстро выяснить наличие или отсутствие ошибки в ней ;
2. следует по возможности избегать тестирования программы ее автором, т.к. кроме уже указанной объективной сложности тестирования для программистов здесь присутствует и тот фактор, что обнаружение недостатков в своей деятельности противоречит человеческой психологии ( однако отладка программы эффективнее всего выполняется именно автором программы ) ;
3. по тем же соображениям организация - разработчик программного обеспечения не должна “единолично ” его тестировать ( должны существовать организации, специализирующиеся на тестировании программных средств ) ;
4. должны являться правилом доскональное изучение результатов каждого теста, чтобы не пропустить малозаметную на поверхностный взгляд ошибку в программе ;
5. необходимо тщательно подбирать тест не только для правильных ( предусмотренных ) входных данных, но и для неправильных (непредусмотренных) ;
6. при анализе результатов каждого теста необходимо проверять, не делает ли программа того, что она не должна делать ;
7. следует сохранять использованные тесты (для повышения эффективности повторного тестирования программы после ее модификации или установки у заказчика) ;
8. тестирования не должно планироваться исходя из предположения, что в программе не будут обнаружены ошибки (в частности, следует выделять для тестирования достаточные временные и материальные ресурсы) ;
9. следует учитывать так называемый “принцип скопления ошибок” : вероятность наличия не обнаруженных ошибок в некоторой части программы прямо пропорциональна числу ошибок, уже обнаруженных в этой части ;
10. следует всегда помнить , что тестирование - творческий процесс, а не относиться к нему как к рутинному занятию.

Существует два основных вида тестирования : функциональное и структурное. При функциональном тестировании программа рассматривается как “черный ящик” (то есть ее текст не используется). Происходит проверка соответствия поведения программы ее внешней спецификации. Возможно ли при этом полное тестирование программы ? Очевидно , что критерием полноты тестирования в этом случае являлся бы перебор всех возможных значений входных данных, что невыполнимо .

Поскольку исчерпывающее функциональное тестирование невозможно, речь может идти о разработки методов, позволяющих подбирать тесты не “вслепую”, а с большой вероятностью обнаружения ошибок в программе.

При структурном тестировании программа рассматривается как “белый ящик” (т.е. ее текст открыт для пользования ). Происходит проверка логики программы. Полным тестированием в этом случае будет такое, которое приведет к перебору всех возможных путей на графе передач управления программы (ее управляющем графе). Даже для средних по сложности программ числом таких путей может достигать десятков тысяч. Если ограничиться перебором только линейных не зависимых путей, то и в этом случае исчерпывающее структурное тестирование практически невозможно, т. к. неясно, как подбирать тесты , чтобы обеспечить “покрытие” всех таких путей. Поэтому при структурном тестировании необходимо использовать другие критерии его полноты, позволяющие достаточно просто контролировать их выполнение, но не дающие гарантии полной проверки логики программы.

Но даже если предположить, что удалось достичь полного структурного тестирования некоторой программы, в ней тем не менее могут содержаться ошибки, т.к.

1. программа может не соответствовать своей внешней спецификации, что в частности, может привести к тому, что в ее управляющем графе окажутся пропущенными некоторые необходимые пути ;
2. не будут обнаружены ошибки, появление которых зависит от обрабатываемых данных (т.е. на одних исходных данных программа работает правильно, а на других - с ошибкой).

Таким образом, ни структурное, ни функциональное тестирование не может быть исчерпывающим. Рассмотрим подробнее основные этапы тестирования программных комплексов.

В тестирование многомодульных программных комплексов можно выделить четыре этапа :

* тестирование отдельных модулей ;
* совместное тестирование модулей ;
* тестирование функций программного комплекса (т.е. поиск различий между разработанной программой и ее внешней спецификацией ) ;
* тестирование всего комплекса в целом (т.е. поиск несоответствия созданного программного продукта сформулированным ранее целям проектирования, отраженным обычно в техническом задании).

На первых двух этапах используются прежде всего методы структурного тестирования, т.к.

на последующих этапах тестирования эти методы использовать сложнее из-за больших размеров проверяемого программного обеспечения ;

последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, которые не обязательно связаны с логикой программы.

При тестировании как отдельных модулей, так и их комплексов должны быть решены две задачи :

1. построение эффективного множества тестов ;
2. выбор способа комбинирования (сборки) модулей при создании трестируемого варианта программы .

**СТРУКТУРНОЕ ТЕСТИРОВАНИЕ .**

Поскольку исчерпывающее структурное тестирование невозможно, необходимо выбрать такие критерии его полноты, которые допускали бы их простую проверку и облегчали бы целенаправленный подбор тестов.

Наиболее слабым из критериев полноты структурного тестирования является требование хотя бы однократного выполнения (покрытия) каждого оператора программы.

Более сильным критерием является так называемый критерий С1 : каждая ветвь алгоритма (каждый переход) должна быть пройдена (выполнена) хотя бы один раз. Для большинства языков программирования покрытие переходов обеспечивает и покрытие операторов, но , например, для программ на языке ПЛ/1 дополнительно к покрытию всех ветвей требуется всех дополнительных точек входа в процедуру (задаваемых операторами ENTRY) и всех ON - единиц.

Использование критерия покрытия условий может вызвать подбор тестов, обеспечивающих переход в программе, который пропускается при использовании критерия С1 (например, в программе на языке Паскаль, использующей конструкцию цикла WHILE х AND y DO... , применение критерия покрытия условий требует проверки обоих вариантов выхода из цикла : NOT x и NOT y ).

С другой стороны покрытие условий может не обеспечивать покрытия всех переходов. Например, конструкция IF A AND B THEN... требует по критерию покрытия условий двух тестов (например, A=true, B=false и A=false, B=true ), при которых может не выполняться оператор, расположенный в then - ветви оператора if.

Практически единственным средством, предоставляемым современными системами программирования, является возможность определения частоты выполнения различных операторов программы (ее профилизации). Но с помощью этого инструмента поддержки тестирования можно проверить выполнение только слабейшего из критериев полноты - покрытие всех операторов.

Правда, с помощью этого же инструмента можно проверить и выполнение критерия С1. Но для этого предварительно текст программы должен быть преобразован таким образом, чтобы все конструкции условного выбора (IF и CASE

или SWITCH ) содержали ветви ELSE или DEFAULT, хотя бы и пустые. В этом случае все ветви алгоритма , не выполнявшиеся на данном тесте будут “видимы” из таблицы частоты выполнения операторов преобразованной программы.

Актуальной остается задача создания инструментальных средств, позволяющих :

1. накапливать информации о покрытых и непокрытых ветвях для всех использованных тестов ;
2. выделять разработчику еще не покрытые при тестировании участки программы, облегчая выбор следующих тестов ;
3. поддерживать более мощные критерии полноты структурного тестирования.

**Совместное тестирование модулей.**

Известны два подхода к совместному тестированию модулей : пошаговое и монолитное тестирование.

При монолитном тестировании сначала по отдельности тестируются все модули программного комплекса, а затем все они объединяются в рабочую программу для комплексного тестирования.

При пошаговом тестировании каждый модуль для своего тестирования подключается к набору уже проверенных модулей.

В первом случае для автономного тестирования каждого модуля требуется модуль - драйвер ( то есть вспомогательный модуль, имитирующий вызов тестируемого модуля) и один или несколько модулей - заглушек ( то есть вспомогательных модулей, имитирующих работу модулей, вызываемых из тестируемого). При пошаговом тестировании модули проверяются не изолированно друг от друга, поэтому требуются либо только драйверы, либо только заглушки.

**А**

**B C D**

**E F**

ðèñ. 1

12

При сравнении пошагового и монолитного тестирования можно отметить следующие преимущества первого подхода :

1. меньшая трудоемкость (для примера на рис.1 при монолитном тестировании требуются 5 драйверов и 5 заглушек ; при пошаговом тестировании требуются или только 5 драйверов - если модули подключаются “снизу вверх ”, - или только 5 заглушек - если модули подключаются “сверху вниз”) ;
2. более раннее обнаружение ошибок в интерфейсах между модулями (их сборка начинается раньше, чем при монолитном тестировании ) ;
3. легче отладка, то есть локализация ошибок (они в основном связаны с последним из подключенных модулей ) ;
4. более совершенны результаты тестирования (более тщательная проверка совместного использования модулей).

Есть преимущества и у монолитного тестирования :

1. меньше расход машинного времени (хотя из-за большей сложности отладки может потребоваться дополнительная проверка программы и это преимущество будет сведено на нет) ;
2. предоставляется больше возможностей для организации параллельной работы на начальном этапе тестирования.

В целом более целесообразным является выбор пошагового тестирования. При его использовании возможны две стратегии подключения модулей : нисходящая и восходящая.

Нисходящее тестирование начинается с главного (или верхнего) модуля программы, а выбор следующего подключаемого модуля происходит из числа модулей, вызываемых из уже протестированных. Одна из основных проблем , возникающих при нисходящем тестировании, - создание заглушек. Это тривиальная задача, т. к. как правило недостаточно, чтобы в заглушке выполнялся вывод соответствующего информационного сообщения и возврат всегда одних и тех же значений выходных данных.

Другая проблема , которую необходимо решать при нисходящем тестировании, - форма представления тестов в программе, так как, как правило, главный модуль получает входные данные не непосредственно, а через специальные модули ввода, которые при тестировании в начале заменяются заглушками. Для передачи в главный модуль разных тестов нужно или иметь несколько разных заглушек, или записать эти тесты в файл во внешней памяти и с помощью заглушки считывать их.

Поскольку после тестирования главного модуля процесс проверки может продолжаться по-разному, следует придерживаться следующих правил :

1. модули, содержащие операции ввода-вывода, должны подключаться к тестированию как можно раньше ;
2. критические (т.е. наиболее важные ) для программы в целом модули также должны тестироваться в первую очередь.

12

Основные достоинства нисходящего тестирования :

уже на ранней стадии тестирования есть возможность получить работающий вариант разрабатываемой программы ;

быстро могут быть выявлены ошибки, связанные с организацией взаимодействие с пользователем.

Недостатки нисходящей стратегии продемонстрируются с помощью рис.2.

Допустим , что на следующем шаге тестирования заглушка модуля H заменяется его реальным текстом. Тогда

1. может оказаться трудным или даже невозможным построить такой тест на входе модуля J, который соответствовал бы любой заданной наперед последовательности значений данных на входе модуля Н ;
2. не всегда окажется возможным легко оценить соответствие значений данных на входе модуля J требуемым тестам для проверки модуля Н;
3. т. к. результаты выполнения программы на построенном для проверки модуля Н тесте выводятся не им, а модулем I , может оказаться трудным восстановлении действительных результатов работы модуля Н.

Другие проблемы, которые могут возникать при нисходящем тестировании :

* появляется соблазн совмещения нисходящего проектирования с тестированием, что, как правило, неразумно, т.к. проектирование - процесс итеративный и в нем неизбежен возврат на верхние уровни и исправление принятых ранее решений, что обесценивает результаты уже проведенного тестирования ;
* может возникнуть желание перейти к тестированию модуля следующего уровня до завершения тестирования предыдущего по объективным причинам (необходимости создания нескольких версий заглушек, использования модулями верхнего уровня ресурсов модулей нижних уровней).

При восходящем тестировании поверка программы начинается с терминальных модулей (т.е. тех, которые не вызывают не каких других модулей программы). Эта стратегия во многом противоположна нисходящему тестированию (в частности, преимущества становятся недостатками и наоборот).

Нет проблемы выбора следующего подключаемого модуля - учитывается лишь то , чтобы он вызывал только уже протестированные модули. В отличие от заглушек драйверы не должны иметь несколько версий, поэтому их разработка в большинстве случаев проще (кроме того, использование средств автоматизации и отладки облегчает создание как раз драйверов, а не заглушек).

Другие достоинства восходящего тестирования :

* поскольку нет промежуточных модулей (тестируемый модуль является для рабочего варианта программы модулем самого верхнего уровня), нет проблем, связанных с возможностью или трудностью задания тестов ;
* нет возможности совмещения проектирования с тестированием ;
* нет трудностей, вызывающих желание перейти к тестированию следующего модуля , не завершив проверки предыдущего.

Основными недостатком восходящего тестирования является то , что проверка всей структуры разрабатываемого программного комплекса возможна только на завершающей стадии тестирования.

Хотя однозначного вывода о преимущества той или иной стратегии пошагового тестирования сделать нельзя (нужно учитывать конкретные характеристики тестируемой программы), в большинстве случаев более предпочтительным является восходящее тестирование.

На третьем этапе тестирования программных комплексов (тестировании функций) используются прежде всего методы функционального тестирования.

**Функциональное тестирование.**

Обзор методов проектирования тестов при функциональном тестировании начнем с метода эквивалентного разбиения.

Т.к. нашей целью является построения множества тестов, характеризующегося наивысшей вероятностью обнаружения большинства ошибок в тестируемой программе, то тест из этого множества должен :

1. уменьшать (более чем на единицу) число других тестов, которые должны быть разработаны для достижения заранее поставленной цели “удовлетворительного” тестирования ;
2. покрывать собой значительную часть других возможных тестов .

Другими словами :

1. каждый тест должен заключать в себе проверку наибольшего числа задаваемых внешней спецификацией входных условии (ограничений на входные данные) для того, чтобы минимизировать общее число необходимых тестов ;
2. необходимо разбить область значений входных данных на конечное число подобластей (которые будут называться классами эквивалентности), чтобы можно было полагать каждый тест, являющийся представителем некоторого класса, эквивалентным любому другому тесту этого класса (т.е. обнаруживающим одни и те же ошибки).

В общем случае использование термина “класс эквивалентности” является здесь не вполне точным, т.к. выделенные подобласти могут пересекаться.

Проектирование тестов по методу эквивалентного разбиения проводится в два этапа :

* выделение по внешней спецификации классов эквивалентности;
* построение множества тестов.

На первом этапе происходит выбор из спецификации каждого входного условия и разбиение его на две или более группы, соответствующие так называемым “правильным” (ПКЭ) и “неправильным” классом эквивалентности (НКЭ), т.е. областям допустимых для программы и недопустимых значений входных данных. Этот процесс зависит от вида входного условия. Если входное условие описывает область (например, х <=0.5) или количеством (например, размер массива А равен 50) допустимых значений входных данных, то определяются один ПКЭ (х <=0.5 или размер А равен 50) и два НКЭ (х< -0.5 ; х>0.5 или размер А меньше 50 ; размер А больше 50).

Если входное условие описывает дискретное множество допустимых значений входных данных (например, В может равно -1, 0 или 1) , то определяются ПКЭ для каждого значения из множества (в данном примере 3) и один НКЭ (В<>-1 & В<>0 & В<>1).

Если входное условие описывает ситуацию “ложно быть ” (например, N>0), то определяются один ПКЭ (N>0) и один НКЭ (N<=0).

На втором этапе метода эквивалентного разбиения выделенные классы эквивалентности используются для построения тестов :

* каждому классу присваивается свой номер ;
* проектируются тесты для ПКЭ таким образом, что каждый тест покрывает как можно больше еще не покрытых ПКЭ, до тех пор, пока все ПКЭ не будут покрыты ;
* проектируются тесты для НКЭ таким образом, что каждый тест покрывает один и только один НКЭ, до тех пор, пока все НКЭ не будут покрыты.

Нарушение третьего условия приводит к тому, что некоторые тесты с недопустимыми значениями входных данных проверяют только одну ошибку и скрывают реакцию программы на другие ошибки.

Метод эквивалентного разбиения значительно лучше случайного подбора тестов, но имеет свои недостатки. Основной из них - пропуск определенных типов высокоэффективных тестов (т.е. тестов, характеризующихся большой вероятностью обнаружения ошибок). От этого недостатка во многом свободен метод анализа граничных условий.

Под граничными условиями понимают ситуации, возникающие непосредственно на границе определенного в спецификации входного или выходного условия, выше или ниже ее . Метод анализа граничных условий отличается от метода эквивалентного разбиения следующим :

* выбор любого представителя класса эквивалентности осуществляется таким образом, чтобы проверить тестом каждую границу этого класса ;
* при построении тестов рассматриваются не только входные условия, но и выходные (т.е. определенные во внешней спецификации ограничения на значения входных данных).

Общие правила метода анализа граничных условий :

1. построить тесты для границ области допустимых значений входных данных и тесты с недопустимыми значениями, соответствующими незначительному выходу за границы этой области (например, для области [-1.0 ; 1.0] строим тесты -1.0 ; 1.0 ; -1.001 ; 1.001) ;
2. построить тесты для минимального и максимального значений входных условий, определяющих дискретное множество допустимых значений входных данных, и тесты для значений, больших или меньших этих величин (например, если входной файл может содержать от 1 до 225 записей, то выбираются тесты для пустого файла, содержащего 1, 255 и 256 записей) ;
3. использовать правило 1 для каждого выходного условия (например, программа вычисляет ежемесячный расход частного лица или небольшого предприятия, минимум которого 0.00 $, а максимум 1165.50 $; тогда необходимо построить тесты, вызывающие отрицательный расход, расходы, равные 0.00 $ и 1165.50 $, и расход, больший 1165.50 $) ;
4. использовать правило 2 для каждого выходного условия (например, программа ищет и отображает на экране дисплея наиболее подходящие , в зависимости от входного условия, рефераты статей, но не более четырех ; тогда необходимо построить тесты, приводящие к отображению 0, 1, 4 рефератов и попытки ошибочного отображения 5 рефератов) ;
5. если входные и выходные данные программы представляют собой упорядоченное множество (последовательный файл, линейный список, таблицу), то пре тестировании сосредоточить внимание на первом и последнем элементе множества ;
6. попытаться найти и проверить тестами другие граничные условия.

Важность проверки границ выходных условий объясняется тем, что не всегда граничным значениям входных данных соответствуют граничные значения результатов работы программ.

Для иллюстрации необходимости анализа граничных условий приведем тривиальный пример. Пусть имеется программа, осуществляющая ввод трех чисел интерпретирующая их как длины сторон треугольника и выводящая сообщение о типе треугольника (“разносторонний”, “равнобедренный” или “равносторонний ”). Допустим также, что в программе содержится ошибка : при проверке условия построения треугольника (сумма длин любых двух сторон должна быть больше третьей) используется операция отношения >= вместо >. При проектировании тестов по методу эквивалентного разбиения будут построены тесты для случаев возможности построения треугольника (например, 3, 4, 5) и невозможности его построения (например, 1, 2, 4), т.е. ошибка в программе не будет обнаружена (на входные данные 1, 2, 3 будет выведено сообщение “разносторонний треугольник”). Но подобный тест будет получен при использовании метода анализа граничных условий.

Анализ граничных условий - один из наиболее полезных методов проектирования тестов. Но он часто оказывается неэффективным из-за того , что граничные условия иногда едва уловимы, а их выявление весьма трудно.

Общим недостатком двух рассмотренных выше методов функционального тестирования является то, что при их применение исследуются возможные комбинации входных условий. Следует, правда, заметить, что из-за весьма большого числа таких комбинаций, их анализ вызывает существенные затруднения. Но существует метод (метод функциональных диаграмм), позволяющий в этом случае систематическим образом выбрать высоко эффективные тесты. Полезным побочным эффектом этого метода является обнаружение неполноты и противоречивости во внешних спецификациях.

Функциональная диаграмма - это текст на некотором формальном языке, на который транслируется спецификация, составленная на естественном или полуформальном языках. Далее будет называться причиной отдельное входное условие и следствием - выходное условие или преобразование системы (т.е. остаточное действие программы, вызванное определенным входным условием или их комбинацией). Например, для программы обновления файла изменение в нем является преобразованием системы, а подтверждающее это изменение сообщение - выходным условием.

Метод функциональных диаграмм состоит из шести основных этапов. На первом из них (необязательном) внешняя спецификация большого размера разбивается на отдельные участки (например, спецификация компилятора языка программирования разбивается на участки, определяющие синтаксический контроль отдельных операторов языка).

На втором этапе в спецификации выделяются причины и следствия, а на третьем - анализируется семантическое содержание спецификации и она преобразуется в булевский граф, связывающий причины и следствия и называющийся функциональной диаграммой. На рис.3 приведены базовые символы для записи функциональных диаграмм (каждый узел функциональной диаграммы может находиться в состоянии 1 - “существует” - или 0 - “не существует”).

а) Тождество : (а=1=>b=1) & (а=0=>b=0)

а b

б) Отрицание : (а=1=>b=0) & (a=0=>b=1)

~

a b

в) Дизъюнкция : (a=1⎪b=1=>c=1) & (a=0&b=>0 >c=0)

a

⎪ c

b

г) Конъюнкция : (a=1&b=1=>c=1) & (a=0⎪b=0=>c=0)

a

& c

b

рис.3

На четвертом этапе функциональная диаграмма снабжается комментариями, которые задают ограничения на комбинации причин и следствий. На рис.4 приведены знаки комментариев, задающих эти ограничения.

а) Исключение одной из причин :

a

E ((a=1⎪b=1)^~(a=1&b=1)) ⎪ (a=0&b=0)

b

б) Включение хотя бы одной причины :

a

I (a=1⎪b=1)&~(a=0&b=0)

b

в) Существует одна и только одна причина :

a

O (a=1⎪b=1)&~(a=1&b=1)&~(a=0&b=0)

b

г) Одна причина влечет за собой другую :

a

R ~(a=1&b=0)

b

д) Одно следствие скрывает в себе другое :

a

M (a=1&b=0)&(a=1&b=1)

b

рис.4

Пятый этап - функциональная диаграмма преобразуется в таблицу решений :

выбирается следствие, которое устанавливается в 1 ;

находятся все комбинации причин (с учетом ограничений),

которые устанавливают выбранное следствие в 1