## Способы тестирования условий

Цель этого семейства способов тестирования — строить тестовые варианты для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.

Рассмотрим используемую здесь терминологию.

Простое условие — булева переменная или выражение отношения.

Выражение отношения имеет вид

Е1 <оператор отношения> E2*,*

где El, Е2 — арифметические выражения, а в качестве оператора отношения используется один из следующих операторов: <, >, =, *, .*

Составное условие состоит из нескольких простых условий, булевых операторов и круглых скобок. Будем применять булевы операторы OR, AND (&), NOT. Условия, не содержащие выражений отношения, называют булевыми выражениями.

Таким образом, элементами условия являются: булев оператор, булева переменная, пара скобок (заключающая простое или составное условие), оператор отношения, арифметическое выражение. Эти элементы определяют типы ошибок в условиях.

Если условие некорректно, то некорректен по меньшей мере один из элементов условия. Следовательно, в условии возможны следующие типы ошибок:

* ошибка булева оператора (наличие некорректных / отсутствующих / избыточных булевых операторов);
* ошибка булевой переменной;
* ошибка булевой скобки;
* ошибка оператора отношения;
* ошибка арифметического выражения.

Способ тестирования условий ориентирован на тестирование каждого условия в программе. Методики тестирования условий имеют два достоинства. Во-первых, достаточно просто выполнить измерение тестового покрытия условия. Во-вторых, тестовое покрытие условий в программе — это фундамент для генерации дополнительных тестов программы.

Целью тестирования условий является определение не только ошибок в условиях, но и других ошибок в программах. Если набор тестов для программы А эффективен для обнаружения ошибок в условиях, содержащихся в А, то вероятно, что этот набор также эффективен для обнаружения других ошибок в А. Кроме того, если методика тестирования эффективна для обнаружения ошибок в условии, то вероятно, что эта методика будет эффективна для обнаружения ошибок в программе.

Существует несколько методик тестирования условий.

Простейшая методика — *тестирование ветвей.* Здесь для составного условия *С* проверяется:

* каждое простое условие (входящее в него);
* Тruе-ветвь;
* False-ветвь.

Другая методика — *тестирование области определения.* В ней для выражения отношения требуется генерация 3-4 тестов. Выражение вида

Е1 <оператор отношения> Е2

проверяется тремя тестами, которые формируют значение Е1 большим, чем Е2, равным Е2 и меньшим, чем Е2.

Если оператор отношения неправилен, а Е1 и Е2 корректны, то эти три теста гарантируют обнаружение ошибки оператора отношения.

Для определения ошибок в Е1 и Е2 тест должен сформировать значение Е1 большим или меньшим, чем Е2, причем обеспечить как можно меньшую разницу между этими значениями.

Для булевых выражений с *п* переменными требуется набор из 2*n* тестов. Этот набор позволяет обнаружить ошибки булевых операторов, переменных и скобок, но практичен только при малом *п.* Впрочем, если в булево выражение каждая булева переменная входит только один раз, то количество тестов легко уменьшается.

Обсудим способ тестирования условий, базирующийся на приведенных выше методиках.

### Тестирование ветвей и операторов отношений

Способ тестирования ветвей и операторов отношений (автор К. Таи, 1989) обнаруживает ошибки ветвления и операторов отношения в условии, для которого выполняются следующие ограничения [72]:

* все булевы переменные и операторы отношения входят в условие только по одному разу;
* в условии нет общих переменных.

В данном способе используются естественные ограничения условий (ограничения на результат). Для составного условия *С,* включающего *п* простых условий, формируется ограничение условия:

ОУс = (*d1,d2,d3.....dn),*

где *di* — ограничение на результат *i*-го простого условия.

Ограничение на результат фиксирует возможные значения аргумента (переменной) простого условия (если он один) или соотношения между значениями аргументов (если их несколько).

Если i-e простое условие является булевой переменной, то его ограничение на результат состоит из двух значений и имеет вид

*di =* (true,false).

Если *j*-е простое условие является выражением отношения, то его ограничение на результат состоит из трех значений и имеет следующий вид:

*dj= (>,<,=).*

Говорят, что ограничение условия ОУc (для условия *С*)покрывается выполнением *С,* если в ходе этого выполнения результат каждого простого условия в *С* удовлетворяет соответствующему ограничению в ОУc.

На основе ограничения условия ОУ создается ограничивающее множество ОМ, элементы которого являются сочетаниями всех возможных значений *d1, d2, d3, ..., dn.*

Ограничивающее множество — удобный инструмент для записи задания на тестирование, ведь оно составляется из сведений о значениях переменных, которые влияют на значение проверяемого условия. Поясним это на примере. Положим, надо проверить условие, составленное из трех простых условий:

*b&(х>у)&а.*

Условие принимает истинное значение, если все простые условия истинны. В терминах значений простых условий это соответствует записи

(true, true, true),

а в терминах ограничений на значения аргументов простых условий — записи

(true, >, true).

Ясно, что вторая запись является прямым руководством для написания теста. Она указывает, что переменная *b* должна иметь истинное значение, значение переменной *х* должно быть больше значения переменной *у,* и, наконец, переменная *а* должна иметь истинное значение.

Итак, каждый элемент ОМ задает отдельный тестовый вариант. Исходные данные тестового варианта должны обеспечить соответствующую комбинацию значений простых условий, а ожидаемый результат равен значению составного условия.

**Пример 1**. В качестве примера рассмотрим два типовых составных условия:

*С& = а & Ь, Сor =а or b,*

где *а* и *b —* булевы переменные. Соответствующие ограничения условий принимают вид

ОУ*&*=( *d*1,*d*2), ОУor=( *d*1,*d*2),

где *d1* = *d2* = (true, false).

Ограничивающие множества удобно строить с помощью таблицы истинности (табл. 6.1).

**Таблица 6.1.** Таблица истинности логических операций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Вариант** | ***а*** | ***b*** | ***a & b*** | ***a or b*** |
| 1 | false | false | false | false |
| 2 | false | true | false | true |
| 3 | true | false | false | true |
| 4 | true | true | true | true |

Видим, что таблица задает в ОМ четыре элемента (и соответственно, четыре тестовых варианта). Зададим вопрос — каковы возможности минимизации? Можно ли уменьшить количество элементов в ОМ?

С точки зрения тестирования, нам надо оценивать влияние составного условия на программу. Составное условие может принимать только два значения, но каждое из значений зависит от большого количества простых условий. Стоит задача — избавиться от влияния избыточных сочетаний значений простых условий.

Воспользуемся идеей сокращенной схемы вычисления — элементы выражения вычисляются до тех пор, пока они влияют на значение выражения. При тестировании необходимо выявить ошибки переключения, то есть ошибки из-за булева оператора, оперируя значениями простых условий (булевых переменных). При таком инженерном подходе справедливы следующие выводы:

* для условия типа И *(а & b)* варианты 2 и 3 поглощают вариант 1. Поэтому ограничивающее множество имеет вид:

ОМ& = {(false, true), (true, false), (true, true)};

* для условия типа ИЛИ *(а* or *b)* варианты 2 и 3 поглощают вариант 4. Поэтому ограничивающее множество имеет вид:

ОМor = {(false, false), (false, true), (true, false)}.

Рассмотрим шаги **способа тестирования ветвей и операторов отношений.**

Для каждого условия в программе выполняются следующие действия:

1) строится ограничение условий ОУ;

2) выявляются ограничения результата по каждому простому условию;

3) строится ограничивающее множество ОМ. Построение выполняется путем подстановки в константные формулы ОМ& или OMOR выявленных ограничений результата;

4) для каждого элемента ОМ разрабатывается тестовый вариант.

**Пример 2.** Рассмотрим составное условие *С*1вида:

*В1 &(E1,E2),*

где *В1* — булево выражение, *E1, Е2* — арифметические выражения.

Ограничение составного условия имеет вид

ОУ =( *d*1,*d*2),

где ограничения простых условий равны

*d1 =* (true, false), *d2 =* (=, <, >).

Проводя аналогию между *С1* и *С&* (разница лишь в том, что в *С1* второе простое условие — это выражение отношения), мы можем построить ограничивающее множество для *С1* модификацией

ОМ& = {(false, true), (true, false), (true, true)}.

Заметим, что *true* для (*E1*= *E2)* означает =, a false для *(E1* = *E2)* означает или <, или >. Заменяя (true, true) и (false, true), ограничениями (true, =) и (false, =) соответственно, a (true, false) — ограничениями (true, <) и (true, >), получаем ограничивающее множество для *С1*:

ОМ = {(false,=),(true,<),(true,>),(true,=)}.

Покрытие этого множества гарантирует обнаружение ошибок булевых операторов и операторов отношения в С1.

**Пример 3.** Рассмотрим составное условие *С2* вида

*(E3 >E4)&(E1=E2),*

где *E1, Е2, Е3, Е4* — арифметические выражения. Ограничение составного условия имеет вид

ОУ =( *d*1,*d*2),

где ограничения простых условий равны

*d*1=(=,<,>), *d*2 =(=,<,>).

Проводя аналогию между *С2* и *С1* (разница лишь в том, что в *С2* первое простое условие — это выражение отношения), мы можем построить ограничивающее множество для *С2* модификацией ОМ:

ОМ = {(=, =), (<, =), (>, <),(>, >),(>, =)}.

Покрытие этого ограничивающего множества гарантирует обнаружение ошибок операторов отношения в *С2.*

## Способ тестирования потоков данных

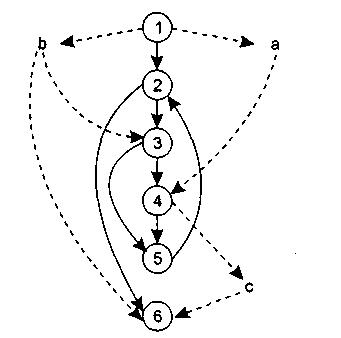
В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается информационная структура программы.

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Рассмотрим пример.

Пусть потоковый граф программы имеет вид, представленный на рис. 6.8. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

* в вершине 1 определяются значения переменных *а, b;*
* значение переменной *а* используется в вершине 4;
* значение переменной *b* используется в вершинах 3, 6;
* в вершине 4 определяется значение переменной с, которая используется в вершине 6.



**Рис. 6.8.** Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

* множество определений данных

DEF(i) = { х | i -я вершина содержит определение х};

* множество использований данных:

USE (i) = { х | i -я вершина использует х}.

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

*x*:=*f*(…).

*Использование данных —* это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

:=*f*(*x*).

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).

Назовём *DU-цепочкой (цепочкой определения-использования)* конструкцию *[х, i,j],* где *i,j —* имена вершин; х определена в *i*-й вершине *(х * DЕF(*i*)) и используется в *j* -й вершине *(х * USE(*j*)).

В нашем примере существуют следующие DU-цепочки:

[а,1,4],[b, 1,3], [b, 1,6], [с, 4, 6].

Способ *DU-тестирования* требует охвата всех DU-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Очевидно, что для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. Критерий для выбора пути — покрытие максимального количества DU-цепочек.

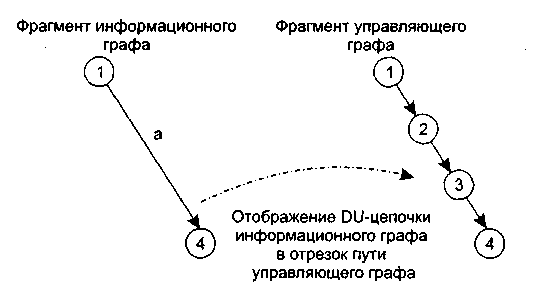
*Шаги способа DU-тестирования:*

1) построение управляющего графа (УГ) программы;

2) построение информационного графа (ИГ);

3) формирование полного набора DU-цепочек;

4) формирование полного набора отрезков путей в управляющем графе (отображением набора DU-цепочек информационного графа, рис. 6.9);



**Рис. 6.9.** Отображение DU-цепочки *в* отрезок пути

5) построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа;

6) подготовка тестовых вариантов.

*Достоинства DU-тестирования:*

* простота необходимого анализа операционно-управляющей структуры программы;
* простота автоматизации.

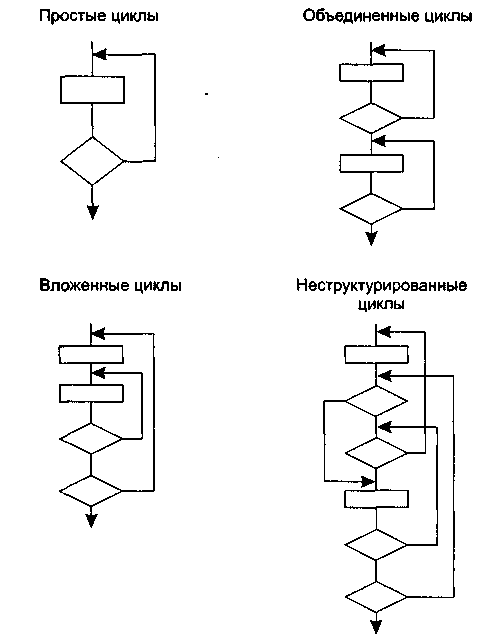
*Недостаток DU-тестирования:* трудности в выборе минимального количества максимально эффективных тестов.

*Область использования DU-тестирования:* программы с вложенными условными операторами и операторами цикла.

## Тестирование циклов

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в ПО. Тестирование циклов производится по принципу «белого ящика», при проверке циклов основное внимание обращается на правильность конструкций циклов.

Различают 4 типа циклов: простые, вложенные, объединенные, неструктурированные. Структура циклов приведена на рис. 6.10.



**Рис. 6.10.** Типовые структуры циклов

### Простые циклы

Для проверки простых циклов с количеством повторений *п* может использоваться один из следующих наборов тестов:

1) прогон всего цикла;

2) только один проход цикла;

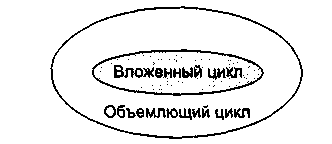
3) два прохода цикла;

4) *т* проходов цикла, где *т<п;*

5) *п* - 1, *п, п +* 1 проходов цикла.

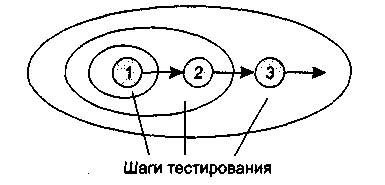
### Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов [13]. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный циклы (рис. 6.11).



**Рис. 6.11.** Объемлющий и вложенный циклы

Порядок тестирования вложенных циклов иллюстрирует рис. 6.12.



**Рис. 6.12.** Шаги тестирования вложенных циклов

*Шаги тестирования.*

1. Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.

2. Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.

3. Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.

4. Работа продолжается до тех пор, пока не будут протестированы все циклы.

### Объединенные циклы

Если каждый из циклов независим от других, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

### Неструктурированные циклы

Неструктурированные циклы тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций.

## Особенности тестирования «черного ящика»

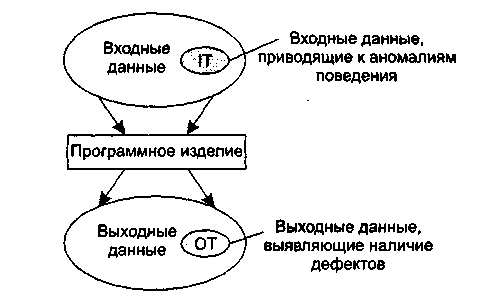
Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе [14]. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

* набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *IT);*
* набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *ОТ).*

Как показано на рис. 7.1, любой способ тестирования «черного ящика» должен:

* выявить такие входные данные, которые с высокой вероятностью принадлежат набору IT;
* сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора ОТ.

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных, обсуждаемый в данной главе, может использоваться как полезное дополнение к эвристическому знанию.



**Рис. 7.1.** Тестирование «черного ящика»

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

1) некорректных или отсутствующих функций;

2) ошибок интерфейса;

3) ошибок во внешних структурах данных или в доступе к внешней базе данных;

4) ошибок характеристик (необходимая емкость памяти и т. д.);

5) ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

* сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);
* выявление классов ошибок, а не отдельных ошибок.

## Способ разбиения по эквивалентности

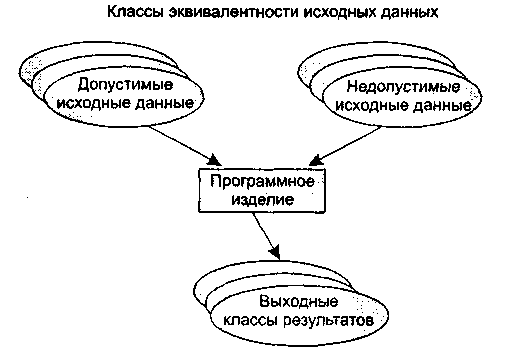
Разбиение по эквивалентности — самый популярный способ тестирования «черного ящика» [3], [14].

В этом способе входная область данных программы *делится* на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности — набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рис. 7.2 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.



**Рис. 7.2.** Разбиение по эквивалентности

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000...70 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

* числа меньшие, чем 15 000;
* числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

1) определенное значение;

2) диапазон значений;

3) множество конкретных величин;

4) булево условие.

Сформулируем *правила формирования классов эквивалентности.*

1. Если условие ввода задает диапазон *п...т,* то определяются один допустимый и два недопустимых класса эквивалентности:

* V\_Class={*n*.. *.т*} *—* допустимый класс эквивалентности;
* Inv\_С1аss1={*x*|для любого *х: х < п*} *—* первый недопустимый класс эквивалентности;
* Inv\_С1аss2={*y*|для любого *у: у > т*} *—* второй недопустимый класс эквивалентности.

2. Если условие ввода задает конкретное значение *а,* то определяется один допустимый и два недопустимых класса эквивалентности:

* V\_Class={*a*};
* Inv\_Class1 *={х*|длялюбого *х: х < а*};
* Inv\_С1аss2={*y*|для любого *у: у* > *а*}*.*

3.Если условие ввода задает множество значений *{а, b, с},* то определяются один допустимый и один недопустимый класс эквивалентности:

* V\_Class={*a, b, с*};
* Inv\_С1аss={*x*|для любого *х: (х  а)&(х  b)&(х  с)}.*

4. Если условие ввода задает булево значение, например true, то определяются один допустимый и один недопустимый класс эквивалентности:

* V\_Class={true};
* Inv\_Class={false}.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

## Способ анализа граничных значений

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения [3], [14], [69]. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;

2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

Сформулируем **правила анализа граничных значений.**

1. Если условие ввода задает диапазон *п...т*, тотестовые варианты должны быть построены:

* для значений *п* и *т;*
* для значений чуть левее *п* ичуть правее *т* на числовой оси.

Например, если задан входной диапазон -1,0...+1,0, то создаются тесты для значений - 1,0, +1,0, - 1,001, +1,001.

2. Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:

* для проверки минимального и максимального из значений;
* для значений чуть меньше минимума и чуть больше максимума.

Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для О, 1, 255, 256 записей.

3. Правила 1 и 2 применяются к условиям области вывода.

Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).

4. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

5. Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

Большинство разработчиков используют этот способ интуитивно. При применении описанных правил тестирование границ будет более полным, в связи с чем возрастет вероятность обнаружения ошибок.

Рассмотрим применение способов разбиения по эквивалентности и анализа граничных значений на конкретном примере. Положим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве элементов *М,* возвращается индекс *I* элемента массива, значение которого соответствует ключу поиска *Key.*

*Предусловия:*

1) массив должен быть упорядочен;

2) массив должен иметь не менее одного элемента;

3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

*Постусловия:*

1) если элемент найден, то флаг Result=True, значение I — номер элемента;

2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость предусловий, на втором уровне — выполнимость постусловий. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;

2) массив из четного количества элементов;

3) массив из нечетного количества элементов, большего единицы.

Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;

2) работа с последним элементом массива;

3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рис. 7.3.

