**Лекция 3. Тема занятия.** Логически обратимая машины Тьюринга. Универсальная машина Тьюринга. Методика доказательства правильности программ.

**ЦЕЛЬ**: Ознакомить с понятиями: Логически обратимая машины Тьюринга.Универсальная машина Тьюринга. Методика доказательства правильности программ.

**ВОПРОСЫ**:

1. Логически обратимая машины Тьюринга.
2. Универсальная машина Тьюринга.
3. Методика доказательства правильности программ.

**ОСНОВНЫЕ ПОНЯТИЯ:** Логически обратимая машины Тьюринга.Универсальная машина Тьюринга. Методика доказательства правильности программ.

**Логически обратимые и необратимые компьютеры**

 Определение.(1) (По [Кван­то­вый,1999,с.18]) **Ло­ги­че­с­ки не­об­ра­ти­мым устро­йством** на­зо­вём та­кое устро­йство, ес­ли по сиг­на­лу на его вы­хо­де не­льзя одно­знач­но опре­де­лить сиг­нал на его вхо­де. (2) (По [Кван­то­вый,1999,с.34]) Бу­дем го­во­рить, что **ком­пь­ю­тер яв­ля­ет­ся ло­ги­че­с­ки не­об­ра­ти­мым**, ес­ли его фун­кция пе­ре­хо­да (час­тич­ная фун­кция, отоб­ра­жа­ющая каж­дое гло­ба­ль­ное со­сто­яние ма­ши­ны на по­сле­ду­ющее ему со­сто­яние (ес­ли, ко­неч­но, та­ко­вое име­ет­ся)) не об­ла­да­ет одно­знач­ной об­рат­ной фун­кци­ей.

**Ло­ги­че­с­ки не­об­ра­ти­мый ком­пь­ю­тер** всег­да мож­но сде­лать об­ра­ти­мым, со­хра­няя всю ин­фор­ма­цию, ко­то­рая бы в дру­гих слу­ча­ях те­ря­лась. На­при­мер, ма­ши­не мож­но до­ба­вить до­пол­ни­те­ль­ную лен­ту (из­на­ча­ль­но пус­тую), на ко­то­рую мож­но за­пи­сы­вать каж­дую опе­ра­цию, как то­ль­ко она бу­дет про­из­ве­де­на, де­лая это до­ста­то­чно под­роб­но так, что пред­шес­тву­ющее со­сто­яние мож­но бы­ло бы одно­знач­но опре­де­лить по на­сто­яще­му со­сто­янию и по­след­ней за­пи­си на лен­те. Од­на­ко, как от­ме­чал Р.Ландауэр, это то­ль­ко отод­ви­нет про­бле­му от­бра­сы­ва­ния не­нуж­ной ин­фор­ма­ции, по­ско­ль­ку лен­та до­лжна быть очи­ще­на пе­ред но­вым ис­по­ль­зо­ва­ни­ем. Та­ким об­ра­зом, не­об­хо­ди­мы та­кие об­ра­ти­мые ком­пь­ю­те­ры, ко­то­рые при оста­но­вке сти­ра­ют все про­ме­жу­то­чные ре­зу­ль­та­ты, остав­ляя то­ль­ко не­об­хо­ди­мые вы­ход­ные и на­ча­ль­ные вход­ные дан­ные. (Ма­ши­на до­лжна иметь воз­мож­ность со­хра­нять вход­ные дан­ные - в про­тив­ном слу­чае она не бу­дет об­ра­ти­мой и по-пре­жне­му бу­дет про­во­дить вы­чис­ле­ния, при ко­то­рых вход­ные дан­ные не опре­де­ля­ют­ся вы­ход­ны­ми одно­знач­но).

**Универсальной машиной Тьюринга** называют [машину Тьюринга](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3%D0%B0), которая может заменить собой любую машину Тьюринга. Получив на вход программу и входные данные, она вычисляет ответ, который вычислила бы по входным данным машина Тьюринга, чья программа была дана на вход.

**Формальное определение УМТ.** Программу любой детерминированной машины Тьюринга можно записать, используя некоторый конечный алфавит, состоящий из символов состояния, скобок, стрелки и т. п.; обозначим этот машинный алфавит как Σ1. Тогда универсальной машиной Тьюринга U для класса машин с алфавитом Σ2 и k входными лентами называется машина Тьюринга с k+1 входной лентой и алфавитом \Sigma_1 \cup \Sigma_2 такая, что если подать на первые k лент входное значение, а на k+1 — правильно записанный код некоторой машины Тьюринга M1, то U выдаст тот же ответ, какой выдала бы на этих входных данных M1, или будет работать бесконечно долго, если M1 на этих данных не остановится.

**Теорема об универсальной машине Тьюринга** утверждает, что такая машина существует и моделирует другие машины с не более чем квадратичным замедлением (то есть если исходная машина произвела t шагов, то универсальная произведёт не более ct²). Теорема была предложена и доказана [Тьюрингом](http://ru.wikipedia.org/wiki/%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3,_%D0%90%D0%BB%D0%B0%D0%BD) в [1947](http://ru.wikipedia.org/wiki/1947) г.

Универсальной Машиной Тьюринга (УМТ) для алфавита 𝐴 называется такая машина U, на которой может быть промоделирована любая МТ над алфавитом А.

На самом деле можно эффективно построить УМТ, моделирующую любую МТ над любым алфавитом. Для этого фиксируется некоторый алфавит (например, А2 = {0, 1}) и добавляется кодирование и декодирование.

Идея УМТ заключается в следующем:

На ленту УМТ записывается программа моделируемой МТ (таблица) и исходные данные моделируемой МТ. УМТ по состоянию и текущему символу МТ находит на своей ленте команду моделируемой МТ, выясняет, какое действие нужно выполнить, выполняет его.

Теперь должно быть понятно, чем нам удобна МТ с ограниченной слева лентой. Слева мы можем хранить программу, а дальше данные моделируемой МТ.

Рассмотрим представление программы моделируемой МТ Т:

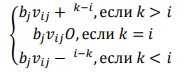
• Рабочий алфавит Ар

• Состояния 𝑞0, 𝑞1, … , 𝑞𝑠

• Правила 𝑞𝑖𝑎𝑗 → 𝑣𝑖𝑗𝑞𝑘, где i, k = 0, …, n; j = 1, …, p; 𝑣𝑖𝑗 ∈ {𝑎1, 𝑎2, … , 𝑎𝑝,𝑙, 𝑟, ℎ} Универсальная машина:

• Алфавит 𝐵𝑝 = {𝑏1, 𝑏2, … , 𝑏𝑝}

• Дополнительные символы {l, r, h, +, -, O} Для правила 𝑞𝑖𝑎𝑗 → 𝑣𝑖𝑗𝑞𝑘 запись выглядит так



+ k-i означает символ, повторенный k-i раз

Правило 𝑞𝑖𝑎𝑗 → 𝑣𝑖𝑗𝑞𝑘 укороченное и означает, что любой паре состояние-символ соответствует инструкция, описывающая действие (сдвиг или запись символа) и переход в новое состояние. 𝐵𝑝 – это зеркальный алфавит для исходного алфавита, то есть каждый символ из алфавита является отличительной копией символа из исходного алфавита. Зеркальный алфавит используется для того, чтобы определить, что будет записываться на ленте слева, а что справа.

Рассмотрим укороченный пример. Допустим у МТ есть два состояния: 𝑞0, 𝑞1 и ее алфавит содержит два символа: 0,1. Соответственно есть группа из 4 правил: 𝑞0, 0

𝑞0, 1

𝑞1, 0

𝑞1, 1

Эта группа легко делится по состояниям на 𝑞0, 0; 𝑞0, 1 и 𝑞1, 0; 𝑞1, 1. Для каждого состояния нужно понять, как записать слово на ленте, соответствующее описанию этих правил. Как для пары состояние-символ записать инструкцию с новым состоянием? Можно просто записать номер этого состояния, для этого нужно завести какую-то систему счисления, закодировать все эти номера в этой системе счисления и просто записать номер. Тогда, чтобы понять, где на ленте будет запись для следующего состояния нужно будет как-то этот номер раскодировать, что не очень просто. Поэтому пользуются, по сути, относительной адресацией. То есть, если мы записываем для состояния 𝑞2 правило: 𝑞2, 0 → 0, 𝑞4. То есть мы в слове, где есть группа правил для 𝑞2, записываем в зеркальном виде символ 1 как 1̃, потом записываем, что нужно сделать: либо символ из исходного алфавита, либо действие. Дальше нужно указать, что мы попадем в четвертое состояние 𝑞4. Для этого просто записываем, насколько 4 больше 2. Если разница положительна, ставим нужное количество «+», если отрицательна – нужное количество «-», если разницы нет, ставится 0. В нашем случае достаточно поставить два «+». В итоге получаем запись одного правила:̃10++

Пример.

𝑞0, 0 0, 𝑞0

𝑞0, 1 0, 𝑞1

𝑞1, 0 r, 𝑞1

𝑞1, 1 1, 𝑞0

Записываем все правила в порядке номеров состояний, то есть сначала записываем все правила для состояния 𝑞0: 0̃∅𝑂1̃∅+

Для 0 записываем 0 (∅) и остаемся в том же состоянии (разница между состояниями равна 0, то есть O)

Аналогично для 𝑞1: 0̃𝑟𝑂1̃1 –

Это слово описывает все правила для состояния 𝑞1. Чтобы эти слова удобно отделить на ленте, используем маркер, например, С. В общем виде: Слово-программа: 𝑐𝑤0𝑐𝑤1 … 𝑐𝑤𝑛§, где 𝑤𝑖 – слово с записью подряд всех правил состояния 𝑞𝑖

• Слова правил разных состояний отделяются друг от друга вспомогательным маркером с.

• Вся программа заканчивается маркером §.

Лента в начальном состоянии (рис.1):

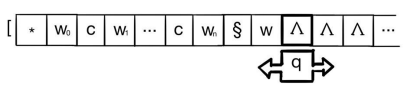


рис. 1 Лента в начальном состоянии

w- исходные данные моделируемой МТ

\* - маркер начального состояния

Для интерпретации моделируемой МТ необходимо сделать следующее:

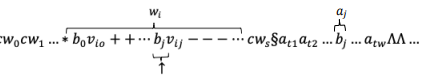
1. Найти правила для выполнения

1.1 «Запоминаем» обозреваемый символ 𝑎𝑖 размножением состояний

1.2 Заменяем символ 𝑎𝑗 на его зеркальную пару 𝑏𝑗

1.3 Ищем слово 𝑤𝑖 , содержащее запись правила

1.4 Ищем запись правила для символа 𝑎j



2. Изменить текущее состояние моделируемой МТ

2.1 Сдвигаемся на один символ вправо, пропуская 𝑣𝑖𝑗

2.2 По описанию сдвига пропускаем соответствующее количество символовмаркеров с и ставим маркер текущего состояния \*

2.3 Возвращаемся на символ описания 𝑣𝑖𝑗 действия (см. рис.2)

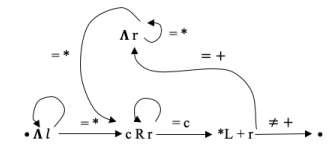


Рис. 2 Диаграмма изменения текущего состояния

3. Выполнить действия моделируемой МТ

3.1 Ищем ячейку ленты, на которой находится УГ моделируемой МТ

3.2 Выполняем считанное действие (запись символа или сдвиг)

Если при сдвиге УГ попала на символ §, отделяющий программу моделируемо МТ от данных, это означает, что моделируема МТ зашла за левый край ленты

1. Перейти на выполнение нового такта По сути, ничего не делаем, так как машина уже готова вернуться в пункт 1.

Это и есть процесс выполнения основного цикла интерпретации моделируемой машины.

Находясь в состоянии, когда справа на данных моделируемой машины необходимо найти правило для выполнения, запоминаем размножением состояний обозреваемый символ, заменяем его на зеркальный, чтобы иметь возможность в него вернуться, передвигаемся налево до \*, передвигаемся направо до запомненного символа. Таким образом находим правило. Потом нужно выполнить сдвиг звездочки на нужное количество с направо, либо налево в зависимости от + или -, либо остаться на месте, если 0. После этого в зависимости от действия вернуться на зеркальный символ, который был до этого справа от §, записать этот символ, либо сдвинуться и заменить этот символ на незеркальный, обычный символ из алфавита А𝑝. На этом цикл закончился. Мы выполнили основную работу по построению универсальной машины. Теперь нужно понять, как выйти из цикла, то есть сделать останову. Если при сдвиге маркера текущего состояния (шаг 2.2) происходит переход на символ §, то следующим состоянием будет являться состояние останова. В таком случае УМТ нужно выполнить действие моделируемой машины, а потом остановиться. Лента в состоянии останова будет выглядеть следующим образом (рис.3). С остановом все просто: нужно либо записать H (вместо L или R), либо присваиваем состоянию останова самый большой номер. Например, на рис. 3 у нас два состояния 𝑞0 и 𝑞1, состоянию останова присваиваем 𝑞2 и кодируем его так же, как и другие состояния, разницей из + и -. Далее, если мы попытаемся перейти в состояние остановы и будем сдвигать маркер \* через с, в какой-то момент мы будем искать следующее c, которой не будет, потому что мы упремся в §. То есть мы захотим взять следующее слово, а его не будет. Это и будет признак остановы. В таком случае нужно выполнить последнее действие, иначе мы его потеряем и остановиться. В результате работа машины закончится на первой пустой ячейке благодаря нормальным вычислениям, а \* исчезнет. Если бы мы кодировали состояние остановы символом H, мы могли бы оставить по желанию \*.

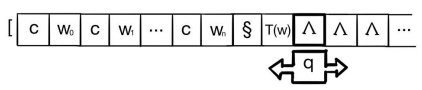


Рис. 3. Лента в состоянии останова

Таким образом мы доказали существование Универсальной Машины Тьюринга, построив ее.

**Имитация машины Тьюринга на компьютере**.

Пусть *Т* - машина Тьюринга, одним из составляющих которой является ее конечное управление. Поскольку *Т* имеет конечное число состояний и конечное число правил перехода, программа компьютера может закодировать состояния в виде цепочек символов, как и символы ее внешнего алфавита, и использовать таблицу переходов машины *Т* для преобразования цепочек. Бесконечную ленту машины Тьюринга можно имитировать сменными дисками, размещаемыми в двух магазинах, соответственно для данных, расположенных слева и справа от считывающей головки на ленте. Чем дальше в магазине расположены данные, тем дальше они от головки на ленте.

***Для имитации компьютера на машине Тьюринга*** существенны две вещи:

существуют ли инструкции, выполняемые компьютером, и недоступные для машины Тьюринга;

работает ли компьютер быстрее машины Тьюринга, т.е. более, чем полиномиальная зависимость разделяет время работы компьютера и машины Тьюринга при решении какой-то проблемы.

***Неформальная модель реального компьютера***:

память, состоящая из последовательности слов и их адресов. В качестве адресов будут использоваться натуральные числа 0,1, …;

программа компьютера, записанная в слова памяти, каждое из которых представляет простую инструкцию. Допускается “непрямая адресация” по указателям;

каждая инструкция использует конечное число слов и изменяет значение не более одного слова;

имеются слова памяти с быстрым доступом (регистры), но скорость доступа к различным словам влияет лишь на константный сомножитель, что не искажает полиномиальную зависимость.

Возможная конструкция машины Тьюринга для имитации компьютера представлена на рис.4.



Рис.4

Машина имеет несколько лент. Первая лента представляет всю память компьютера – адреса и значения (в двоичной системе). Адреса заканчиваются маркером \*, значения – маркером #. Начало и конец записей 1-й ленты обозначаются маркером *$*. Вторая лента – “счетчик инструкций”, содержит одно двоичное целое, представляющее одну из позиций считывающей головки на первой ленте, адрес инструкции, которая должна быть выполнена следующей. Третья лента содержит адрес и значение по нему после того, как этот адрес устанавливается на первой ленте. Для выполнения инструкции машина Тьюринга должна найти значение по одному или нескольким адресам памяти, где хранятся данные, участвующие в вычислении. Нужный адрес копируется на ленту 3 и сравнивается с адресами на ленте 1 до совпадения. Значение по этому адресу копируется на третью ленту и перемещается на нужное место, как правило, по одному из начальных адресов, представляющих регистры компьютера. Четвертая лента имитирует входной файл. Пятая лента - рабочая память, служит для выполнения вычислений.

**МЕТОДИКА ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ ПРОГРАММ**

**Формальные методы доказательства правильности программ и их спецификаций.** Традиционные методы анализа ПО связаны с **доказательством правильности программ (верификация программ).** (подробнее см. литературу)

Фундаментальный вклад в **теорию верификации** внес Ч. Хоор, высказавший идеи проведения доказательства частичной правильности программы в виде вывода в некоторой логической системе, а Э. Дейкстра ввел понятие слабейшего предусловия, позволяющее одновременно как соответствие друг другу предусловия и постусловия, так и завершимость. Методы доказательства правильности программ принесли определенную пользу программированию. Было отмечено, что эти методы указывают способ рассуждения о ходе выполнения программ, дают удобную систему комментирования программ и устанавливают взаимосвязи между конструкциями языков программирования и их семантикой. Если принять более широкое толкование термина "анализ программ", подразумевая доказательство разнообразных свойств программ или доказательство теорем о программах, то ценность методов анализа станет более ясной. В частности можно исследовать характер изменения выходных значений программы, количество операций при выполнении программы, наличие зацикливаний, незадействованных участков программы. Таким образом, в некоторых частных случаях методы верификации могут применяться и для доказуемого обнаружения программных дефектов.

Формальное доказательство в виде вывода в некоторой логической системе вполне надежно, но сами доказательства оказываются очень длинными и часто необозримыми.

**Методика доказательства правильности программ.** Рассматриваемая методика предназначена для доказательства правильности алгоритмов, представленных в виде графов, вершинам которых поставлены в соответствие операторы над памятью, а дугам переходы от оператора к оператору. Одну из вершин назовем входной, ей соответствует оператор, с которого начинается выполнение алгоритма, а выходных вершин может быть несколько. Считаем, что входная и выходная вершины помечены, соответственно, входящей и выходящей стрелками. Такие представления алгоритмов называют блок-схемами.

**Доказать правильность алгоритма это значит доказать утверждение вида:**

**«Если входные данные удовлетворяют входному условию, то алгоритм через конечное число шагов завершает работу, и выходные данные удовлетворяют требуемому выходному условию».**

На практике, такое утверждение часто разбивают на два.

(1) «Если входные данные удовлетворяют входному условию и алгоритм через конечное число шагов завершает работу, то выходные данные удовлетворяют требуемому выходному условию».

(2) «Если входные данные удовлетворяют входному условию, то алгоритм через конечное число шагов завершает работу».

Алгоритм, для которого доказано утверждение (1) называется частично правильным или частично корректным. Если же доказаны утверждения (1) и (2) то алгоритм называется правильным или корректным.

Заметим, что когда доказательство утверждения (2) представляет непреодолимые трудности, то ограничиваются доказательством утверждения (1). Таковы, например, итерационные алгоритмы, для которых не известна область сходимости. В таком случае, если алгоритм в приемлемое время завершает свою работу, то правильность ответа гарантируется.

Остановимся на доказательстве частичной корректности. **Методика заключается в следующем.**

1. Для контроля над ходом вычислений выбираются, так называемые, контрольные дуги. К числу контрольных обязательно относят входную и все выходные дуги, а также некоторое количество других дуг так, чтобы в граф-схеме алгоритма оказались «разрезанными» все циклы.

2. Для каждой контрольной дуги формулируется индуктивное условие, которому предположительно должно удовлетворять содержимое памяти алгоритма при каждом его прохождении через рассматриваемую дугу. Считаем, что все контрольные дуги, в дальнейшем будем называть их контрольными точками, и соответствующие им индуктивные утверждения пронумерованы.

3. Для каждой пары i, j контрольных точек, для которых в блок-схеме имеется путь из i в j, минующий другие контрольные точки, выбираются все такие пути и для каждого выбранного пути доказывается утверждение (индуктивный шаг): «Если при очередном проходе через точку i выполнялось индуктивное предположение Pi и если реализуется рассматриваемый путь, то при достижении точки j будет выполняться условие Pj ».

Если все индуктивные шаги доказаны, то, используя принцип математической индукции, можно утверждать частичную корректность алгоритма. Для доказательства полной корректности остается доказать завершаемость программы через конечное число шагов.

**Методы доказательства правильности программ**

К методам проверки правильности программ относятся:

1) методы доказательство правильности программ;

2) верификация и аттестация программ.

Методы доказательства правильности программ, появились в 80–е годы, делятся на два класса:

1. Точные методы доказательства правильности программ.

2. Методы доказательства частичной правильности программ.

Наиболее известными точными методами доказательства программ являются метод рекурсивной индукции или индуктивных утверждений Флойда и Наура и метод структурной индукции Хоара и др. Эти методы основываются на утверждениях и пред и пост–условиях.

Сущность формального доказательства заключается в преобразовании кода программы к логической структуре.

Чтобы доказать, что программа корректная, необходимо последовательно расположить все части, которые начинаются с А1 и заканчиваются Аend. Последовательность этих частей определяет, что истинность входного условия обеспечивает истинность выходного условия. После идентификации всех частей проверяется истинность каждой части программы с утверждением, что входные утверждения являются следствием выходного утверждения, которые отвечают преобразованиям ее частей. Доказательство программы завершено.

Формальные методы тесно связаны с математическими техниками спецификаций, верификацией и доказательством правильности программ. Эти методы содержат математическую символику, формальную нотацию и аппарат вывода. Правила доказательства являются громоздкими и поэтому на практике редко используются рядовыми программистами. Однако с теоретической точки зрения они развивают логику применения математического метода индукции при проверке правильности программ. На основе спецификации программ проводится частичное и полное доказательство правильности программ.

Под доказательством частичной правильности понимается проверка выполнения свойств данных программы с помощью утверждений, которые описывают то, что должна получить эта программа, когда закончится ее выполнение в соответствии с условиями заключительного утверждения. Полностью правильной программой по отношению к ее описанию и заданным утверждениям будет программа, если она частично правильная и заканчивается ее выполнение при всех данных, удовлетворяющих ей.

Для доказательства частичной правильности используется метод индуктивных утверждений, сущность которого состоит в следующем. Пусть утверждение A связано с началом программы, B - с конечной точкой программы и утверждение C отражает некоторые закономерности значений переменных, по крайней мере, в одной из точек каждого замкнутого пути в программе (например, в циклах). Если при выполнении программа попадает в i -ю точку и справедливо утверждение Ai, а затем она проходит от точки i к точке j, то будет справедливо утверждение *Aj*.

**Теорема 1**. Если выполнены все действия метода индуктивных утверждений для программы, то она частично правильна относительно утверждений A, B, C.

Требуется доказать что, если выполнение программы закончится, то утверждение B будет справедливым. По индукции, при прохождении точек программы, в которых утверждение C будет справедливым, то и n -я точка программы будет такой же. Таким образом, если программа прошла n -точку и утверждения A и B справедливы, то тогда, попадая из n -ой точки в n+1 точку, утверждение An+1 будет справедливым, что и требовалось доказать.

**1. Характеристика формальных методов доказательства**

Наиболее известными формальными методами доказательства программ являются метод рекурсивной индукции или утверждений Флойда, Наура, метод структурной индукции Хоара и др.

**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 доказательства ее правильности.

Данный метод доказательства уменьшает число ошибок и время тестирования программы, обеспечивает отработку спецификаций программы на полноту, однозначность и непротиворечивость.

Как известно, универсальные вычислительные машины могут быть запрограммированы для решения самых разнородных задач - в этом заключается одна из основных их особенностей, имеющая огромную практическую ценность. Один и тот же компьютер, в зависимости от того, какая программа находится у него в памяти, способен осуществлять арифметические вычисления, доказывать теоремы и редактировать тексты, управлять ходом эксперимента и создавать проект автомобиля будущего, играть в шахматы и обучать иностранному языку. Однако успешное решение всех этих и многих других задач возможно лишь при том условии, что компьютерные программы не содержат ошибок, которые способны привести к неверным результатам.

Можно сказать, что требование отсутствия ошибок в программном обеспечении совершенно естественно и не нуждается в обосновании. Но как убедиться в том, что ошибки, в самом деле, отсутствуют? Вопрос не так прост, как может показаться на первый взгляд.

К неформальным методам доказательства правильности программ относят отладку и тестирование, которые являются необходимой составляющей на всех этапах процесса программирования, хотя и не решают полностью проблемы правильности. Существенные ошибки легко найти, если использовать соответствующие приемы отладки (контрольные распечатки, трассировки).

Тестирование – процесс выполнения программы с намерением найти ошибку, а не подтвердить правильность программы. Суть его сводится к следующему. Подлежащую проверке программу неоднократно запускают с теми входными данными, относительно которых результат известен заранее. Затем сравнивают полученный машиной результат с ожидаемым. Если во всех случаях тестирования налицо совпадение этих результатов, появляется некоторая уверенность в том, что и последующие вычисления не приведут к ошибочному итогу, т.е. что исходная программа работает правильно.

С интуитивной точки зрения программа будет правильной, если в результате ее выполнения будет достигнут результат, с целью получения которого и была написана программа. Сам по себе факт безаварийного завершения программы еще ни о чем не говорит: вполне возможно, что программа в действительности делает совсем не то, что было задумано. Ошибки такого рода могут возникать по различным причинам.

В дальнейшем мы будем предполагать, что обсуждаемые программы не содержат синтаксических ошибок, поэтому при обосновании их правильности внимание будет обращаться только на содержательную сторону дела, связанную с вопросом о том, достигается ли при помощи данной программы данная конкретная цель. Целью можно считать поиск решения поставленной задачи, а программу рассматривать как способ ее решения. Программа будет правильной, если она решит сформулированную задачу.

Метод установления правильности программ при помощи строгих средств известен как верификация программ.

В отличие от тестирования программ, где анализируются свойства отдельных процессов выполнения программы, верификация имеет дело со свойствами программ.

В основе метода верификации лежит предположение о том, что существует программная документация, соответствие которой требуется доказать. Документация должна содержать:

* спецификацию ввода-вывода (описание данных, не зависящих от процесса обработки);
* свойства отношений между элементами векторов состояний в выбранных точках программы;
* спецификации и свойства структурных подкомпонентов программы;
* спецификацию структур данных, зависящих от процесса обработки.

К такому методу доказательства правильности программ относится метод индуктивных высказываний, независимо сформулированный К. Флойдом и П. Науром.

Суть этого метода состоит в следующем:

1) формулируются входное и выходное высказывания: входное высказывание описывает все необходимые входные условия для программы (или программного фрагмента), выходное высказывание описывает ожидаемый результат;

2) предполагая истинным входное высказывание, строится промежуточное высказывание, которое выводится на основании семантики операторов, расположенных между входом и выходом (входным и выходным высказываниями); такое высказывание называется выведенным высказыванием;

3) формулируется теорема (условия верификации):

из выведенного высказывания следует выходное высказывание;

4) доказывается теорема: доказательство свидетельствует о правильности программы (программного фрагмента).

Доказательство проводится при помощи хорошо разработанных математических методов, использующих исчисление предикатов первого порядка.

Условия верификации можно построить и в обратном направлении, т.е., считая истинным выходное высказывание, получить входное высказывание и доказывать теорему: из входного высказывания следует выведенное высказывание.

Такой метод построения условий верификации моделирует выполнение программы в обратном направлении. Другими словами, условия верификации должны отвечать на такой вопрос: если некоторое высказывание истинно после выполнения оператора программы, то, какое высказывание должно быть истинным перед оператором?

Построение индуктивных высказываний помогает формализовать интуитивные представления о логике программы. Оно и является самым сложным в процессе доказательства правильности программы. Это объясняется, во-первых, тем, что необходимо описать все содержательные условия, и, во-вторых, тем, что необходимо аксиоматическое описание семантики языка программирования.

Важным шагом в процессе доказательства является доказательство завершения выполнения программы, для чего бывает достаточно неформальных рассуждений.

Таким образом, алгоритм доказательства правильности программы методом индуктивных высказываний представляется в следующем виде:

1) Построить структуру программы.

2) Выписать входное и выходное высказывания.

3) Сформулировать для всех циклов индуктивные высказывания.

4) Составить список выделенных путей.

5) Построить условия верификации.

6) Доказать условие верификации.

7) Доказать, что выполнение программы закончится.

Этот метод сравним с обычным процессом чтения текста программы (метод сквозного контроля). Различие заключается в степени формализации.

Преимущество верификации состоит в том, что процесс доказательства настолько формализуем, что он может выполняться на вычислительной машине. В этом направлении в восьмидесятые годы проводились исследования, даже создавались автоматизированные диалоговые системы, но они не нашли практического применения.

Для автоматизированной диалоговой системы программист должен задать индуктивные высказывания на языке исчисления предикатов. Синтаксис и семантика языка программирования должны храниться в системе в виде аксиом на языке исчисления предикатов. Система должна определять пути в программе и строить условия верификации.

Основной компонент доказывающей системы - это построитель условий верификации, содержащий операции манипулирования предикатами, алгоритмы интерпретации операторов программы. Вторым компонентом системы является подсистема доказательства теорем.

Отметим трудности, связанные с методом индуктивных высказываний. Повторим, что трудно построить «множество основных аксиом, достаточно ограниченное для того, чтобы избежать противоречий, но достаточно богатое для того, чтобы служить отправной точкой для доказательства высказываний о программах». Вторая трудность - семантическая, заключающаяся в формировании самих высказываний, подлежащих доказательству. Если задача, для которой пишется программа, не имеет строгого математического описания, то для нее сложнее сформулировать условия верификации.

Перечисленные методы имеют одно общее свойство: они рассматривают программу как уже существующий объект и затем доказывают ее правильность.

Метод, который сформулировали К. Хоар и Э. Дейкстра основан на формальном выводе программ из математической постановки задачи

**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.

**Метод Маккарти** состоит в структурной проверке функций, работающих над структурными типами данных, изменяет структуры данных и диаграммы перехода во время символьного выполнения программ. Эта техника включает в себя моделирование выполнения кода с использованием символов для изменяемых данных. Тестовая программа имеет входное состояние, данные и условия ее выполнения.

Выполняемая программа рассматривается как серия изменений состояний. Самое последнее состояние программы считается выходным состоянием и если оно получено, то программа считается правильной. Данный метод обеспечивает высокое качество исходного кода.

**Метод Дейкстры** предлагает два подхода к доказательству правильности программ. Первый подход основан на модели вычислений, оперирующей с историями результатов вычислений программы, анализом путей прохождения и правил обработки большого объема информации. Второй подход базируется на формальном исследовании текста программы с помощью предикатов первого порядка. В процессе выполнения программа получает некоторое состояние, которое запоминается для дальнейших сравнений.

Основу метода составляет математическая индукция, абстрактное описание программы и ее вычисление. Математическая индукция применяется при прохождении циклов и рекурсивных процедур, а также необходимых и достаточных условий утверждений. Абстракция позволяет сформулировать некоторые количественные ограничения. При вычислении на основе инвариантных отношений проверяются на правильность границы вычислений и получаемые результаты.

Процесс формального доказательства правильности программ методом математической индукции зарекомендовал себя как система правил статической проверки правильности программ за столом для обнаружения в них формальных ошибок. С помощью этого метода можно доказать истинность некоторого предположения P(n) в зависимости от параметра n для всех n ≥ n0, и тем самым доказать случай P(n0). Исходя из истинности P(n) для любого значения n, доказывается P(n+1), что достаточно для доказательства истинности P(n) для всех n ≥ n0. Путь доказательства следующий. Пусть даны описание некоторой правильной программы (ее логики) и утверждение A относительно этой программы, которая при выполнении достигает некоторой определенной точки. Проходя через эту точку n раз, можно получить справедливость утверждения A(n), если индуктивно доказать, что:

1. A(1) справедливо при первом проходе через заданную точку,
2. если A(n) справедливо при n проходах через заданную точку, то справедливо и A(n+1) прохождение через заданную точку n+1 раз.

Исходя из предположения, что программа в конце концов успешно завершится, утверждение о ее правильности будет справедливым.

**ВЫВОДЫ:**

Ком­пь­ю­тер яв­ля­ет­ся ло­ги­че­с­ки не­об­ра­ти­мым, ес­ли его фун­кция пе­ре­хо­да не об­ла­да­ет одно­знач­ной об­рат­ной фун­кци­ей. Ло­ги­че­с­ки не­об­ра­ти­мый ком­пь­ю­тер всег­да мож­но сде­лать об­ра­ти­мым, со­хра­няя всю ин­фор­ма­цию, ко­то­рая бы в дру­гих слу­ча­ях те­ря­лась. Универсальной машиной Тью́ринга называют [машину Тьюринга](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3%D0%B0), которая может заменить собой любую машину Тьюринга. Традиционные методы анализа ПО связаны с доказательством правильности программ (верификация программ). Доказать правильность алгоритма это значит доказать утверждение вида: «Если входные данные удовлетворяют входному условию, то алгоритм через конечное число шагов завершает работу, и выходные данные удовлетворяют требуемому выходному условию».