
**Министерство образования и науки Украины**
**Севастопольский национальный технический университет**

# МЕТОДИЧЕСКИЕ УКАЗАНИЯ

**к выполнению лабораторных работ по дисциплине «Технология проектирования программных систем » для студентов специальности 7.091501 «Компьютерные системы и сети» дневной формы обучения**

**Севастополь**
**2009**

УДК 681.3.068(075)

Методические указания к выполнению лабораторных работ по дисциплине «Технология проектирования программных систем» /Сост. В.Ф.Ротко. – Севастополь: Изд-во СевНТУ, 2009. – 48 с.

Целью методических указаний является оказание помощи студентам, выполняющим лабораторные работы по дисциплине «Технология проектирования программных систем». Методические указания ориентированы на пояснение сложных принципиальных понятий и методов технологического подхода к проектированию программных систем.

Каждая тема лабораторной работы предваряется изложением основных теоретических положений, необходимых для выполнения лабораторной работы. К каждой теме прилагается список рекомендуемой литературы, а также список контрольных вопросов. Для каждой темы лабораторной работы предусмотрено средство выбора индивидуального задания.

Методические указания рассмотрены и утверждены на заседании кафедры КиВТ (протокол № 8 от 24.04. 2009 года).

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент:
Карлусов В.Ю., канд. техн. наук, доцент кафедры ИС.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Обеспечение надежности и эффективности программных систем является ключевым направлением в области создания программных систем. Предполагается, что решить эту проблему удастся только путем технологического подхода в течение целого жизненного цикла системы. Обучение студентов с целью подготовки их к профессиональной деятельности в этой чрезвычайно широкой (и плохо определенной) предметной области наталкивается на определенные трудности.

Во-первых, в настоящее время идет бурное и не всегда согласованное накопление знаний в этой сфере с неудовлетворительным взаимодействием между теорией и практикой.

Во-вторых, зафиксированное на данный момент деление на учебные дисциплины, имеющих отношение к области создания качественных программных систем, не вполне соответствует сложившемуся положению в этой области.

По инициативе следующих международных обществ: «Компьютерное общество института инженеров по электротехнике и электронике» (IEEE CS), «Ассоциация по вычислительной технике» (ACM) и «Ассоциация предприятий компьютерных и информационных технологий» (АП КИТ) были разработаны рекомендации по преподаванию программной инженерии в университетах [1].

В связи с тем, что область знаний, связанная с информационными технологиями («компьютинг»), очень сильно разрослась и ее трудно полностью осветить в рамках одного университетского курса, было принято решение о ее разделении на четыре основные дисциплины: информатика (computer science), программная инженерия (software engineering), проектирование аппаратных платформ (hardware engineering) и информационные системы (information systems).

Дисциплина «Технология проектирования программных систем» (ТППС) выделена с некоторыми нарушениями системного подхода, который программную систему должен трактовать как целостное образование, существующее в рамках его «жизненного цикла». Кроме того, не существует единственной универсальной технологии проектирования любой программной системы.

При ограниченном бюджете учебного времени, выделенного для дисциплины ТППС, возможная конкретная организация содержательного материала дисциплины представлена в подготовленном учебном пособии [2].

Настоящие методические указания предназначены для оказания помощи студентам при выполнении лабораторных работ, предусмотренных в рамках изучения дисциплины ТППС. Описание каждой лабораторной работы состоит из указания цели ее проведения, основных теоретических положений, описания средств выбора варианта задания, списка контрольных вопросов по тематике лабораторной работы и рекомендуемой литературы.

Каждый вариант задания на выполнение лабораторной работы характеризуется кодом задания, представляющим последовательность

значений $v_{m-1} = v(A_{m-1}),\ v_{m-2} = v(A_{m-2}),\ ...,\ v_0 = v(A_0)$ атрибутов $A_{m-1},\ A_{m-2},\ ...,\ A_0$, и порядковым номером $V$ варианта задания в десятичной системе счисления. Студенту сообщается порядковый номер варианта $V$ задания, по которому он должен найти код варианта. По коду варианта задания описывается содержательный смысл задания.

Фактически код задания представляет собой запись числа в обобщенной позиционной системе счисления, для которой справедливы следующие веса позиций: $w_i = w_{i-1} \cdot c_{i-1},\ w_0 = 1,$ где $w_i = w(A_i),\ c_i = c(A_i)$ - мощность множества $D_i$=$D(A_i)$ возможных значений атрибута $A_i : D_i = \{0,1,...,c_i - 1\}$. Порядковый номер $V$ определяется следующим соотношением: $V = \sum_{i=0}^{m-1} v_i \cdot w_i$ .

Для нахождения кода по заданному порядковому номеру $V$ варианта задания необходимо последовательно находить значения $v_0, v_1, ..., v_{m-1}$, положив $V_0$=$V$ при $i$=0 и вычисляя $v_i = V_i \mod c_i,\ V_{i+1} = \lfloor V_i / c_i \rfloor$. Здесь $\lfloor V_i / c_i \rfloor$ - целая часть от $V_i / c_i$ .

Пусть $m$=5, $c_0$=2, $c_1$=3, $c_2$=2, $c_3$=2, $c_4$=3 и указан порядковый номер 57 варианта задания (нумерация вариантов начинается с нуля). Для рассматриваемого примера число возможных вариантов равно 2·3·2·2·3=72(0…71). Варианту с порядковым номером 57 соответствует код <20111>.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Рекомендации по преподаванию программной инженерии и информатики в университетах = Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering; Computing Curricula 2001: Computer Science. – М.: ИНТУИТ.РУ «Интернет - Университет Информационных Технологий», 2007. – 462 с.
2. Ротко В.Ф. Технологии проектирования программных систем: учеб. пособие / В.Ф.Ротко, А.В. Скатков. – Севастополь: Изд-во СевНТУ, 2006. – 309 с.

# ЛАБОРАТОРНАЯ РАБОТА № 0

# СПЕЦИФИКАЦИЯ ПРОГРАММ И ПОДГОТОВКА ТЕСТОВ ДЛЯ ОТЛАДКИ И ПРОВЕРКИ

Цель работы: демонстрация студентом знаний и умений систематического подхода к разработке, отладке и контролю программы путем тестирования.

**Основные теоретические положения**

Разработка программ тесно связана с тем, что принято называть спецификацией программы. Спецификация программы – это исходное или промежуточное (на пути к программе) описание задачи, которую должна решать проектируемая программа, приводя в действие вычислительную машину [1]. Создание спецификации тоже называется спецификацией, так что спецификация – это и вид деятельности, и результат.

Программа – задание для вычислительной машины, написанное программистом. Спецификация – задание для программиста постановщиком задачи или системным аналитиком. Часто программист выступает как в роли постановщика задачи, так и в роли разработчика программы. Понятие спецификации относительно и многоаспектно. В принципе спецификация должна быть описанием задачи наиболее простым и естественным образом. Спецификация – точное, однозначное, недвусмысленное описание. Написать спецификацию – значит, прежде всего, подобрать точные понятия, адекватные задаче. Круг математических понятий, используемых в спецификациях, гораздо шире того, что используется в программах. Огромный опыт и культура обращений с точными понятиями, накопленные в математике, применимы в разработке спецификаций в большей мере, чем в разработке программы. К сожалению, многие программисты пытаются обойти этап спецификации программы.

Традиционный и привычный для программиста способ организации понятийных средств – это язык с фиксированным синтаксисом и семантикой. В спецификациях от понятийных средств, в первую очередь требуется богатство и гибкость изобразительных средств, а не эффективная машинная реализация.

Доведение программы до работоспособного состояния обычно производится путем выполнения отладки с использованием отладочных тестов. Затем производится контроль программы с использованием системы независимых тестов. Желательно, чтобы разработку программы и ее контроль производили разные лица. Современная практика тестирования программ (как правило) базируется в основном на квалификации и интуиции разработчиков и независимых контролеров (аудиторов). Внедрение в практику методик и средств автоматизации систематического тестирования сопряжено с большими не только техническими, но и психологическими трудностями. Опытные программисты предпочитают использовать свои собственные приемы тестирования, полученные в результате длительной практической работы.

В общем случае практически невозможно создать единственный универсальный метод тестирования и приходится использовать множество различных методов, ориентированных на различные критерии тестирования. Очевидно, что применимость того или иного критерия тестирования существенно

связана со сложностью программы. Мы ограничимся рассмотрением программ небольшой сложности, принимая во внимание их учебный характер.

**Краткая характеристика основных методов тестирования**

К настоящему времени известно очень много методов и методик тестирования программ [2, 3, 4]. Обычно при классификации методов тестирования выделяют классы последовательных и параллельных программ. Последний класс характерен для систем реального времени. Кроме этого, методы тестирования принято классифицировать по их области применимости, характеризуемой сложностью программ. Принято выделять класс программных модулей и класс программных комплексов. Естественно разделить методы тестирования на классы в соответствии с принятым основополагающим принципом тестирования. В качестве такого принципа выбирают понятия детерминизма и недетерминизма [2]. В соответствии с эти принципом различают систематическое и статистическое тестирование программ. Областью наших интересов будет только систематическое тестирование, в первую очередь, последовательных программ.

**Систематическое тестирование**

Тестирование по методу черного ящика

Этот метод характеризуется тем, что тесты выбираются только на основе спецификации программы [3]. Для последовательной программы тест представляет собой пару: входное состояние программы, ожидаемое выходное состояние программы. Для программ реального времени первый элемент этой пары в общем случае представляет собой временную последовательность входных состояний программы.

Обычно различают «положительное» и «отрицательное » тестирования. При «положительном» тестировании тесты выбираются в соответствии со спецификацией программы в порядке убывания (невозрастания) вероятностей возможных входных состояний программы. Совокупность тестов должна быть представительной настолько, насколько это возможно с точки зрения функциональной спецификации. Целью «отрицательного» тестирования является проверка реакции программы на необычные и даже непредусмотренные события.

**Иерархическое тестирование**

Тестирование может проводится различными путями: от «корня» иерархической системы к «листьям», от «листьев», иерархической системы к «корню». При этом можно пользоваться различными стратегиями обхода элементов иерархической системы: в глубину, в ширину.

Тестирование от «листьев» иерархической системы к «корню» принято называть восходящим тестированием. Восходящее тестирование подразделяют на несколько стадий: модульное тестирование, сборочное тестирование, системное тестирование.

Модульное тестирование обеспечивается путём создания тестовой среды, позволяющей генерировать тесты как на основе информации о спецификации модуля, так и на основе информации о его структуре.

Сборочное тестирование характеризуется особенностями тестовой среды для проверки элементов более высокого уровня иерархии.

Системное тестирование характеризуется особенностями тестовой среды для элемента высшего уровня иерархии.

Тестирование от «корня» иерархической системы к «листьям» принято называть нисходящим тестированием. Этот подход характеризуется тем, что тестирование ведётся по мере разработки. Модули более низкого уровня иерархии в этом случае имитируются посредством заглушек.

Опыт разработки сложных иерархических программ показывает, что целесообразно использовать совместно и нисходящее и восходящее тестирования в соответствии с ограничениями на временные, людские и финансовые ресурсы проекта.

Если по результатам тестирования в программу вносятся коррекции, требуется проводить повторное тестирование. В зависимости от типа изменений проводят тестирование на ранее выбранной совокупности тестов, а также на разработанных новых дополнительных тестах. Тестирование на ранее выбранной совокупности тестов называют регрессионным тестированием. Для систем реального времени проведение регрессионного тестирования оказывается в большинстве случаев затруднительным в силу изменения временны*ы*х ограничений, последовательности событий и т.д.

**Тестирование по методу прозрачного ящика**

Этот метод тестирования предполагает доступность информации о структуре программы. Структура программы в рамках одной из возможных моделей достаточно адекватно может быть представлена сетью в некотором смысле подобной сети Петри [5]. Эта сеть отображает два аспекта программы императивной модели: поток управления и поток данных. Условимся множество узлов сети, связанных с отображением данных, обозначать через P, а множество узлов, связанных с отображением преобразований данных, - через T. С множествами P и T связаны «веса» W(P) W(T), которые идентифицируют данные и преобразования соответственно. В этом случае программа может быть описана ориентированной сетью N (P,T,D(P,T),C(T),W(P),W(T)). Здесь D(P,T) представляет дуги двудольной частичной сети, C(T) представляет дуги подсети сети $N$. Будем также полагать, что для всех $t \in T$ дуги $A(t) \subseteq C(T)$ упорядочены в соответствии с заданием $W(T)$ в виде условного выражения Мак Карти $(\rho 1 \to O1,\ \rho 2 \to O2,..., \rho n \to On)$ [6]. Здесь $\rho 1, \rho 2,...\rho n$ - конечное число предикатов ($n$>0), $O1$, $O2$, …, $On$ – имена конечного числа любых объектов. Семантика этого выражения может быть записана в виде $(!x)(\exists_i)[\rho i \wedge (\forall j)(j < i \to \neg \rho j) \wedge x = Oi]$.

Видно, что значение такого выражения – это имя первого (считая слева направо) объекта $Oi(0 < i <= n)$, которому соответствует истинное высказывание.

Дуги $A^{-1}(t) \subseteq D(P,T)$ упорядочены в соответствии с порядком аргументов, зафиксированном в $W(t)$.

При графическом представлении $N$ узлы из $P$ будем обозначать кружками, а узлы из T – «полочками». Дуги из $C(T)$ будем выделять жирными линиями. На рисунке 1 приведен пример такого описания программы.

Один из методов, используемых при тестировании на основе доступной структуры программы, учитывает только поток управления. Будем полагать, что программа имеет только одну точку входа $s$ и одну точку выхода $f$.

Эта модель представляется адекватной для программ, решающих задачи преобразования исходных данных, описывающих задачу, в результирующие

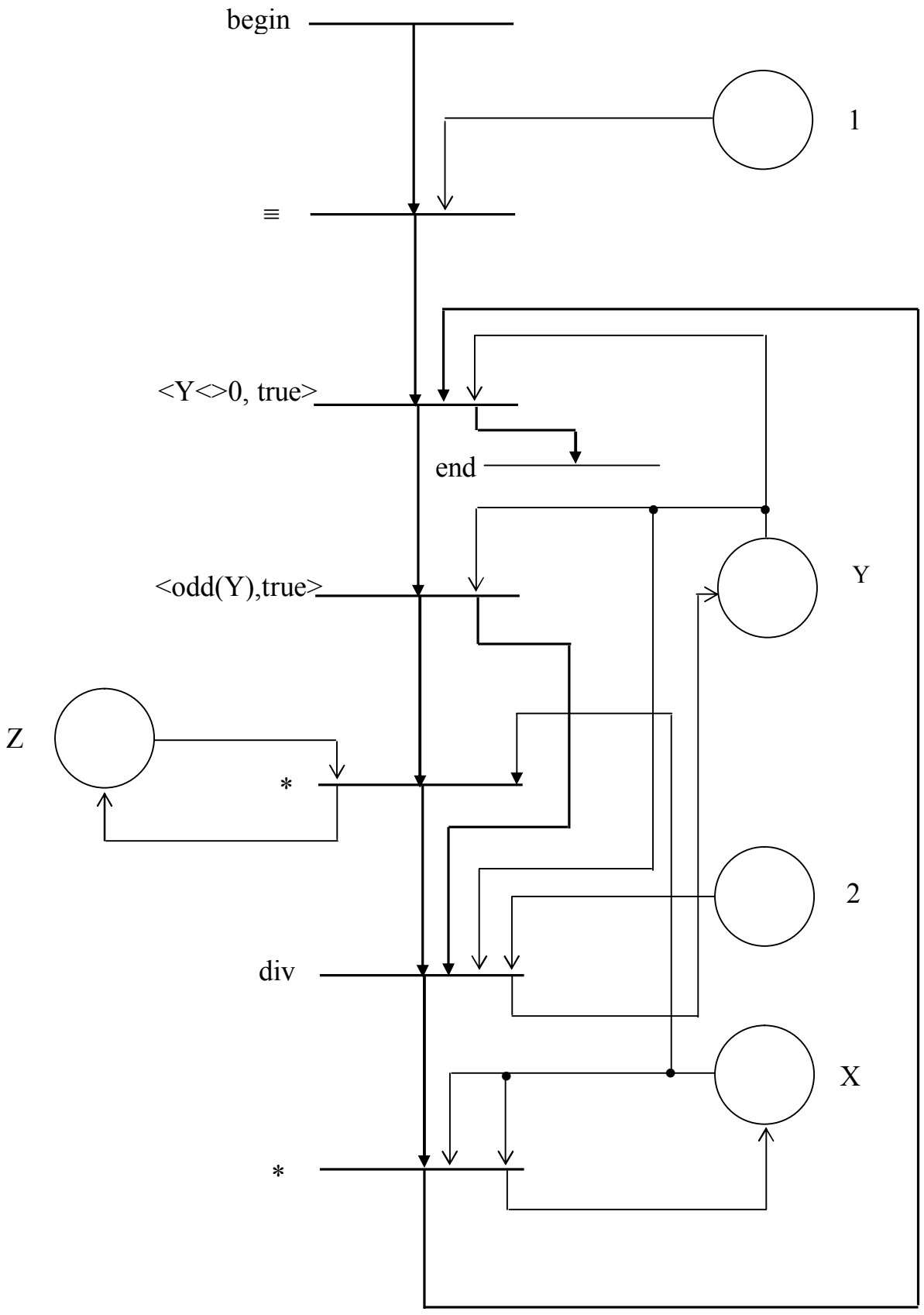


Рисунок 1 – Возведение X в натуральную степень Y ($Z=X^Y$).
Структурная схема программы

данные, представляющие решение задачи. Другую разновидность программ представляют так называемые реагирующие системы [7] или реактивные системы, для которых применяют специфичные языки спецификации, основанные на временны*ы*х логиках, и методы проверки соответствия системы спецификации, известные под названием Model Checking [7].

Пусть $X$ и $Y$ описывают домены исходных и результирующих данных, зафиксированные в декларативной области программы. В множестве $X$, представляющем область отправления программы, выделяют $D \subseteq X$, для каждого элемента $d \in D$ которого выполняется следующее условие: при задании $d$ в точке входа $s$ программа завершает работу в точке выхода $f$. В этом случае реализуется $(s,f)$ – маршрут в графе управления программы. Естественно считать $D$ областью определения программы, а область $X \backslash D$ – областью, где программа не определена (бесконечный циклический маршрут, маршрут, завершающийся в точке, отличной от $f$). Будем полагать, что программа реализует функцию $F$ такую, что $R = F(D)$, $R \subseteq Y$. Область $Y$ естественно назвать областью прибытия программы, а $R$ – областью значений программы.

В общем случае при работе программы используется также домен Z вспомогательных данных, который декларируется в программе. Универсальный домен программы U=X×Y×Z.

Множеству синтаксических $(s,t)$ – маршрутов поставим во взаимно однозначное соответствие разбиение множества $D$, полагая, что в разбиении могут быть «различимые» пустые множества.

**Путев*о*е тестирование**

Целью путев*о*го тестирования является выполнение каждого реализуемого маршрута в программе путем выбора подходящего множества тестов. Такая цель практически нереализуема, так как большинство программ представляют огромное число маршрутов. Естественно ограничиться рассмотрением только представительного множества маршрутов, опираясь на некоторые критерии тестирования. Простейшие критерии основываются на покрытии маршрутами вершин графа управления ($C_0$), дуг графа управления ($C_1$), пар входящих и выходящих дуг для каждой вершины графа управления ($C_2$) и т.д. Эти критерии могут быть обобщены на случай полного графа программы (описывающего также поток данных).

Условие, описывающее множество данных, при которых реализуется один и тот же $(s,t)$ – маршрут, можно найти методом «обратного прослеживания». Это условие часто называют путев*ы*м предикатом. Подобный метод используется при формировании условий верификации корректности последовательной программы [8]. Для решения задачи выбора подходящего множества тестов можно воспользоваться также методом, известным как символьное выполнение [1].

Каждый путевой предикат можно трактовать как описание уравнения $P(x)=1$ относительно исходных данных $x$. Часто путевой предикат может быть описан в виде системы уравнений и неравенств.

Наличие неосуществимых синтаксических $(s,t)$ – маршрутов осложняет решение проблемы выбора тестов на основе путев*ы*х предикатов. Путев*ы*е

предикаты могут быть столь сложными, что доказательство отсутствия решения не может быть проведено эффективным образом.

Запуски программы на тестовых данных, соответствующих путевым предикатам (или слабейшим посылкам) выполняют проверки только по отношению к этим слабейшим посылкам. Соответствие же самих слабейших посылок по отношению к спецификации должно быть доказано отдельно.

Если тестирование проводится на большой совокупности тестов, то желателен метод для автоматической оценки правильности выходных данных, так называемый «оракул». Такие оценки могут выполняться сравнением выходных данных тестируемой программы с выходными данными «эталонных» программ, или путем проверки истинности выходного предиката, входящего в спецификацию программы.

Следует заметить, что путевое тестирование принципиально не решает проблемы проверки программы при ошибках, связанных с упущением некоторых маршрутов.

Метод «обратного прослеживания» рассмотрим для простейших программ, использующих только скалярные переменные из множества $X = \{x_1, x_2, ..., x_n\}$, операторы присваивания вида $x_* := f(x_1 x_2, ..., x_n),\ x_* \in X$ и условные операторы вида $< Q(x_1 x_2, ..., x_n) \to m_1,\ \text{true} \to m_0 >$.

Пусть $(s,t)$ – маршрут вычислений имеет вид $M = s, O_1, O_2 ..., O_m, t$, где $O_i$ – $i$-ый оператор. Прохождение маршрута через условный оператор с условием $Q(x_1 x_2, ..., x_n)$ через метку $m_1$ будем обозначать как $Q(x_1 x_2, ..., x_n)$, а через метку $m_0$ – как $\neg Q(x_1 x_2, ..., x_n)$.

Пусть искомое «слабейшее предусловие», связанное с маршрутом М, есть $\rho(x_1 x_2, ..., x_n)$.

Алгоритм построения $\rho(x_1 x_2, ..., x_n)$ может быть представлен в виде следующей последовательности шагов:

1. Положить $\rho(x_1 x_2, ..., x_n) = \text{true}$.
2. Просматривать М, начиная с $t$, до $s$.

   Если встречается условный оператор $Q(x_1 x_2, ..., x_n)$ при прохождении через метку $m_1$, то положить $\rho(x_1 x_2, ..., x_n) = \rho(x_1 x_2, ..., x_n) \,\&\, Q(x_1 x_2, ..., x_n)$, иначе $\rho(x_1 x_2, ..., x_n) = \rho(x_1 x_2, ..., x_n) \,\&\, \neg Q(x_1 x_2, ..., x_n)$.

   Если встречается оператор «действия» $x_* := f(x_1 x_2, ..., x_n)$, положить $\rho(x_1, x_2, ..., x_*, ..., x_n) = \rho(x_1, x_2, ..., f(x_1, x_2, ..., x_n), ..., x_n)$.

Рассмотрим программу, структурная схема которой приведена на рисунке 1. Возьмем маршрут

$\text{begin}\,(z := 1)(y <> 0)\neg\text{odd}(y)(y := y \,\text{div}\, 2)(x := x * x)(y <> 0)$
$\text{odd}(y)(z := z * x)(y := y \,\text{div}\, 2)(x := x * x)\neg(y <> 0)\ \text{end}$

Имеем следующую последовательность построения «слабейшего предусловия»:

$\rho = \text{true}$
$\rho = \text{true} \,\&\, \neg(y <> 0)$
$\rho = \text{true} \,\&\, \neg(y \,\text{div}\, 2 <> 0)$

$\rho$ = true & $\neg$ ($y$ div 2 <> 0) & odd( $y$) & ($y$ <> 0)

$\rho$ = true & $\neg$ (($y$ div 2)div 2 <> 0) & odd($y$ div 2) & ($y$ div 2 <> 0) & $\neg$ odd($y$) & ($y$ <> 0)

Легко показать, что предикат $\rho(y)$ принимает значение true при $y$=2.

**Символическое выполнение**

Символическое выполнение естественно назвать методом «прямого прослеживания», при котором значения переменных обозначаются символами. Условимся имена переменных обозначать заглавными буквами, а символические значения переменных обозначать соответствующими строчными буквами.

Символическое выполнение основано на операции подстановки вместо символов символьных выражений. Проиллюстрируем символическое выполнение на примере простой программы (фрагменте программы на языке Паскаль). Пусть фрагмент программы на языке Паскаль имеет следующий вид.

Read ln($A$, $B$) :
$C := A - B$;
if $C < 0$ then $D := -C$ else $D := 2 * C$;
$C := D + C - B$;
Write ln($C$);

Результат символического выполнения может быть представлен следующим образом.

$A = a, B = b$;
$C = a - b$;
if $(a - b) < 0$ then $D = -(a - b)$ else $D = 2 * (a - b)$;
*if* $(a - b) < 0$ then
$C = -(a - b) + (a - b) - b = -b$
else
$C = 2 * (a - b) + (a - b) - b = 3 * a - 4 * b$
*if* $(a - b) < 0$ then write ln$(-b)$ else
writeln$(3 * a - 4 * b)$

При заданном маршруте процесс символического выполнения упрощается и его естественно применять при нахождении преобразования, связанного с маршрутом.

Иллюстрацию «прямого и обратного прослеживания» удобно проводить в форме таблицы. После завершения «прямого прослеживания» выходные переменные будут содержать описание функциональных связей с входными переменными. «Обратное прослеживание» позволит найти «слабейшее предисловие», описывающее область определения преобразований.

**Мутационный анализ**

Метод мутационного анализа применяется к почти «правильной программе». Сущность этого метода состоит в трансформации программы путём внесения локальных изменений текста программы. Эти изменения производятся с учётом реально допускаемых ошибок в процессе программирования. Затем выполняются обычные тесты, и результаты сравниваются с результатами тестирования программы до изменений. При выполнении тестов все внесённые ошибки должны быть обнаружены. Если они не обнаруживаются, то необходимы дополнительные тесты. Опыт показывает, что при таком подходе подобранные тесты обладают способностью обнаружения не только внесенных, но и исходных ошибок.

Этот прием может использоваться для демонстрации полезности и полноты набора тестов. Следует заметить, что выбор подходящих мутаций является трудно формализуемым процессом.

Рассмотренные подходы к организации систематического тестирования наталкиваются на определенные трудности при усложнении структур данных. Эти трудности проявляются уже при использовании массивов. Смена парадигмы структурного проектирования и программирования на объектно-ориентированную не решает радикально проблему разработки корректного программного обеспечения. Тем не менее, определенные достижения в области тестирования современного программного обеспечения существуют [3,4,9]. Следует однако отметить, что они в основном представляют собой практические рекомендации без теоретического обоснования.

Определенные успехи имеются при разработке надежных программных систем с использованием формальных и формализованных методов [7,10].

Лабораторная работа № 0 преследует цель формирования у студента систематического подхода к спецификации и тестированию программ в роли разработчика и независимого контролера (аудитора) «чужих» программ. При формировании заданий на выполнение лабораторной работы № 0 приняты следующие атрибуты:

$A_0$, $A_1$ – указывают варианты разрабатываемой и контролируемой программы:

0 – шейкер-сортировка [11, с. 85];
1 – сортировка Шелла [11, с. 87];
2 – быстрая сортировка [11, с. 96];
3 – пирамидальная сортировка [11, с. 91];
4 – внешняя сортировка простым слиянием [11, с. 108];
5 – внешняя сортировка естественным слиянием [11, с. 115];
6 – внешняя сортировка сбалансированным многопутевым слиянием [11, с. 122];
7 – внешняя многофазная сортировка [11, с. 128];
8 – распределение начальных серий с помощью пирамиды [11, с. 141];
9 – анализатор контекстно-свободных языков методом Эрли [12, с. 47].

$A_2$ – указывает критерий тестирования программ:

0 – критерий $C_0$ (покрытие вершин графа управления);
1 – критерий $C_1$ (покрытие дуг графа управления);
2 – критерий $C_2$ (покрытие пар входящей и выходящей дуг для каждой вершины графа управления).

Вариант задания определяется как число, записанное в пространстве атрибутов $< A_2\ A_1\ A_0 >$. Варианты заданий лежат в интервале 0…299. Так вариант 279 =<279> требует проведения разработки анализатора контекстно-свободных языков методом Эрли и независимого контроля (аудита) программы внешней многофазной сортировки при использовании критерия тестирования $C_2$.

Контрольные вопросы

1. Что понимают под спецификацией программы?

2. Какими особенностями характеризуются тестирование, верификация и валидация программной системы?
3. Какие основополагающие принципы классификации методов тестирования программных систем?
4. Какие Вам известны методы тестирования программных систем?
5. Как выбрать тест, обеспечивающий прохождение заданного маршрута в программе?
6. Какие основные трудности организации систематического тестирования?
7. Какие принципиальные трудности тестирования программ реального времени?
8. Какие основные направления обеспечения надежности программы?

## СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Агафонов В.Н. Спецификация программ: понятийные средства и их организация /В.Н. Агафонов. – Новосибирск: Наука СО, 1990. – 224 с.
2. Проверка и утверждение программ реального времени /Под ред. У.Д. Квирка. – К.: Наука. думка, 1990. – 216 с.
3. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем /Б. Бейзер. – СПб.: Питер, 2004. – 318 с.
4. Макгрегор Д. Тестирование объектно-ориентированного программного обеспечения /Д. Макгрегор, Д. Сайкс. – К.: ООО «ТИД ДС», 2002. – 432 с.
5. Петерсон Д. Сети Петри и моделирование систем /Д. Петерсон. – М.: Мир, 1988. – 264 с.
6. Оллонгрен А. Определение языков программирования интерпретирующими автоматами /А. Оллонгрен. – М.: Мир, 1977. – 288 с.
7. Кларк Э.М. Верификация моделей программ: Model Checking /Э.М. Кларк, О. Грамберг, Д. Пелед. – М.: МЦНМО, 2002. – 416 с.
8. Непомнящий В.А. Прикладные методы верификации программ /В,А. Непомнящий, О.М. Рякин. – М.: Радио и связь, 1988. – 256 с.
9. Калбертсон Р. Быстрое тестирование /Р. Калбертсон, К. Браун, Г. Кобб. – М.: ИД «Вильямс», 2002. – 384 с.
10. Мейер Б. Объектно-ориентированное конструирование программных систем /Б. Мейер. – М.: ИТД «Русская Редакция», 2005. – 1232 с.
11. Вирт Н. Алгоритмы + структуры данных = программы /Н. Вирт. – М.: Мир, 1985. – 406 с.
12. Языки и автоматы. Сборник переводов под редакцией А.Н. Маслова и Э.Д. Стоцкого. – М.: Мир, 1975. – 358 с.

# ЛАБОРАТОРНАЯ РАБОТА № 1

## НАХОЖДЕНИЕ ФУНКЦИИ ПУТИ ПРОГРАММЫ

Цель работы: освоение методов нахождения описания функции, реализуемой при прохождении предписанного пути (маршрута) в программе. Описание функции представляет собой пару <D, F>, где D – описывает область определения функции, F – описывает зависимость выходных переменных от входных переменных.

**Основные теоретические положения**

Алгоритмическая реализация некоторой функции $f$ (в общем случае не полностью определенной) с областью отправления $U$ и областью определения $D \subseteq U$ представляет собой разложение на множество функций $\Phi = \{f(tr) | tr \in T_r\}$, ортогональных по своим областям определения $D$(tr). Здесь $t_r$ – маршрут в графе управления алгоритма, $T_r$ – множество ($s$, $t$)-маршрутов (маршрутов, которые начинаются в $s$ и заканчиваются в $t$). Очевидно, что $D = \underset{t_r \in T_r}{U} D(t_r)$. Функция $f$ не определена на множестве $U \setminus D$. Таким образом, каждому ($s$, $t$)-маршруту $i$ можно сопоставить описание функции $< D_i, F_i >$. Маршруты, для которых $D_i = \varnothing$, будем называть нереализуемыми.

При организации систематического тестирования желательно уметь находить описание функции, связанной с предписанным ($s$, $t$)-маршрутом алгоритма, которую принято было называть функцией пути программы. Описание функции пути программы может быть найдено методом «обратного» и «прямого» прослеживания.

Проблема характеризации реализуемости маршрута оказывается достаточно сложной даже при скалярных переменных. Положение усугубляется, если при описании алгоритма (программы) используются сложные структуры данных.

Выполнение лабораторной работы № 1 связано с разработкой не только программы построения описания функции пути, но также программ генерации ($s$, $t$)-маршрутов и упрощения логических (и функциональных) выражений. В программу упрощения логических выражений рекомендуется включить наиболее часто используемые средства эквивалентных преобразований.

Отладку программы построения описания функции пути (маршрута) рекомендуется проводить путем задания исходных данных для исследуемой программы и фиксации пройденного при этих данных маршрута. Задавая полученный маршрут в качестве исходных данных для программы построения описания функции пути (маршрута), ожидаем получить логическое выражение, истинное при соответствующих исходных данных исследуемой программы (и, возможно, также для других исходных данных).

Проиллюстрируем нахождение функций путей для следующих двух маршрутов программы, структурная схема которой приведена на рисунке 1.

Маршрут 1: begin $(z := 1)(y <> 0)\neg\text{odd}(y)(y := y \text{ div } 2)(x := x * x)$
$(y <> 0)\text{odd}(y)(z := z * x)(y := y \text{ div } 2)(x := x * x)\neg(y <> 0)$
end

Маршрут 2: begin $(z := 1)(y <> 0)\text{odd}(y)(z := z * x)(y := y \text{ div } 2)$

$(x := x * x)(y <> 0)\neg\text{odd}(y)(y := y \text{ div } 2)(x := x * x)\neg(y <> 0)$
end

С целью компактности описания функции пути $n$-кратное применение операции div 2 будем обозначать $\text{div}^n$ 2, $n$>1. Тогда для маршрута 1 имеем следующее описание функции пути:

$$< \neg(y \text{ div}^2 2 <> 0) \& \text{odd}(y \text{ div } 2) \& (y \text{ div } 2 <> 0) \& \neg\text{odd}(y) \& (y <> 0), z = x * x > .$$

Область определения функции пути для маршрута 1 в явном виде описывается условием $Y$=2. Мощность этой области в пространстве аргументов $X$ и $Y$ фактически определяется мощностью домена для $x$.

Для маршрута 2 имеем следующее описание функции пути

$$< \neg(y \text{ div}^2 2 <> 0) \& \neg\text{odd}(y \text{ div } 2)(y \text{ div } 2 <> 0) \& \text{odd}(y) \& (y <> 0), z = x > .$$

Анализируя предикат, описывающий область определения функции пути, приходим к выводу, что он описывает пустое множество. Следовательно, маршрут 2 является нереализуемым.

Задание на выполнение лабораторной работы № 1 определяется двумя атрибутами $A_0$ и $A_1$.

Атрибут $A_0$ указывает номер варианта исследуемой программы. Программа представлена в виде процедуры на языке Паскаль.

$A_0$ =0: «быстрое» возведение в натуральную степень, вариант 0.

```
Procedure PXN0 (x: real; n: word; var p: real);
var
y: real; m: word;
begin
y: = x; m: =n; p: =1;
while (m<>0) do begin
   if odd (m) then p: =p*y;
   m: =m div 2; y: = y*y
   end
end
; {PXN0}
```

$A_0$ =1: «быстрое» возведение в натуральную степень, вариант 1.

```
Procedure PXN1 (x: real; n: word; var p: real);
var
y: real; m: word;
begin
y: = x; m: =n; p: =1;
if (m<>0) then repeat
   if odd (m) then p: = p*y; m: =m div 2;
   if (m<>0)  then y: = y*y
   until (m=0)
end
; {PXN1}
```

$A_0$ =2: «быстрое» возведение в целую степень, вариант 0.

```
Procedure PXZ0 (x: real; z: integer; var p: real);
var
y: real; m: integer;
begin
y: = x; p: =1;
if (z<0) then m:=-z else m: =z;
while (m<>0) do begin
   if odd (m) then p: =p*y;
   m: =m div 2; y: = y*y  end;
if (z<0) then p: =1/p
end
; {PXZ0}
```

$A_0$ =3: «быстрое» возведение в целую степень, вариант 1.

```
Procedure PXZ1 (x: real; z: integer; var p: real);
var
y: real; m: integer;
begin
y: = x; p: =1;
if (z<0) then m:=-z else m: =z;
if (m<>0) then repeat
   if odd (m) then p: =p*y; m: =m div 2;
   if (m<>0) then y: =y*y
until (m=0);
if (z<0) then p: =1/p
end
; {PXZ1}
```

$A_0$ =4: нахождение наибольшего общего делителя двух натуральных чисел, вариант 0.

```
Procedure HGD0 (x, y: word; var d: word);
var
a: word;
begin
d: = x; a: =y;
repeat
     while (a>d) do a:=a-d;
     while (d>a) do d:=d-a
until (a=d)
end
; {HGD0}
```

$A_0$ =5: нахождение наибольшего общего делителя двух натуральных чисел, вариант 1.

```
Procedure HGD1 (x, y: word; var d: word);
var
   r: word;
begin
   d: = x; r: =y;
   while (r<>0) do begin
   d:=d+r; r: =d-r; d:=d-r; r: =r mod d
   end
end
; {HGD1}
```

$A_0$ =6: нахождение наибольшего общего делителя двух целых чисел, вариант 0.

```
Procedure HGDZ0 (x, y: integer; var d: integer);
var
a: integer;
begin
   if (x<0) then d:=-x else d:=x;
   if (y<0) then a:=-y else a: =y;
repeat
   while (a>d) do a: =a-d;
   wrile (d>a) do d:=d-a
until (a=d)
end
; {HGDZ0}
```

$A_0$ =7: нахождение наибольшего общего делителя двух целых чисел, вариант 1.

```
Procedure HGDZ1 (x, y: integer; var d: integer);
var
   r: integer;
begin
   if (x<0) then d:=-x else d:=x;
   if (y<0) then r:=-y else r: =y;
   while (r<>0) do begin
   d:=d+r; r: =d-r; d:=d-r; r: =r mod d
end
end
; {HGDZ1}
```

$A_0$ =8: нахождение наименьшего общего кратного двух натуральных чисел, вариант 0.

```
Procedure LGKN0 (x, y: word; var k: longint);
var
a,d:  word;
begin
d: = x; a: =y;
repeat
     while (a>d) do  a:= a-d;
```

```
    while (d>a) do d:=d-a
until (a=d)
if  (d<>0) then k: =x*y div d else k: =0
end
; {LGKN0}
```

$A_0$ =9: нахождение наименьшего общего кратного двух натуральных чисел, вариант 1.

```
Procedure LGKN1 (x, y: word; var k: longint);
var
r,d: word;
begin
d: = x; r: =y;
   while (r<>0) do  begin
d:= d+r; r:=d-r;  d:=d-r; r:=r mod d
end;
if  (d<>0) then k: =x*y div d else k: =0
end
; {LGKN1}
```

$A_0$ =10: «расширенный» алгоритм Евклида для нахождения $d$=НОД($a$,$b$) и $x$,$y$, для которых $ax+by=d$, вариант 0 [1].

```
Procedure AE0 (a, b: word; var d: word; var x,y: integer);
var
c,q,r: word; t,u,v: integer;
begin
c: =a; d: =b; x:=0; y: =1; u: =1; v: =0;
q: =c div d; r:=c mod d;
   while (r<>0) do  begin
c:=d; d:=r;
t:=u; u:=x; x:=t-q*x;
t:=v; v:=y; y:=t-q*y;
q: =c div d; r:=c mod d;
end
end
; {AE0}
```

$A_0$ =11: «расширенный» алгоритм Евклида для нахождения $d$=НОД($a$,$b$) и $x$,$y$, для которых $ax+by=d$, вариант 1.

```
Procedure AE1 (a, b: word; var d: word; var x,y: integer);
var
c,q,r: word; t,u,v: integer;
begin
c: =a; d: =b; x:=0; y: =1; u: =1; v: =0;
q: =c div d; r:=c mod d;
```

```
   while (r<>0) do  begin
c:=d; d:=r;
t:=v; v:=y; y:=t-q*y;
t:=u; u:=x; x:=t-q*x;
q: =c div d; r:=c mod d
end
end
; {AE1}
```

$A_0$ =12: нахождение приближенного значения корня уравнения $f(x)$=0 с заданной точностью $e$ для монотонно возрастающей на интервале $[a,b]$ функции $f(x)$ [2].

```
Procedure Root (a, b, e: real; var r: real);
var
h: real;
begin
r: =a; h: =b-a;
   while (h>e) and (f(r)<>0) do  begin
h:=h/2;
if  (f(r)>0) then r: =r-h else r: =r+h
end
end
; {Root}
```

Замечание. Предполагается, что функция $f(x)$ задается в виде соответствующей подпрограммы-функции.

$A_0$ =13: нахождение приближенного значения корня уравнения $f(x)$=0 с заданной точностью $e$ для на интервале $[a,b]$ методом дихотомии.

```
Procedure DICH (a, b, e: real; var r: real var p: boolean);
var
v, w: real;
begin
p: =true;
if  (f(a)*f(b)<0) then begin
     v:=a; w:=b; r:=(v+w)/2;
   while ((w-r) >e) do  begin
   if  (f(r)*f(v)<0) then  w:=r else v:=r;
   r:=(v+w)/2
end
end
; {DICH}
```

Замечание. Предполагается, что функция $f(x)$ задается в виде соответствующей подпрограммы-функции.

$A_0$ =14: бинарный поиск минимального значения функции [3]

```
Procedure DMGE (a, b, d, e: real; var c, fc: real);
var
x, y, s0, s1, f0, f1: real;
begin
   x: =a; y: =b;
   repeat
   c:=(x+y)/2; s0:=c-d; s1:=c+d;
   f0:=f(s0); f1:=f(s1);
   if  (f0<f1) then y:=s1 else x: =s0
   until ((c-x)/(1+abs (c))<e);
   fc:=f(c)
end
; {DMGE}
```

$A_0$ =15: Поиск минимума функции методом Фибоначчи при заданном ограничении на число вычислений функции [3]

```
Procedure FMN (a, b: real; n: nat46; var x, y, s0, f0: real);
const
r5=sqrt(5); la=ln((1+r5)/2);
var
fn1, fn2: longint; c: nat46; h, s1, f1: real
begin
x: =a; y: =b; c: =n; fn1:=round (exp(la*(n-1)/r5);
fn2:=round (exp(la*(n+2)/r5); h:=fn1/fn2;
s0:=x*h+y*(1-h); f0:=f(s0); c:=c-1;
if (c>0) then repeat
   c:=c-1;
if (f0<f1) then begin
y: =s1; s1:=s0; s0:=x+y-s1; f1:=f0; f0:=f(s0)
end else begin
x: =s0; s0:=s1; s1:=x+y-s0; f0:=f1; f1:=f(s1)
end
until (c=0)
end
; {FMN}
```

$A_0$ =16: Поиск минимума функции методом «золотого сечения» [3]

```
Procedure GCE (a, b, e: real; var s0, f0: real);
var
x, y, h, s1, f1: real;
begin
x: =a; y: =b; h:=(sqrt(5)-1)/2;
s0:=x*h+y*(1-h);  s1:=x+y- s0; f0:=f(s0); f1:=f(s1);
while ((y-x)/(1+abs(s0))>e) do
if (f0<f1) then begin
y: =s1; s1:=s0; s0:=x+y-s1; f1:=f0; f0:=f(s0)
```

```
end else begin
x: =s0; s0:=s1; s1:=x+y-s0; f0:=f1; f1:=f(s1)
end
end
; {GCE}
```

$A_0$ =17: разделение одномерного массива относительно некоторого его элемента d[4]

```
Procedure PARMAS (n: word; d: integer; var A: mas1i; var l, r: word);
var
   t: integer;
begin
  l:=1; r:=n;
  repeat
  while A[l]<d do l:=l+1;
  while A[r]>d do r:=r-1;
  if l<=r then  begin
  t:= A[l]; A[l]:=A[r]; A[r]:=t;
  l:=l+1; r:=r-1;
end
until (l>r);
  l:=l-1; r:=r+1
end
; {PARMAS}
```

$A_0$ =18: поиск в одномерном массиве *k*–го по порядку элемента[4]

```
Procedure FINDOK (k, n: word; var A: mas1i; var l, r: word);
var
d, i, j, t: integer;
begin
l:=1; r:=n;
while (l<r) do begin
d:=A[k]; i:=l; j:=r;
repeat
while (A[i]<d) do i:=i+1;
while (A[i]>d) do j:=j-1;
if (i<=j) then begin
t:=A[i]; A[i]:=A[j]; A[j]:=t; i:=i+1; j:=j-1
end
until (i>j);
if (j<k) then l:=i;
if (i>k) then r:=j
end
end
:{FINDOK}
```

$A_0$ =19: поиск в одномерном массиве медианы [4]

```
Procedure FINMED (n: word; var A: mas1i; var l, r: word);
var
d, i, t: integer; k: word;
begin
l:=1; r:=n; k:=n div 2+1;
while (l<r) do begin
d:=A[k]; i:=l; j:=r;
repeat
while (A[i]<d) do i:=i+1;
while (A[i]>d) do j:=j-1;
if (i<=j) then begin
t:=A[i]; A[i]:=A[j]; A[j]:=t; i:=i+1; j:=j-1
end
until (i>j);
if (j<k) then l:=i;
if (i>k) then r:=j
end
end
; {FINMED}
```

Атрибут $A_1$ указывает вариант ограничения на максимальную длину $l_{max}$ маршрута, для которого обязательно должно быть найдено описание функции пути {$A_1$ = 0: $l_{max}$ =100; $A_1$=1: $l_{max}$ =200}.

Контрольные вопросы

1. Как может быть описана программная реализация некоторой функции?
2. Что понимают под функцией пути?
3. Как описывается функция пути?
4. Как построить описание области определения функции пути?
5. Как построить описание преобразования, связанного с функцией пути?
6. С какими принципиальными трудностями связано решение проблемы построения описания функции пути?
7. Какие принципиальные трудности связаны с решением задачи характеризации реализуемости пути (маршрута)?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Алагич С. Проектирование корректных структурированных программ / С.Алагич, М. Арбиб. - М.: Радио и связь, 1984. – 264 с.
2. Непомнящей В.А. Прикладные методы верификации программ/ В.А.Непомнящий, О.М.Рякин.- М.: Радио и связь, 1988.- 256 с.
3. Амосов А. А. Вычислительные методы для инженеров / А.А. Амосов, Ю.А. Дубинский, Н.В.Копченова.-М.: Высшая школа, 1994.- 544с.
4. Вирт Н. Алгоритмы + структуры данных = программы / Н.Вирт. - М.: Мир, 1985.- 406 с.

# ЛАБОРАТОРНАЯ РАБОТА № 2

## ПОСТРОЕНИЕ СИСТЕМЫ ТЕСТОВ, УДОВЛЕТВОРЯЮЩЕЙ ЗАДАННОМУ КРИТЕРИЮ

Цель работы: освоение методов систематической подготовки системы тестов, удовлетворяющей заданному критерию. Рассматриваются только структурные критерии $C_0$, $C_1$ и $C_2$.

**Основные теоретические положения**

Структурные критерии тестирования $C_0$, $C_1$ и $C_2$ основаны на понятии покрытия вершин, дуг и пар дуг (входящих и выходящих для каждой вершины) графа управления программы, реализуемыми $(s,t)$-маршрутами.

Представляется целесообразным с критериями $C_0$, $C_1$ и $C_2$ ассоциировать покрытия множества вершин, множества пар вершин и триад вершин графа управления программы соответственно.

Под $(s,t)$-маршрутом в графе управления программы будем понимать последовательность вершин, начинающуюся с вершины s и заканчивающуюся вершиной t, с выполнением требования допустимости переходов. Множество синтаксических маршрутов может быть описано на языке регулярных выражений.

Развернутое изложение метода выбора маршрутов (путей) программы для построения тестов можно найти в работе [1].

Рассмотрим кратко основные понятия и задачи, связанные с проблемой выбора совокупности тестов, удовлетворяющей некоторому критерию. В общем случае критерий может быть условным. Структурные критерии $C_0$, $C_1$ и $C_2$ будем рассматривать как безусловные. При выполнении этих критериев будем полагаться на критерий минимальной совокупности тестов, затем на критерий минимальной длины маршрутов (по каждом тесту), затем критерий минимизации представления данных (по каждому тесту).

Отношение числа покрытых элементов структуры программы согласно заданному критерию тестирования к числу всех элементов принято называть степенью тестированности. Полная система тестов (ПСТ) соответствует степени тестированности 100%. Покрывающим множеством путей (ПМП) называют систему путей, содержащих в совокупности все элементы структуры программы согласно заданному критерию тестирования программы. При построении ПМП обычно стремятся минимизировать число входящих в него путей.

Если в граф управления программы ввести «отмеченную» дугу $(t, s)$ , то проблема генерации синтаксических классов $(s,t)$-маршрутов (путей) может быть сведена к генерации циклических классов (множеств дуг), содержащих «отмеченную» дугу. Эта задача, в свою очередь, может быть решена путем нахождения базисного множества фундаментальных циклов [2].

Рассмотрим в качестве примера программу, структурная схема которой приведена на рисунке 1. Выделенный граф управления программы с введенной «отмеченной» дугой приведен на рисунке 2. Множество фундаментальных циклов имеет следующий вид: {(0 – 2,7), (2 – 6), (2, 3, 5, 6)}. В первый фундаментальный цикл входит «отмеченная» дуга <7, 0>.

Рассмотрим следующие циклы, содержащие «отмеченную» дугу <7, 0>: (0 – 2,7); (0, 1, (2 – 6), 7); (0,1,(2,3,5,6),7); (0, 1, (2 – 6), (2, 3, 5, 6), 7); (0, 1, (2, 3, 5, 6), (2 – 6), 7).

Очевидно, что последний цикл является покрытием для вершин и дуг (т.е. удовлетворяет критериям $C_0$ и $C_1$) и является реализуемым. Кратчайший маршрут реализуется при $y$=2. При критерии $C_2$ покрытием будет множество циклов {(0-2,7); (0, 1, (2, 3, 5, 6), (2 – 6), 7)}.

Возможны программы, для которых невозможно удовлетворить критерий тестирования $C_2$ (есть пары дуг, которые могут быть покрыты только чисто синтаксическими (нереализуемыми) маршрутами). В этом случае принимается «ослабленный» критерий $C^*_2$ путем исключения из требований покрытия подобного рода пар дуг (троек вершин) графа управления программы.

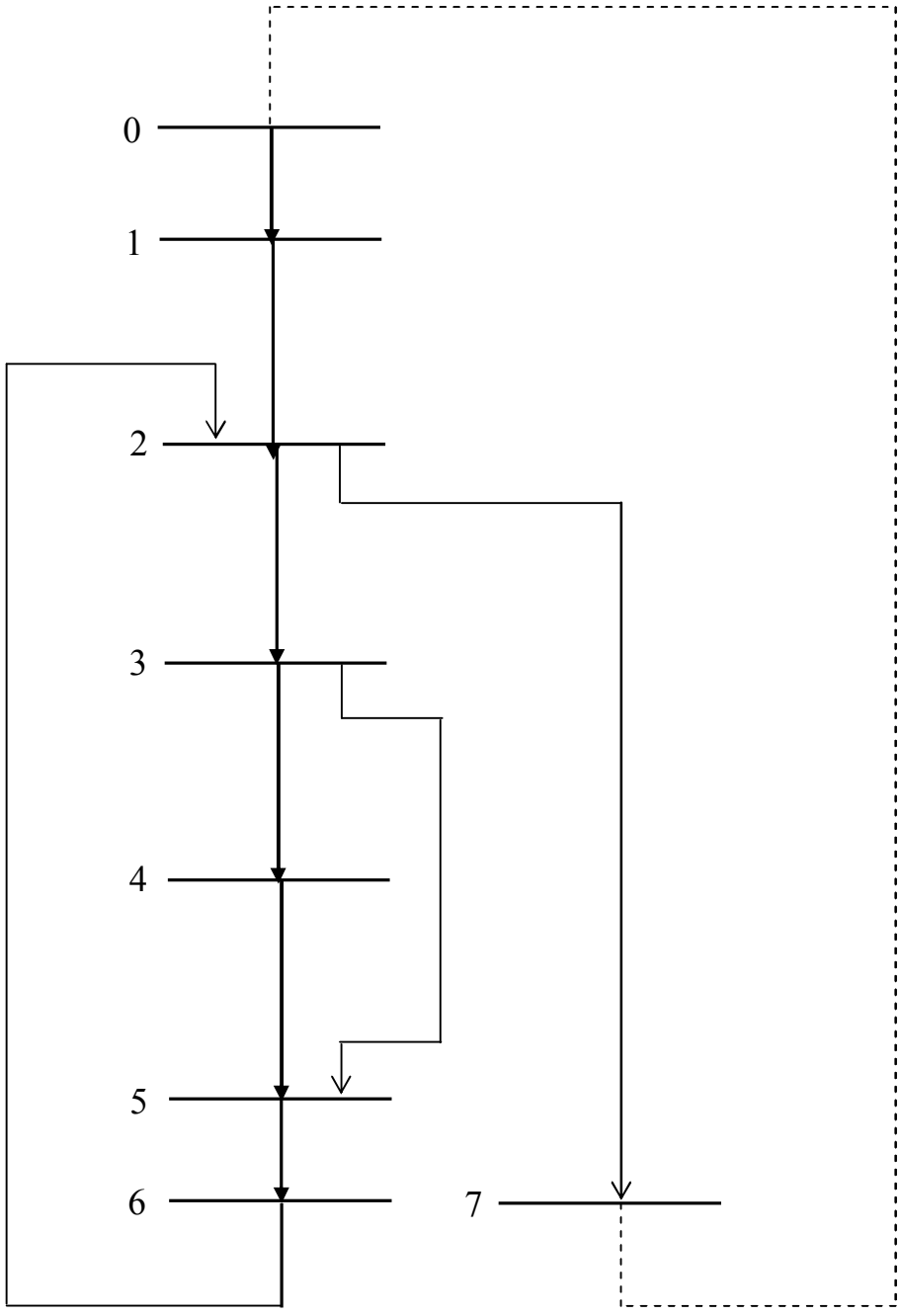


Рисунок 2 – Возведение $X$ в натуральную степень $Y(Z=X^y)$. Граф управления программы

Задание на выполнение лабораторной работы № 2 определяется двумя атрибутами $A_0$ и $A_1$. Атрибут $A_0$ указывает номер варианта исследуемой программы. Варианты программ приведены при описании лабораторной работы № 1. Атрибут $A_1$ указывает вариант структурного критерия тестирования {$A_1$ =0: критерий $C_0$; $A_1$ =1: критерий $C_1$; $A_1$ =2: критерий $C_2$}.

Контрольные вопросы

1. Что понимают под критерием тестирования $C_0$?
2. Что понимают под критерием тестирования $C_1$?
3. Что понимают под критерием тестирования $C_2$?
4. Что понимают под полной системой тестов?
5. Что принято называть покрывающим множеством путей?
6. Какие основные трудности стоят на пути нахождения полной системы тестов?
7. Какие методы поиска покрывающего множества путей Вы знаете?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. 1. Выбор путей программы для построения тестов /Ю.В. Борзов [и др.] – УСиМ, 1989. – № 6. – С.29-36.
2. 2. Кристофидес Н. Теория графов. Алгоритмический подход /Н. Кристофидес. – М.: Мир, 1978. – 432 с.
3.

# ЛАБОРАТОРНАЯ РАБОТА № 3

# АНАЛИЗ ЧАСТИЧНОЙ КОРРЕКТНОСТИ ПОСЛЕДОВАТЕЛЬНОЙ ПРОГРАММЫ

Цель работы: освоение простейших методов проверки частичной корректности последовательной программы

**Основные теоретические положения**

Верификация программ состоит в формальном доказательстве их корректности. Различают два подхода к верификации программ [1]. При аналитическом подходе полагается, что имеется программа и ее спецификация. Математическое доказательство корректности (правильности) программы состоит в выявлении соответствия логического поведения программы ее спецификации. При синтетическом (конструктивном) подходе основной акцент делается на корректную разработку и корректное конструирование (проектирование) программы.

Основная идея аналитического подхода основана на положении о том, что свойства программы могут быть строго исследованы вследствие того, что ее можно рассматривать как математический объект. Флойд Р. показал, что можно выделять инвариантные свойства программы относительно каждой точки структурной схемы программы и представить их в виде формальных утверждений. Чтобы установить соответствие программы ее спецификации необходимо доказать, что входное утверждение влечет выходное утверждение для всех возможных путей между начальной и конечной точками структурной схемы.

Одной из принципиальных трудностей является задание исходного описания функционирования программы и контроль его соответствия основному замыслу. Эта проблема здесь не рассматривается и вместо термина «правильность» предпочтение отдается термину «корректность».

При анализе (верификации) программы целесообразно выделить следующие понятия (свойства):

1) частичная корректность;
2) завершение.

Под частичной корректностью программы понимают соответствие функциональной спецификации при условии завершения выполнения программы. Под завершением понимают достижение в процессе выполнения программы выхода при условии допустимости входного состояния.

Каждое из указанных свойств корректности программы может удовлетворяться или не удовлетворяться. Обычно выделяют следующие основные задачи анализа корректности программы (последовательной):

1) частичная корректность (при условии завершения);
2) завершение программы;
3) незавершение программы;
4) полная корректность (частичная корректность и завершение);
5) частичная некорректность (некорректность при условии завершения);
6) некорректность (незавершение или частичная некорректность).

Верификация программ очень тесно связана с проблемой спецификации программ. Рассмотрим основные принципы логической спецификации свойств программ [1, 2, 3, 4].

Всякая программа реализует некоторую вычислимую функцию, поэтому под спецификацией программы будем, в первую очередь, понимать задание функции программы (функциональная спецификация).

Логическая спецификация функции $f$ описывается путем указания двух предикатов: предусловия pre-$f(d)$, характеризующего область определения функции $f$, и постусловия post-$f(d, r)$, характеризующего функциональное отображение относительно схемы функции $f$: D→R. Формальный смысл этих предикатов устанавливается на основе следующей их взаимосвязи: $\forall\, d(\text{pre-}f(d) \rightarrow \text{post-}f(d, f(d)))$.

Переходя от функции $f$ к программе, вычисляющей эту функцию, следует учитывать следующие особенности. Абстрактные множества D и R интерпретируются на множестве State состояний памяти компьютера. Понятие состояния st∈State является одним из важнейших понятий, вскрывающих содержательную сторону вычислений (семантику вычислений). Состояние можно представить в виде кортежа значений переменных программы $st=<st_1, st_{2,...,}st_m>$. Так как в разных состояниях могут быть доступны только переменные из некоторого подмножества, то, вообще говоря, множество значений переменной $x_i$ должно быть расширено путем включения элемента $\omega_i$, представляющего неопределенное значение. Функцию $f$ можно представить схемой $f$: State→State.

Предусловие и постусловие программы определяется с учетом дополнительного интенсионального свойства – завершение вычислений. Следует иметь в виду, что для некоторых начальных состояний st∈State выполнение программы может не завершиться (зациклиться), либо завершиться досрочно из-за невозможности выполнить некоторую операцию (данные не входят в ее область определения). Это отражает общую природу вычислимых функций как функций частично рекурсивных.

Таким образом, логическая спецификация программы Prgm может быть определена на State предикатами pre-Prgm: State→boolean и post-Prgm: State×State →boolean при условии ∀ st(pre-Prgm (st) & fin (Prgm (st))→post-Prgm (st, f(st))), где st∈State – начальное (входное) состояние Prgm; f(st)∈State – выходное (результат выполнения) состояние Prgm; fin(Prgm(st)) – предикат завершения выполнения Prgm.

Предикат pre-Prgm и post-Prgm характеризуют лишь вход и выход программы. Для более полной логической спецификации свойств программы могут потребоваться характеристики ее поведения во “внутренних” точках структуры. Эту характеристику обычно называют инвариантным утверждением или инвариантом.

Инвариантом сечения $c$ графа управления программы называют формулу логического языка спецификации $\psi_c(x_1, x_2, ..., x_n)$, такую, что $\psi_c$ истинно для набора значений программных переменных $(x_1, x_2, ..., x_n)$ всякий раз, когда выполнение Prgm достигает сечения $c$. Инвариантность утверждения означает независимость его истинности от пути, по которому достигается данное сечение. В сечениях начала и конца программы инвариантные утверждения задают предусловие и постусловие программы соответственно.

Требуется также формализм, характеризующий свойства оператора с точки зрения его воздействия на состояние памяти. Таким формализмом служит тройка Хоара {P}A{Q}, где P,Q – инварианты входа и выхода оператора A (предусловие и постусловие оператора A). Формально тройка Хоара определяет предикат {P}A{Q}: $\forall$st (P(st) & fin (A, st) $\rightarrow$Q(st, $f_A$(st))), где $f_A$ - функция, вычисляемая оператором A.

Тройка Хоара является элементарной единицей рассуждений о свойствах программ (утверждение об операторе) и играет важнейшую роль в задачах верификации. В соответствии с определением тройки Хоара, если оператор A не завершается, то тройка {P}A{Q} является истинной. При использовании троек Хоара оказываются выразимыми многие свойства программ. Свойство завершения является исключением. Интересно, что свойство зацикливания представимо в виде тройки Хоара {P}A {false}.

Примеры истинных троек Хоара: $\{x > 0\} x := x + 1 \{x > 1\}$; $\{x > 0\}$ while $x>0$ do $x:=x-1$ $\{x=0\}$.

Для троек Хоара справедливы следующие законы консеквенции (логического следования):

{P}A{Q}, Q $\rightarrow$V $\vdash$ {P}A{V};

{P}A{Q}, W $\rightarrow$P $\vdash$ {W}A{Q}.

Здесь V является ослаблением Q, а W – усилением P.

В качестве следствий из законов консеквенции можно получить {P}A {true} и {false}A{Q}. Здесь true – слабейшее постусловие, а false – сильнейшее предусловие.

Сильнейшее постусловие $Q^*$ для фиксированных предусловия P и оператора A удовлетворяет соотношению: {P}A{$Q^*$}, $\forall$({P}A{Q}) $\vdash$ $Q^* \rightarrow$ Q.

Слабейшее предусловие $P^*$ для фиксированных постусловия Q и оператора A удовлетворяет соотношению: {$P^*$}A{Q}, $\forall$({P}A{Q}) $\vdash$ P$\rightarrow$ $P^*$.

Слабейшее предусловие $P^*$ полностью характеризует множество исходных данных для заданных Q и A, а сильнейшее постусловие $Q^*$ – полную характеристику функции, вычисляемой A для области исходных данных, определяемой P.

Примеры. Пусть A есть $x := x + 1$ и $P : (x > 0)$. Тогда сильнейшее постусловие есть $Q^* : (x > 1)$, но не $Q^* : (x > 0)$. Если для того же оператора A задано $Q : (x > 1)$, то слабейшее предусловие есть $P^* : (x > 0)$.

Специфицировать программу – значит указать на достаточно высоком уровне абстракции, какими свойствами должно обладать ее выполнение. Часто спецификация связана с заданием утверждений о том, что должно быть истинно в соответствующих точках программы. Эти утверждения называют индуктивными утверждениями. Различие между спецификациями (индуктивными утверждениями) и инвариантами состоит в том, что индуктивные утверждения отражают лишь желаемые свойства, а инварианты – фактические.

Программа с записанными индуктивными утверждениями о ней называется аннотированной программой. Аннотированные программы можно рассматривать как некоторый специальный вид логических формул, способных принимать значения true и false. Фактически аннотированная программа – формула на лексиконе (язык программирования + логический язык спецификации). Такое ее

рассмотрение позволяет формализовать преобразование аннотированных программ и доказательство их свойств.

Рассмотрим теперь некоторые из методологических вопросов использования логического языка спецификации при спецификации свойств программ. Обычно возникающие здесь задачи разделяют на две группы:

1) разработка аксиоматических теорий проблемных областей;
2) формирование индуктивных утверждений для соответствующих точек программы в виде формул разработанной аксиоматической теории.

Прежде чем записывать формулы для тех или иных утверждений о программных объектах, необходимо выбрать подходящую систему понятий (функций) вместе с аксиомами, характеризующими семантику этих понятий. Для этого необходимо следовать некоторой методике, так как эта проблема (в достаточно общей постановке) не может быть решена алгоритмически.

При выборе системы понятий для спецификации следует стремиться к их естественности и выразительности. Число понятий, относящихся к одному уровню иерархии, должно быть ограничено. Здесь рекомендуется соблюдать известный в психологии принцип (закон Миллера), который утверждает, что число одновременно используемых объектов не должно превышать семи. Это справедливо и при выборе аксиом. Таким образом, выбор есть компромисс между выразительностью и сложностью понятий.

Основу используемых при спецификации понятий составляют стандартные (встроенные) функции базового языка программирования (сложение, умножение, деление, операции над строками, массивами и т.п.). Для них должны быть построены соответствующие аксиоматические теории. Это же относится и к специально вводимым функциям, характеризующим понятия проблемной области.

При работе с логическими спецификациями без использования машинных носителей нет необходимости жестко придерживаться формальной синтаксической структуры модулей спецификации. Если же предполагается использование машинных носителей, то целесообразно зафиксировать синтаксис языка спецификаций и семантику (интерпретацию). Обычно интерпретация включает:

1) задание используемых типов данных, сигнатуры и схем функций и предикатов;
2) задание семантики функций и предикатов.

Для аксиоматических теорий семантика задается системой аксиом, истинных в данной проблемной области и расширяющих набор логических аксиом базового исчисления предикатов. Описание аксиоматической теории на языке логической спецификации (LS) обычно подразделяют на синтаксическую и семантическую части. Синтаксическая часть определяет множество используемых типов, схемы и сигнатуры функций и предикатов. Типы выбираются среди типов данных базового языка программирования, а также конструируются средствами LS как абстрактные типы данных. Для описания сигнатуры привлекают известные мощные средства именования операций, функций, предикатов и их композиции. Это позволяет выражать понятия в естественной для рассматриваемой области форме.

При практической спецификации целесообразно выделить следующие группы понятий:

1) понятия, общие для различных областей, такие, как понятия булевского типа, арифметики целых чисел, теоретико-множественные понятия и т.п.

2) понятия, относящиеся ко всему классу задач данной области (например, сортировка данных), обработка файлов, обработка матриц, обработка символьной информации и т.п.
3) понятия, относящиеся к конкретной задаче из данной области (например, сортировка Шелла, поиск в файле и т.п.).

При разработке системы аксиом для вводимых понятий следует учитывать следующие основные требования:

1) непротиворечивость (т.е. невозможность вывода из аксиом некоторого утверждения и его отрицания);
2) полнота (т.е. определение свойств предметной области в пределах допустимой интерпретации).

Основным методом выявления непротиворечивости аксиом является интерпретация рассматриваемой аксиоматической теории в некоторой другой заведомо непротиворечивой аксиоматической теории. В качестве универсальной непротиворечивой теории может рассматриваться арифметика целых чисел.

Проверка полноты сводится к доказательству того, что система аксиом определяет в точности те свойства, которыми мы интуитивно наделяем характеризуемое понятие. Эта задача в общем случае является алгоритмически неразрешимой проблемой.

Рассмотрим теперь формальные постановки задач анализа корректности программ. Как и в задачах спецификации, условимся рассматривать программы «двухполюсники» (Start, Stop). Спецификация программы Prgm осуществляется приписыванием индуктивных утверждений сечениям графа управления программы. Рекомендуется «сечению» подвергать вершины графа управления, разделяя входящие и выходящие дуги. Будем полагать, что при «сечении» вершины $v$ образуются две вершины $v'$ и $v''$, причем в вершины $v'$ могут только входить дуги, а из вершины $v''$ могут только исходить дуги. При этом вершине Start$'$ приписывается входной предикат (предусловие) программы $P(x)$, определяющий множество допустимых значений исходных данных $x$, а вершине Stop$'$ - выходной предикат (постусловие) $Q(x, y)$, определяющий целевую функцию (т.е. желаемую связь между входными данными $x$ и выходными данными $y$).

Свойства частичной корректности программы Prgm определяется при этом следующей формулой логики предикатов: $\mathrm{PCOR}(\mathrm{Prgm}, P, Q): \forall x(P(x) \,\&\, \mathrm{fin}\,(x) \rightarrow Q(x, f(x)))$, где $\mathrm{fin}\,(x)$ – предикат завершения программы Prgm, начатой в состоянии $x$; $f(x)$ – функция, вычисляемая программой Prgm (выходное состояние в момент завершения вычислений). С использованием понятия тройки Хоара это свойство может быть записано в виде $\mathrm{PCOR}(\mathrm{Prgm}, P, Q): \{P\}\ \mathrm{Prgm}\ \{Q\}$. Для формализации свойства частичной корректности потребовалось использовать предикат завершения $\mathrm{fin}\,(x)$ в качестве вспомогательного понятия. Свойство завершения, таким образом, описывается в виде $\mathrm{FIN}(\mathrm{Prgm}, P): \forall x\ (P(x) \rightarrow \mathrm{fin}\,(x))$.

Свойство полной корректности формально выражается как одновременное наличие свойств частичной корректности и завершения: $\mathrm{TCOR}(\mathrm{Prgm}, P, Q): \forall x(P(x) \rightarrow \mathrm{fin}\,(x) \,\&\, Q(x, f(x)))$. Заметим, что разделение свойств частичной корректности и завершения следует рассматривать как методологический прием, направленный на уменьшение сложности соответствующих доказательств.

Представляется целесообразным ввести следующие обозначения троек Хоара для случаев частичной и полной корректности: $\{P.\}\,\mathrm{Prgm}\,\{Q\}$ и $\{P\}\,\mathrm{Prgm}\,\{.Q\}$.

В общей постановке (для произвольных программ) каждое из этих свойств является алгоритмически неразрешимым. Однако, для различных классов программ могут быть развиты достаточно экономичные методы анализа свойств корректности.

Метод индуктивных утверждений – наиболее известный метод доказательства частичной корректности программы. Он позволяет свести доказательство свойства частичной корректности программы к доказательству некоторого (конечного) числа утверждений, записанных в виде формул логического языка спецификации и имеющих интерпретацию в соответствующей предметной области. Этот метод составляет основу большинства разработанных автоматизированных систем верификации программ.

Метод индуктивных утверждений рассмотрим для программ с конечным множеством простых переменных $\{x_1, x_2, \ldots x_n\}$, $n > 0$, в которых допускается использование операторов присваивания вида $x_i := f(x_1, x_2, \ldots, x_i \ldots, x_n)$, условного перехода $if\, L(x_1, x_2, \ldots x_n)$ then $L_+$ else $L_-$, где $L_+$ и $L_-$ – метки операторов, которым передается управление, а также составного оператора вида A; B, где A и B – операторы. Такой состав структурных компонентов вполне достаточен для представления произвольного алгоритма и поэтому не ограничивает общности описываемого метода.

Очевидно, что при выполнении программы для различных исходных данных возможны различные последовательности операторов, начинающиеся Start и заканчивающиеся Stop. Такие последовательности часто называют трассами (маршрутами) вычислений.

Истинность тройки Хоара $\{P\}$ Prgm $\{Q\}$ имеет место, если истинны тройки Хоара $\{P_j\}T_j\{Q\}$ для всех трасс $\{T_j\}$, т.е. $\{P\}$ Prgm $\{Q\}$: $\forall j(\{P_j\}T_j\{Q\})$, где $\{P_j\}$ – предикат «вычисления выполняются по $j$-ой трассе $\{T_j\}$»; $T_j$ – трасса вычислений, соответствующая $P_j$.

Трассы вычислений осуществляют разбиение предусловия $P$ таким образом, что $P = \vee_j P_j$; $\forall j_1, j_2((j_1 \neq j_2) \rightarrow Pj_1 \,\&\, Pj_2 = \mathrm{false})$.

Сложность анализа $\{P\}$ Prgm $\{Q\}$ разбиением на трассы состоит в том, что при наличии циклических вычислений в программе он не может быть выполнен непосредственным перебором всех трасс. Выход состоит во включении в программу дополнительных индуктивных утверждений так, чтобы любой циклический маршрут проходил, по крайней мере, через одно такое утверждение. Индуктивное утверждение, приписанное циклу, принято называть инвариантом цикла. Этот термин не совсем корректен, так как инвариант цикла – фактическое (а не ожидаемое) утверждение.

Условимся считать, что программа аннотируется выбором контрольных точек на входе, выходе и на каждом циклическом пути программы. С каждой контрольной точкой $t_i$ ассоциируется индуктивное утверждение (инвариант), записываемое на логическом языке утверждений: $\mathrm{inv}_{t0}(x_1, x_2, \ldots, x_n) : P$; $\mathrm{inv}_{t1}(x_1, x_2, \ldots, x_n) : Q$; $\mathrm{inv}_{ti}(x_1, x_2, \ldots, x_n)$, $i > 1$, где $t_0$ – точка входа ((Start), $t_1$ – точка выхода (Stop), $t_i$, $i$>1 – контрольные точки инвариантов циклов.

Пусть $\alpha =< r, A_1; A_2; ...; A_k, t >$ – путь между соседними контрольными точками графа управления программы, а $U_\alpha$ - условие верификации частичной корректности программы относительно этого пути. Тогда условие $U_\alpha$ можно определить следующим образом: $U_\alpha : \text{inv}_r\ (x_1, x_2, ..., x_n) \to U_0$, где последовательность $U_k, U_{k-1}, ..., U_0$ задается по индукции (методом обратной подстановки) следующим образом:

1) $U_k : \text{inv}_t\ (x_1, x_2, ..., x_n)$;

2.1) если $A_i$ – оператор вида $x_s := f\ (x_1, x_2, ..., x_n)$, тогда $U_{i-1} : U_i\ (x_s \leftarrow f\ (x_1, x_2, ..., x_n))$, где $x_s \leftarrow f\ (x_1, x_2, ..., x_n)$ есть подстановка (замещение) $x_s$ термом $f\ (x_1, x_2, ..., x_n)$);

2.2) если $A_i$ - оператор условия $P_\varepsilon, \varepsilon \in \{+, -\}$, то $U_{i-1} : P \to U_i$ при $\varepsilon = +$, либо $U_{i-1} : \neg P \to U_i$ при $\varepsilon = -$.

Пример.

Пусть $\alpha =< r, x := f_1(x);\ q_+(x);\ x := f_2(x);\ h_-(x);\ x := f_3(x), t >$.

Тогда

$\text{P} : \text{inv}_\text{r}(x)$;

$\text{U}_5 : \text{Q} : \text{inv}_\text{t}(x)$;

$\text{U}_4 : \text{Q}(f_3\ (x))$;

$\text{U}_3 : (\neg(h(x)) \to (\text{Q}(f_3\ (x))))$;

$\text{U}_2 : (\neg(h(f_2(x))) \to (\text{Q}(f_3\ (x)))))$;

$\text{U}_1 : (q(x) \to (\neg(h(f_2(x))) \to (\text{Q}(f_3\ (f_2\ (x))))))$;

$\text{U}_0 : (q(f_1(x)) \to (\neg(h(f_2(f_1(x))))) \to (\text{Q}(f_3\ (f_2(f_1\ (x)))))))$.

Условие частичной корректности относительно пути $\alpha$ имеет вид:

$\text{U}_\alpha : P \to (q(f_1(x)) \to (\neg(h(f_2(f_1(x))))) \to (\text{Q}(f_3\ (f_2(f_1\ (x)))))))$.

Введенное понятие условия $U_\alpha$ между соседними контрольными точками позволяет свести задачу анализа частичной корректности программы Prgm к доказательству истинности условий $U_\alpha$ между любыми соседними контрольными точками программы Prgm. Контрольные точки *r* и *t* являются соседними, если в графе управления программы существует путь из точки *r* в точку *t*. Условия $U_\alpha$ называют условиями частичной корректности (или просто условиями верификации). Они являются формулами логического языка спецификации и их набор для всех путей между соседними контрольными точками полностью характеризует свойство частичной корректности. Возможность их доказательства полностью зависит от проблемной области, где интерпретируются эти формулы.

Итак, метод индуктивных утверждений включает в себя следующие основные этапы:

1) получение аннотированной программы заданием индуктивных утверждений для входа, выхода и инвариантов циклов программы в виде формул логического языка спецификаций;

2) определение набора условий корректности $U_\alpha$ для всех путей между соседними контрольными точками в виде формул логического языка спецификаций;
3) доказательство истинности условий корректности как теорем формальной теории соответствующей предметной области.

Проиллюстрируем метод индуктивных утверждений при доказательстве частичной корректности программы вычисления частного и остатка от деления натурального числа *x* на натуральное число *y*. Фрагмент программы имеет следующий вид:

*r*:=*x*; *q*:=0;
while *y*<=*r* do begin
*r*:= *r*-*y*; *q*:=*q*+1
end

Соответствующая аннотированная программа может быть представлена следующим образом:

{(*x*>=*y*)&(*y*>0)}
*r*:=*x*; *q*:=0;
while {(*x*=*r*+*y***q*} *y*<=*r* do begin
*r*:= *r*-*y*; *q*:=*q*+1
end
{(*x*=*r*+*y***q*} &(*r*<*y*)}

Рассмотрим следующие пути:

$\alpha_1$: от входа программы к началу цикла;
$\alpha_2$: от начала цикла к началу цикла;
$\alpha_3$: от начала цикла к выходу программы.

Методом обратной подстановки находим следующие условия путей:

$U_{\alpha_1} : (x >= 0) \,\&\, (y > 0) \to x = r + y * 0$;

$U_{\alpha_2} : (x = r + y * q) \to (y <= r) \to (x = r + y * q))$;

$U_{\alpha 3} : (x = r + y * q) \to ((y > r) \to ((x = r + y * q) \,\&\, (r < y)))$.

Легко показать, что условия $U_{\alpha_1}$, $U_{\alpha_2}$ и $U_{\alpha_3}$ являются истинными и это доказывает свойство частичной корректности приведенной программы относительно ее спецификации.

Остановимся теперь на обсуждении вопросов о формировании инварианта цикла. Инвариант цикла inv – это утверждение, удовлетворяющее следующим условиям:

1) inv истинен перед входом в цикл;
2) inv остается истинным при каждом выполнении тела цикла;
3) при выходе из цикла inv обеспечивает истинность постусловия оператора цикла.

При определенных ограничениях на язык программирования и язык спецификаций для каждой частично корректной программы существует инвариант цикла. Однако задача построения инварианта цикла для заданной программы может оказаться сколь угодно сложной. Поэтому невозможны практичные универсальные методы синтеза инвариантов цикла.

Если при доказательстве обнаружено ложное условие, то причиной этого может быть как ошибка в программе, так и ошибка в построении инварианта цикла, причем формально разделить эти два случая невозможно. Кроме того, не любой инвариант цикла может обеспечить проведение доказательства частичной корректности программы.

При поиске подходящих инвариантов циклов можно руководствоваться некоторыми эвристиками. При использовании простых эвристик инвариант цикла inv представляют в виде конъюнкции: $inv=inv_1$ & $inv_2$ & … & $inv_k$. С помощью эвристических методов строятся некоторые (не обязательно все) конъюнктивные члены inv, называемые иногда частями инварианта цикла. Среди эвристических методов следует отметить следующие методы:

1) метод слабой интерпретации;
2) метод, использующий условие выхода из цикла;
3) стратегия прямого прослеживания.

Метод слабой интерпретации применяется при нахождении частей инвариантов циклов, имеющих вид линейных равенств или неравенств и зависящих чаще всего от двух переменных. Информация для синтеза инвариантов циклов извлекается обычно из входного условия и логических выражений программы. В начале с помощью эвристик строится условие $\beta$, истинное перед входом в цикл.

Примеры типичных эвристик:

1) если переменная $i$ принимает значения $n$ и $n$-1, то предполагается неравенство $i<=n$;
2) если переменная $i$ принимает значения $n$ и $m$ и справедливо $m<=n$, то предполагается соотношение $m<=i<=n$.

Затем $\beta$ модифицируется с учетом изменений данных переменных с помощью операторов присваивания в теле цикла, а также с учетом условных операторов.

Название этого метода связано с символическим выполнением операторов программы на упрощенной модели.

Метод, использующий условие $\neg\ \alpha$ на выходе из цикла while ($\alpha$) do op, состоит в следующем. Так как $\neg\ \alpha \rightarrow Q$, где $Q$ – постусловие оператора цикла, остается истинным при каждом выполнении тела цикла, а также гарантирует истинность постусловия при выходе из цикла, то $\neg\ \alpha \rightarrow Q$ можно рассматривать как кандидата в инвариант цикла. Для получения инварианта цикла $\neg\ \alpha \rightarrow Q$ усиливается с помощью следующих эвристик:

1) уменьшение области действия квантора существования в $Q$;
2) увеличение области действия квантора общности в $Q$;
3) если из $\neg\ \alpha$ следует $x=e$, то выполнение замены в $Q$ переменной $x$ выражением $e$.

Стратегия прямого прослеживания применяется при получении некоторых частей инварианта цикла из входного условия, а также из ранее построенных частей инварианта. В частности, извлекаются те утверждения, которые не зависят от переменных, изменяемых в теле цикла. Как и в методе слабой интерпретации, стратегия прямого прослеживания включает модификацию утверждений с учетом условных операторов и операторов присваивания в теле цикла.

Задание на выполнение лабораторной работы № 3 определяется двумя атрибутами $A_0$ и $A_1$. Атрибут $A_0$ указывает номер варианта исследуемой программы.

Варианты программ приведены при описании лабораторной работы № 1. Атрибут $A_1$ указывает вариант допустимых средств, которыми разрешено воспользоваться при формировании инвариантов циклов $\{A_1 = 0$: разрешено использование только простых эвристик; $A_1$=1: допустимо использование всех известных средств$\}$.

Если доказательство частичной корректности программы завершилось успешно, рекомендуется провести мутации программы в следующих вариантах:

1) мутация должна явно привести к нарушению работы программы;
2) мутация не должна нарушить работу программы. Провести формально характеризацию их частичной корректности.

Контрольные вопросы

1. Что понимают под логической спецификацией программы?
2. Как обеспечивается иерархическое описание логической спецификации программы?
3. Как определяется частичная корректность программы?
4. Как определяется полная корректность программы?
5. Каковы основные положения метода индуктивных утверждений?
6. Что понимают под инвариантом цикла?
7. Какие принципиальные трудности стоят на пути автоматизации проверки частичной корректности программы?
8. Какие методы построения инварианта цикла Вы знаете?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Непонящий В.А. Прикладные методы верификации программ /В.А. Непомнящий, О.М. Рякин. – М.: Радио и связь, 1988. – 256 с.
2. Агафонов В.Н. Спецификация программ: понятийные средства и их организация /В.Н. Агафонов. – Новосибирск: Наука СО, 1990. – 224 с.
3. Алагич С. Проектирование корректных структурированных программ /С. Алагич, М. Арбиб. – М.: Радио и связь, 1984. – 264 с.
4. Бейбер Р.Л. Программное обеспечение без ошибок /Р.Л. Бейбер. – М.: Джон Уайли энд Санз, Радио и связь, 1996. – 176 с.

# ЛАБОРАТОРНАЯ РАБОТА № 4

# АНАЛИЗ ЗАВЕРШАЕМОСТИ ПОСЛЕДОВАТЕЛЬНОЙ ПРОГРАММЫ

Цель работы: освоение простейших методов анализа завершаемости последовательной программы.

**Основные теоретические положения**

При анализе корректности программы обычно преследуется цель доказательства ее полной корректности. С методологической точки зрения оказывается целесообразнее проводить отдельно доказательство частичной корректности и завершаемости программы. Свойства завершения вычислений в программе (т.е. достижения в процессе выполнения программы оператора Stop) является одним из важных свойств корректности программы. Особенность свойства завершения состоит в том, что оно не зависит от постусловия программы и для фиксированной программы полностью определяется предусловием. В общем случае формальный анализ свойства завершения представляет весьма сложную проблему.

Имеются по меньшей мере две причины, по которым программа с заданным предусловием *P* может не иметь нормального завершения:

1) обращение к частично определенной функции со значением аргумента вне ее области определения;
2) выполнение бесконечной последовательности операторов в цикле (зацикливание).

Первая причина может приводить к аварийному завершению даже не циклических программ (например, при выполнении деления на 0). Вторая причина присуща программам с циклами. Зацикливание возможно даже для программ, не использующих частично определенные функции.

Ограничимся рассмотрением методов анализа завершения циклических вычислений без учета других возможных причин не завершения (т.е. предлагается, что любой нециклический оператор завершается).

Основной принцип доказательства завершаемости программы легко понять при решении следующих простых представительных задач.

Задача 1. Имеется квадратная таблица, заполненная числами. Разрешается выполнять следующие операции:

1) менять знаки всех чисел в одной строке;
2) менять знаки всех чисел в одном столбце.

Необходимо доказать, что можно добиться того, чтобы сумма чисел в каждой из строк и в каждом из столбцов была бы неотрицательной.

В соответствии с законами арифметики, если сумма чисел в строке (столбце) отрицательна, то после перемены знаков сумма чисел в этой строке (столбце), станет положительной. Таким образом, меняя знаки в строках и столбцах с отрицательной суммой, можно добиться желаемого. Можно предложить следующий алгоритм: пока есть хоть одна строка или столбец с отрицательной суммой, выполнять изменение знаков чисел в одной из таких строк или столбцов.

Ясно, что после завершения работы этого алгоритма во всех строках и столбцах суммы чисел будут неотрицательными (т.е. алгоритм является частично

корректным). Однако не вполне ясно, что это завершение произойдет. Можно было бы, вообще говоря, опасаться такой ситуации: после изменения знаков в строке становится отрицательной сумма чисел в каком-то столбце, после изменения знаков в столбце становится отрицательной сумма чисел в какой-то строке и т.д. Таким образом, нам необходимо доказать, что наш алгоритм не может работать неограниченно долго.

Ключевой идеей здесь является следующая: при замене знаков в строке (столбце) с отрицательной сумой сумма всех чисел таблицы возрастает (поскольку сумма чисел в строке (столбце) из отрицательной делается положительной, а остальные числа не меняются). С другой стороны, сумма чисел таблицы не может быть больше суммы модулей чисел исходной таблицы. Таким образом, увеличение суммы чисел таблицы не может продолжаться неограниченно долго.

Итак, доказательство завершения работы алгоритма может быть сведено к поиску подходящей ограниченной величины, которая может только возрастать (или только убывать).

Задача 2. Пусть на полке в беспорядке стоят том*а* собрания сочинений. Владелец книг, заметив два т*о*ма, стоящие в неправильном порядке (том с меньшим номером находится правее т*о*ма с б*о*льшим номером), меняет их местами. Необходимо показать, что рано или поздно том*а* собрания сочинений будут расположены по порядку.

Использованный алгоритм может быть сформулирован следующим образом: пока есть два т*о*ма, стоящие в неправильном порядке, выполнять их перестановку. Требуется доказать, что алгоритм заканчивает работу.

В соответствии с общим принципом введем величину, которая возрастает при каждой перестановке. Следуя Э. Дейкстре, воспользуемся физической аналогией. Представим, что все том*а* имеют одинаковую толщину, но разный «вес»: чем больше номер т*о*ма, тем он «тяжелее». В этом случае можно описать наши действия следующим образом: обнаружив более «тяжелый» том слева от более «легкого», мы меняем их местами (не сдвигая с места остальные том*а*). Легко видеть, что при этом «центр тяжести» всего собрания сочинений смещается вправо. Может ли такое происходить бесконечное число раз? Нет, так как число возможных положений «центра тяжести» конечно (оно не больше числа возможных расстановок том*о*в) и «центр тяжести» не может все время сдвигаться вправо.

Если мы теперь желаем перейти от физической аналогии к математическому доказательству, необходимо воспользоваться формулой для координаты «центра тяжести»: $a(c) = \sum_{i}^{n} ia(i) / M$ (здесь «вес» *i*-го т*о*ма равен *i*, а положение *i*-го т*о*ма не полке есть $a(i)$; $M$ - общая масса). Поскольку $M$ является константой, можно рассматривать только $\sum_{i}^{n} ia(i)$.

Будем полагать, что $i>j$ и $a(i)<a(j)$. Перестановка том*о*в приведет к замене в $\sum_{i}^{n} ia(i)$ слагаемых $ia(i)+ja(j)$ на $ja(i)+ia(j)$. Покажем, что $ja(i)+ia(j)>ia(i)+ja(j)$. Представим это соотношение в виде $(i-j)a(j)>(i-j)a(i)$. Так как $i>j$, то $i-j>0$. Кроме того, из $a(i)<a(j)$ следует $a(j)>a(i)$ и

справедливость анализируемого соотношения. Заметим, что $\sum_i^n ia(i) < \sum_i^n n \cdot n = n^3$. Следовательно, сумма не может возрастать неограниченно.

Обычно стараются найти монотонно изменяющуюся величину с натуральными значениями. Не всегда это возможно и удобно. Бывают задачи, когда значения монотонно изменяющейся величины берутся из другого множества.

Задача 3. (Э. Дейкстра). На сортировочной станции имеется несколько поездов. Разрешаются два типа действий:

1) расцепление поезда, состоящего из нескольких вагонов, на два поезда;
2) удаление поезда, если в нем всего один вагон.

Требуется доказать, что, выполняя эти действия в произвольном порядке, мы рано или поздно удалим все вагоны.

Предлагаемый алгоритм работы с поездами описывается следующим образом: пока есть хоть один поезд, выполнить (выбрать какой-то поезд; если в нем один вагон, то удалить его, иначе расцепить его).

В этой задаче ни число поездов, ни число вагонов не годятся на роль монотонно убывающей величины (число поездов может расти при расцеплении поезда; число вагонов при расцеплении остается прежним).

Для решения этой задачи предлагается ввести следующее понятие. Говорят, что пара $(a, b)$ натуральных чисел меньше пары $(c, d)$, если $a<c$ или $a=c$ и $b<d$. Рассмотрим пару (число вагонов, число сцепок), где число сцепок есть число межвагонных соединений, равное числу вагонов минус число поездов. Эта пара убывает, так как при удалении поезда уменьшается число вагонов, а при расцеплении поезда число вагонов не меняется, а число сцепок убывает.

Можно доказать следующую лемму: не существует бесконечной убывающей последовательности пар натуральных чисел $(a_1, b_1)> (a_2, b_2)>\ldots> (a_n, b_n)>\ldots$ .

Доказательство. Последовательность $a_1,a_2,\ldots$ – невозрастающая. Поскольку члены этой последовательности – натуральные числа, то, начиная с некоторого места, все они равны (они не могут бесконечное число раз убывать). Начиная с этого места, вторые члены пар (согласно нашему определению)образуют строго убывающую последовательность натуральных чисел, а бесконечных строго убывающих последовательностей натуральных чисел не бывает. Таким образом, задача 3 решена.

Рассмотрим еще одну задачу.

Задача 4. Имеется конечная последовательность нулей и единиц, содержащая ровно две единицы. Разрешается заменять группу символов 01 на группу 10…0 (число нулей любое). Требуется доказать, что такие операции не могут выполняться бесконечно много раз.

Решение этой задачи может быть получено следующим образом. Операции не изменяют числа единиц, поэтому в любой момент в последовательности будет ровно две единицы. Рассмотрим пару (число нулей перед первой единицей, число нулей перед второй единицей). При каждой замене группа 01 на 10…0 эта пара уменьшается. Если заменяется группа, содержащая первую единицу, то уменьшается первый член пары (второй при этом может увеличиваться!); если заменяется группа со второй единицей, то первый член пары не меняется, а второй

уменьшается. Согласно доказанной лемме этот процесс не может продолжаться бесконечно.

Перейдем теперь к определению понятия фундированного множества. Пусть *S*– частично упорядоченное множество относительно бинарного отношения > на его элементах, т.е. на множестве *S* для $a,b,c \in S$ выполняются свойства:

1) антирефлексивность: $\neg(a > a)$;
2) антисимметричность: $a > b \to \neg(b > a)$
3) транзитивность: $a > b \wedge b > c \to a > c$.

Множество *S* называется фундированным, если не существует бесконечной убывающей последовательности элементов из $S: \alpha_0 > \alpha_1 > \alpha_2 > \ldots$. Примерами фундированных множеств являются: множество Nat={0, 1, 2, …} с отношением >(строго больше); множество $\Sigma^*$ всех слов в алфав*и*те Σ с отношением $W_1 > W_2$, означающим, что $W_2$ есть собственное подслово $W_1$. Множество Int={…-2, -1,0, 1, 2,…} не является фундированным.

Рассмотрим теперь два наиболее известных метода анализа завершения циклических вычислений:

1) метод Флойда;
2) метод счетчиков.

Метод Флойда тесно связан с методом индуктивных утверждений, также предложенным Флойдом.

Пусть Prgm – аннотированная программа, для которой доказана частичная корректность в соответствии с методом индуктивных утверждений. С каждой контрольной точкой *r*, лежащей на циклическом пути, свяжем частичную функцию $\psi_r(x_1,\ldots,x_n)$, принимающую значения в фундированном множестве *S*.

Эту функцию = называют ограничивающей.

Условие частичной корректности для пути <rαt> можно записать в виде $U_\alpha$: $\text{inv}_r \to \text{inv}_t[\alpha^{-1}>$, где $\alpha^{-1}$ есть инверсия пути $\alpha$ (обратное прохождение от *t* к *r*).

В пути $\alpha(\alpha^{-1})$ будем различать *f*- элементы и *p*-элементы, описывающие подстановки и предикаты соответственно. Последовательность элементов из $\alpha^{-1}$, следующую за элементом $P_i$ будем называть суффиксом для $P_i$.

Выделение из последовательности $\alpha^{-1}$ только *f*-элементов будем называть *f*-проекцией этой последовательности.

Обозначим через $S_i$ – *f*-проекцию суффикса для $P_i$. Тогда $\text{inv}_t[\alpha^{-1}>$ можно представить в виде

$$\bigwedge_{P_i \text{ из } \alpha^{-1}} P_i[S_i > \to \text{inv}_t[S_0 >$$

Обозначим $\bigwedge_{P_i \text{ из } \alpha^{-1}} P_i[S_i >$ через $P_\alpha$. Тогда условие завершения $W_\alpha$, связанное с путем <*r* α *t*>, можно записать следующим образом (*t* и *r* могут совпадать):

$$< r\,\alpha\,t >: W_\alpha \wedge \text{inv}_r \to (P_\alpha \to \psi_r \in S \wedge \psi_t[S_0 > \in S \wedge \psi_r > \psi_t[S_0 >).$$

Здесь $\psi_r$, $\psi_t$ - ограничивающие функции, приписанные точками *r* и *t* соответственно. Эти точки должны находиться на циклическом пути.

По набору $W_\alpha$ условий завершения для всех путей $\alpha$ между соседними контрольными точками, лежащими на циклическом пути, можно определить свойство завершения в соответствии со следующей теоремой.

Теорема (о завершении). Программа Prgm, частично корректная относительно $P$ и $Q$, завершается для всех ее выполнений, удовлетворяющих $P$, если все ее условия завершения $W_\alpha$ истинны.

Заметим, что теорема по существу устанавливает полную корректность программы, так как предполагает ее частичную корректность. При доказательстве завершения может потребоваться усилие инвариантов, используемых при доказательстве частичной корректности. Успех доказательства в значительной мере предопределяется искусством выбора ограничивающих функций $\psi$.

Сущность метода счетчиков состоит в преобразовании программы Prgm в программу Prgm′ путем введения новых переменных-счетчиков, добавляемых по одной в каждый цикл программы. Переменная-счетчик должна инициализироваться перед входом в цикл и увеличивать свое значение при каждом прохождении по циклу. В программе Prgm′ измененные инварианты циклов представляются в виде, содержащем условие ограниченности значений счетчиков функциями, зависящими только от входных переменных. Это указывает на существование верхней грани значений счетчиков. Для программы Prgm′ доказывается частичная корректность, которая одновременно характеризует и завершаемость программы, т.е. ограниченность значений счетчиков.

В отличие от метода Флойда в методе счетчиков не требуется вводить вспомогательные ограничивающие функции в контрольных точках. Переменные-счетчики рассматриваются как органическая часть программы, позволяющая логически выразить свойство завершения в индуктивных утверждениях (инвариантах циклов). Преимуществом такого подхода перед методом Флойда является то, что одни и те же инварианты используются при доказательстве частичной корректности и завершения. Однако метод Флойда обладает б*о*льшей общностью.

Успех доказательства завершения в методе счетчиков определяется тем, удастся ли подобрать соответствующее (доказуемое) условие ограниченности счетчика в инварианте цикла. Подбор ограничивающих функций (в методе Флойда) и подходящих инвариантов циклов в программе, дополненной счетчиками, требует творческого подхода. В библиографии [1-4] можно найти дополнительную важную информацию по рассмотренной проблеме.

Задание на выполнение лабораторной работы № 4 определяется двумя атрибутами $A_0$ и $A_1$. Атрибут $A_0$ указывает номер варианта исследуемой программы. Варианты программ приведены при описании лабораторной работы № 1. Атрибут $A_1$ указывает вариант метода доказательства завершения программы {$A_1$=0: метод Флойда; $A_1$=1: метод счетчиков}.

Контрольные вопросы

1. Что понимают под завершением программы?
2. Какие причины приводят к незавершаемости программы?
3. Какие основные положения метода Флойда?
4. Каковы основные положения метода счетчиков?
5. Какие принципиальные трудности связаны с использованием метода Флойда?

6. Какие принципиальные трудности связаны с использованием метода счетчиков?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Непомнящий В.А. Прикладные методы верификации программ /В.А. Непомнящий, О.М. Рякин. – М.: Радио и связь, 1988. – 256 с.
2. Алагич С. Проектирование корректных структурированных программ /С. Алагич, М.Арбиб. – М.: Радио и связь, 1984. – 264 с.
3. Грис Д. Наука программирования /Д. Грис. – М.: Мир, 1984. – 416 с.
4. Простое и сложное в программировании /Авт. пред. Е.П. Велихова. – М.: Наука, 1988. – 176 с.

# ЛАБОРАТОРНАЯ РАБОТА № 5

# АНАЛИЗ КОРРЕКТНОСТИ ПАРАЛЛЕЛЬНЫХ СИСТЕМ С ИСПОЛЬЗОВАНИЕМ СЕТЕЙ ПЕТРИ

Цель работы: освоение простейших методов анализа параллельных систем, моделируемых сетями Петри.

**Основные теоретические положения**

При выполнении детерминированной последовательной программы для фиксированных значений входных переменных возникает один маршрут вычислений (конечный или бесконечный). Когда выполняется недетерминированная программа, возникает множество маршрутов вычислений, в котором каждый маршрут может развиваться независимо от других.

Для недетерминированной программы $A$ относительно входного условия $P$ и выходного условия $Q$ частичная корректность определяется следующим образом: если программа $A$ начинает выполняться, когда $P$ истинно, то для каждого маршрута выполнения, который достигает выхода программы, на выходе истинно $Q$.

Если маршруты вычислений недетерминированной программы независимы, удается обобщить правила вывода условий корректности детерминированных программ без привлечения новых понятий. Иное положение для параллельных программ, у которых маршруты выполнения (процессы) могут влиять друг на друга в случае работы с одинаковыми переменными. При таком влиянии процессов друг на друга результат параллельной работы программы может зависеть от относительной скорости процессов. Для параллельных программ возможно новое явление, называемое блокировкой процессов. Верификация параллельных программ значительно сложнее верификации последовательных программ.

При анализе корректности параллельных систем самых различных видов широкое применение получило средство моделирования, известное как сеть Петри [1-4]. Основу сетей Петри составляют понятия условие и событие.

Появление событий зависит от состояния системы, которая может выражаться набором условий. Эти условия называются предпосылками события. В результате появления события удовлетворяются некоторые другие условия, которые называют следствиями. В сетях Петри события представляются переходами и графически отображаются «полочками». Предпосылки события называются его входными местами и графически отображаются кружками, связанными с переходом дугами.

Следствия событий называются его выходными местами и отображаются кружками, к каждому из которых ведут дуги от перехода. Краткое изложение основ теории сетей Петри можно найти в методическом пособии [5].

Проиллюстрируем на небольшом примере, как моделируются и анализируются системы с использованием сетей Петри [4]. Пусть в системе действуют два независимых процесса, один из которых $PW$ производит сообщения («писатель»), а другой – $PR$ потребляет сообщения («читатель») и, кроме этого, используется третий независимый процесс $PB$ для координации обработки сообщений («буфер», «посредник»). Условимся путем приписывания префиксов C и $E$ различать условия и события.

На рисунке 3 изображена базовая структура этой системы. В системе выделены следующие основные понятия:

1) попытка записи (write-attempt);
2) подтверждение произведенной записи (ack-written);
3) подтверждение пропуска записи (ack-w-skipped);
4) попытка чтения (read-attempt);
5) подтверждение чтения (ack-read);
6) подтверждение пропуска чтения (ack-r-skipped);

Основные правила поведения процесса *PB* можно описать следующим образом:

1: C. read-attempt & C. full→E.read;
2: E.read→C. empty & C. ack-read;
3: C. write-attempt & C. empty→E.write;
4: E. write→C. full & C. ack-written;
Полагается, что C. empty = ¬ C. full .

Пример системы

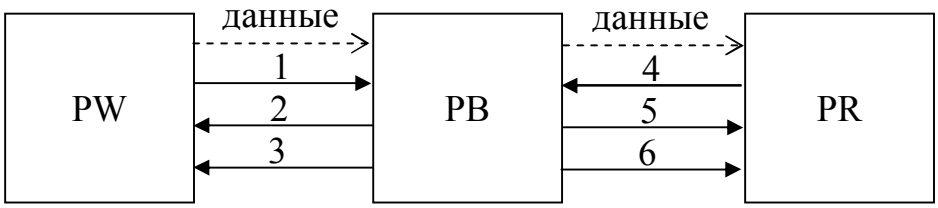


Рисунок 3. – Базовая структура системы: 1 – write_attempt;
2 – ack_written; 3 – ack_w_skipped;
4- read_attempt; 5 – ack_read; 6 - ack_r_skipped

Правила поведения при неудачных попытках записи и чтения можно описать следующим образом:

5: C. read-attempt & C. empty→E. skip-read;
6: E. skip-read →C. ack-r-skipped & C. empty;
7: C. write-attempt & C. full → E. skip-write;
8: E. skip-write →C. fck-w-skipped & C. full.

Есть два входных события, которые приводят к условиям read-attempt и write-attempt:

9: E. I 1→ C. read-attempt;
10: E. I 2→ C. write-attempt.

В рассмотренной модели восемь условий и шесть событий, а сама модель представляется сетью Петри, изображенной на рисунке 4.

Для улучшения свойств модели выделенные пунктиром переходы I 1 и I 2 и инцидентные им дуги удалим из сети Петри. Теперь каждый из оставшихся переходов имеет одинаковое число входящих и выходящих дуг, т.е. сеть Петри обладает свойством сохранения числа меток (ни одна метка не попадает в сеть извне и не покидает ее).

Теперь сеть Петри можно проанализировать, построив, например, граф достижимости обобщенных маркирований.

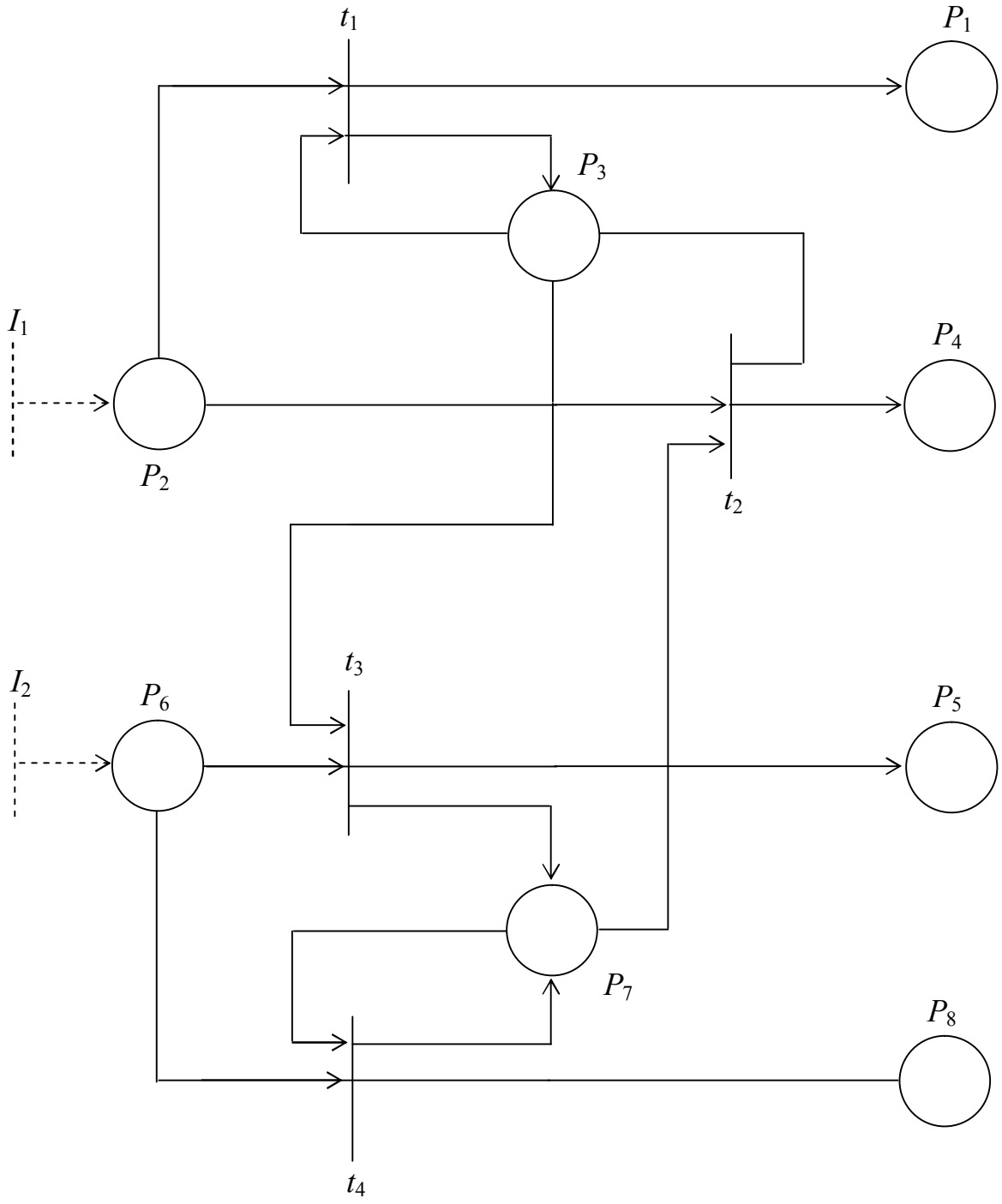


Рисунок 4 – Сеть Петри

Задание на выполнение лабораторной работы № 5 определяется атрибутами $A_0$, $A_1$, $A_2$ и $A_3$.

Атрибут $A_0$ указывает на номер варианта анализируемых поведенческих свойств сети Петри {$A_0$=0: требуется провести характеризацию ограниченности и безопасности сети Петри; $A_0$=1: требуется провести характеризацию активности (живости); $A_0$=2: требуется обеспечить проверку достижимости указанного целевого состояния; $A_0$=3: требуется найти *t*-инварианты сети Петри; $A_0$=4: требуется найти *p*-

инварианты сети Петри; $A_0$=5: требуется найти сифоны сети Петри; $A_0$=6: требуется найти ловушки сети Петри}.

Атрибут $A_1$ указывает номер варианта анализируемых структурных свойств сети Петри {$A_1$=0: требуется провести характеризацию свойств: автоматная сеть Петри, маркированный граф; $A_1$=1: требуется обеспечить характеризацию свойства: сеть свободного выбора; $A_1$=2: требуется обеспечить характеризацию свойства: расширенная сеть свободного выбора; $A_1$=3: требуется обеспечить характеризацию свойства: сеть с асимметричным выбором}.

Атрибут $A_2$ указывает номер варианта метода, который объявляется приоритетным {$A_2$=0: приоритетным объявляется метод построения графа достижимости обобщенных маркирований; $A_2$=1: приоритетным объявляется метод решения уравнений}.

Атрибут $A_3$ указывает номер варианта исследуемой сети Петри. Сеть Петри задается путем описания ее структуры и начального маркирования. Структура сети задается множеством $\{\Gamma^{-1}t : t : \Gamma t \mid t \in T\}$, где $T$ – множество переходов, пронумерованных натуральными числами, начиная с 1; $\Gamma^{-1}t \subseteq P$ и $\Gamma t \subseteq P$ - обратное и прямое отображения относительно перехода $t$; $P$ – множество позиций (мест) пронумерованных натуральными числами, начиная с 1. Начальное маркирование задается множеством $\{p = m \mid p \in P, m > 0\}$, где $m$ указывает число маркеров.

{$A_3$=0: сеть Петри, описывающая протокол передачи сообщений [6] (интерпретация игнорируется).

Структура=
{4,10:1:1,14; 1,5:2:2,4; 2,6:3:3,5; 3,12:4:6,17;
1:5:4; 2:6:5; 2:7:6; 4,14:8:1,14; 7:9:4;
8:10:5; 9:11:6; 3,13:12:6,13; 4,11:13:7,15;
5,7:14:4,8; 6,8:15:5,9; 9,13:16:6,16;
4,15:17:7,15; 9,12:18:6,12; 15,24:19:15,21;
12,23:20:12,20; 15,18:21:10,21; 19,21:22:18,22;
20,22:23:19,23; 16,23:24:12,20; 14,18:25:14,21;
18:26:21; 19:27:22; 20:28:23; 13,23:29:13,26;
24:30:21; 25:31:22; 26:32:23; 14,24:33:11,21
21,25:3422,24; 22,26:35:23,25; 17,23:36:13,26}

Начальное маркирование = $\{P_4 = 1, P_5 = 1, P_{10} = 1, P_{12} = 1, P_{21} = 1, P_{22} = 1, P_{23} = 1\}$

$A_3$=1: сеть Петри, описывающая функционирование производственного модуля [7] (интерпретация игнорируется )

Структура=
{4,18:1:1,3,6,7,13; 1,2:2:9; 6,7:3:4,8;
3,5:4:2; 9:5:5,10; 8,10,19:6:11; 11:7:12;
12,15:8:16,17; 13:9:14; 14,16:10:15;
17:11:18,19}

Начальное маркирование= $\{P_4 = 1, P_5 = 1, P_8 = 1, P_{10} = 1, P_{15} = 1, P_{19} = 1\}$.

$A_3$=2: сеть Петри, описывающая функционирование диска в режиме разделения двух компьютеров (интерпретация игнорируется).

Структура=

{1,18:1:2; 2,18:2:3; 3,17:3:4;
4,18:4:5; 5,18:5:6; 6:6:7,17,18;
7:7:8,18; 8:8:9,18; 9:9:1,18;
10,18:10:11; 11,17:11:12; 12,18:12:13;
13,18:13:14; 14:14:15,17,18;
15:15:16,18; 16:16:10,18}
Начальное маркирование= $\{P_1 = 1,\ \ P_{10} = 1,\ P_{17} = 1,\ P_{18} = 5\}\}$.

Контрольные вопросы

1. Как формально описывается сеть Петри?
2. Что понимается под ограниченностью сети Петри?
3. Что понимается под активностью (живостью) сети Петри?
4. Как формулируется задача о достижимости предписанного маркирования для сети Петри?
5. Какие важные подклассы сетей Петри вы знаете?
6. Что понимается под *t*-инвариантами и *p*-инвариантами сети Петри?
7. Что понимается под сифонами и ловушками сети Петри?
8. Каковы основные положения алгоритма построения графа достижимых обобщенных маркирований?
9. Каковы основные положения алгоритма Тудика?
10. Каковы основные положения алгоритмов поиска сифонов и ловушек?
11. Каковы принципиальные трудности, стоящие на пути решения проблемы достижимости для произвольных сетей Петри?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Питерсон Дж. Теория сетей Петри и моделирование систем /Дж. Питерсон. – М.: Мир, 1984. – 264 с.
2. Котов В.Е. Сети Петри /В.Е. Котов. – М.: Наука. ГРФМЛ, 1984. – 160 с.
3. Мурата Т. Сети Петри: свойства, анализ, приложения /Т. Мурата //ТИИЭР, т.77. – № 4. – 1989. – С. 41-85.
4. Проверка и утверждение программ реального времени /Под ред. А.И. Никитина. – Киев: Наукова думка, 1990. – 216 с.
5. Методические указания к изучению дисциплины «Технология проектирования программных систем» для студентов дневной и заочной форм обучения по специальности 7.091501 «Компьютерные системы и сети». Сети Петри. Ч. 1 Элементы теории. – Севастополь: Изд-во СевГТУ, 1996. – 72 с.
6. Системы параллельной обработки /Под ред. Д. Ивенса. – М.: Мир, 1985. – 216 с.
7. Технология системного моделирования /Е.Ф. Аврамчук [и др.] – М.: Мир, 1985. – 416 с.

# ЗАКЛЮЧЕНИЕ

Данные методические указания поддерживают выполнение цикла лабораторных работа по дисциплине «Технология проектирования программных систем». Верификация программ предназначена для решения одной из важных задач разработки надежного программного обеспечения. Верификация программы не исключает тестирования, а дополняет его. Цикл лабораторных работ, описанных в данных методических указаниях, преследует цель закрепления общетеоретических знаний, связанных с разработкой надежных программных систем.

Теория формализованной (технологической) разработки (проектирования) надежных программных систем еще находится в стадии становления и наталкивается на принципиальные трудности, присущие самой проблеме. Перспективные направления решения этой проблемы находятся в русле совершенствования технологии программирования, поддерживающей конструктивный (синтетический) подход при разработке программного обеспечения.

Заказ № _____ от «____»_____ 2009 Тираж 50 экз.
Изд-во СевНТУ