ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Екатерина Лаврищева, Владимир Петрухин
Лекция 7. Формальные спецификации, доказательство и верификация программ

Современные направления в области проверки правильности программ – формальные спецификации и методы доказательства их правильности. Для доказательства того, что *спецификация программы* задает правильное решение некоторой задачи, для которой она разработана, привлекается математический аппарат.

В формальных методах нет рутинного написания спецификации на ЯП, а есть *анализ* текста и описание поведения программы в стиле, близком математической нотации, путем рассуждений и доказательств, принятых в математике. Формальные методы в программировании появились одновременно с самим программированием, на которое повлияли работы *по* теории алгоритмов А.А. Маркова [6.1], А.А. Ляпунова [6.2], схемы Ю.И.Янова [6.3], формальные нотации языка описания взаимодействующих процессов К.А. Хоара [6.4] и др.

В 70-х годах прошлого столетия появились формальные спецификации, которые близки ЯП и предоставляют средства, облегчающие проводить рассуждение о свойствах формальных тестов и сближающие их с математической нотацией. Несмотря на это, исследования формальных методов носили в основном академический, теоретический характер, поскольку извлечь из них практическую пользу в программировании не удавалось в силу огромных затрат на формальную спецификацию программ и разработку дополнительных [6.5-6.10] аксиом, утверждений и условий, называемых предварительными условиями (предусловиями) и постусловиями, определяющими заключительные правила получения правильного результата.

Под спецификацией понимается формальное описание функций и данных программы, с которыми эти функции оперируют. Различают видимые данные, т.е. входные и *выходные параметры*, а также скрытые данные, которые не привязаны к реализации и определяют *интерфейс* с другими функциями.

Предусловия – это ограничения на совокупность входных параметров и постусловия – ограничения на *выходные параметры*. *Предусловие* и *постусловие* задаются предикатами, т.е. функциями, результатом которых будет булевская величина ($true$ / $false$). *Предусловие* истинно тогда, когда входные параметры входят в область допустимых значений данной функции. *Постусловие* истинно тогда, когда совокупность значений удовлетворяет требованиям, задающим формальное *определение* критерия правильности получения результата.

Доказательство проводится с помощью *утверждений*, которые составляются в формальном языке и служат способом проверки правильности программы в заданных точках. Набор утверждений использует предусловия и последовательность операций, приводящих к проверке результата относительно отмеченной точки программы, для которой сформулировано заключительное утверждение. Если утверждение соответствует конечному оператору программы, где требуется получить окончательный результат, то с помощью заключительного утверждения и постусловия делается окончательный *вывод* о частичной или полной правильности работы программы.

6.1. Анализ языков формальной спецификации программ

Языки спецификаций, используемые для формального описания *свойств программ*, более высокого уровня, чем ЯП. Их можно классифицировать *по* таким категориям: универсальные языки с общематематической основой (например, RAISE, Z, *API*, *VDM* и др.) [6.6-6.10]; языки спецификации проблемных областей (например, ЯП, языки спецификаций ПрО или доменов – *DSL* и др.) [6.11-6.14]; специализированные языки спецификации (например, языки таблиц, логики, равенств и подстановок и др.) [6.5]; языки, ориентированные на спецификацию параллельных процессов (например, CIP-L, Ada-68 *Concurrent Pascal* и др.) [6.11] (рис. 6.1).

Спецификация программы – это точное, однозначное и недвусмысленное описание программы с помощью математических понятий, терминов, правил синтаксиса и семантики *языка спецификации*. В *языке спецификаций* могут быть понятия и конструкции, которые нельзя выполнить на компьютере, они представляются последовательностью операций, функций, понятных для интерпретации.

Описание задачи в языке спецификации включает в себя описание общего контекста всех понятий, через которые определяются понятия, участвующие в формулировке задачи или в описании модели ПрО (домена).

Описание задачи дается в виде аксиом, утверждений, пред- и постусловий, требующих для их реализации не систем программирования, а специального аппарата для доказательства или верификации описания задач, в частности интерпретаторов или метасистем.

увеличить изображение

Рис. 6.1. Категории языков спецификации

Универсальные языки спецификации (*VDM*, Z, RAISE и др.) имеют общематематическую основу и следующие виды средств:

- логики первого порядка, включая кванторы;
- арифметические операции;

- средства образования множеств с помощью логических формул и операций над множествами;
- средства описания конечных последовательностей (кортежей, списков) и операции над ними;
- средства описания конечных функций и операции над ними;
- средства описания древовидных структур;
- средства построения областей или множества объектов, включая произведения, объединения и рекурсивные определения;
- определение функций с помощью выражений и равенств, включая рекурсивные определения;
- процедурные средства ЯП (операторы присваивания, цикла, выбора, выхода);
- *операции композиции*, аргументами и результатами которых могут быть функции, выражения, операторы.

В *VDM* и RAISE нет средств описания графовых структур, управления и параллелизма, однако имеется механизм конструирования новых структур данных.

Языки спецификации областей включают в себя следующие языки:

- спецификации доменов;
- описания взаимодействий;
- спецификации ЯП и трансляторов;
- спецификации БД и знаний;
- *спецификации пакетов* прикладных программ и др.

Каждый из этих языков имеет специализированные средства, отображающие специфические особенности соответствующей области.

Язык спецификации доменов *DSL* (*Domain* Specific *Language*) представляет некоторое *подмножество* языка программирования и специально средства для описания специальных проблем домена [6.14]. Он подразделяется на внешние и внутренние языки. *Внешние языки* (типа Unix, *XML* и др.) *по* уровню выше языка описания приложения. Описание в нем сводится к языку *DSL* специальными генераторами или текстовыми редакторами, трансформирующими абстрактные понятия домена к понятиям языка *DSL*. *Внутренние языки* (C, C++), а также языки *Java*, Smalltalk ограничены синтаксисом и семантикой основного базового языка программирования приложений.

Языки описания взаимодействий и параллельного выполнения в отличие от ЯП позволяют специфицировать процессы управления вычислениями, передачей сообщений и взаимодействием объектов в распределенных системах.

Метаязыки позволяют специфицировать контекстные зависимости синтаксиса ЯП, лексический и синтаксический *анализ* трансляторов с помощью регулярных выражений КС-*грамматик в форме Бэкуса-Наура*. Для спецификации семантики языков используется *формализм* равенств. Техника описания ЯП основывается на атрибутных грамматиках и абстрактных типах данных. Задача описания ЯП для перевода решаются путем использования денотационных, алгебраических и атрибутных подходов, а также логических терминов, ориентированных на верификацию [6.11-6.16].

Языки описания средств программирования включают в себя языки, основанные на равенствах и подстановках с операционной семантикой (*Лисп*, Рефал); логические языки; языки операций (APL) над последовательностями и матрицами; табличные языки; сети, графы [6.5, 6.11]. Язык логики предикатов с набором базисных функций используется для записи пред- и постусловий, инвариантов.

Отдельные *операции* логики предикатов используются также в языках логического программирования (например, *Пролог*).

Основой описания математических объектов являются равенства и подстановки. Для определения семантики равенства используется денотационное, операционное и аксиоматическое описание. *Операционная семантика* связана с подстановками (замена, продукция) и определяется в терминах операций, приводящих к вычислениям алгоритмов. При этом фиксируется порядок и динамика выполнения операций. Денотационный подход к семантике предпочитает статическое описание в терминах математических свойств объектов, а аксиоматической - специфицирует свойства объектов в рамках некоторой логической системы, содержащей правила вывода формул и/или интерпретаций.

Продукция или правила подстановки общего вида - это $\lambda \to \rho$, где λ и ρ - произвольные слова в фиксированном алфавите. *Нормальный алгоритм Маркова* [6.1] представляет собой упорядоченный набор правил, некоторые из них отмечены как завершающие. Применение правила $\lambda \to \rho$ к слову ϕ состоит в подстановке слова ρ вместо самого левого слова λ в ϕ. *Вычисление* заканчивается, когда применяется завершающее правило, состоящее в порождении одного слова.

Языки *спецификации программ* или универсальные языки (Z, *VDM*, RAISE [6.5, 6.7-6.10]) базируются на аппарате математической логики и теории множеств и требуют от пользователей математической подготовки при применении их в трудно формализуемых областях -описание трансляторов с ЯП, *системы реального времени*, где правильность и *точность* программ основополагающие. На формальную спецификацию, разработку аксиом и теорем требуется несоизмеримо больше времени, чем в обычных языках программирования. Кроме того, формальные *спецификации программ* более громоздкие и требуют много времени при прокручивании таких программ за столом и интерпретации их на редких инструментальных средствах математического доказательства.

Эти особенности языков формальной спецификации препятствовали практическому их использованию. Их фактически отодвинул более конструктивный и наглядный стиль представления программ на языке *UML*, предоставив пользователям аппарат мышления объектами реального мира, диаграммным представлением их взаимодействия и многочисленными инструментами. В настоящее время интерес к формальным методам доказательства программ на основе спецификаций снова возник [6.15, 6.16], и поэтому студентам с математическим мышлением будет интересно познакомиться с особенностями техник спецификации и формального доказательства программ.

6.1.1. VDM-спецификация программ

Язык *VDM* (Vienna Development Method) разработан в венской лаборатории компании IBM для описания языков типа ПЛ/1, трансляторов и систем со сложными структурами данных [6.7, 6.8]. Главная его цель -

специфицировать правильно программу и описать набор утверждений для ее доказательства, принося в жертву скорость разработки и эффективность, даже если полученная программа громоздка и не всегда удобна в использовании, но является правильной.

Этот язык имеет математическую символику, которая легко воспринимается математически подготовленными студентами последних курсов университетов за 5-6 лекций. В языке содержатся следующие типы данных:

X - натуральные числа с нулем;
N - натуральные числа без нуля;
Int - целые числа;
$Bool$ - булевы;
$Qout$ - строки символов;
$Token$ - знаки и специальные обозначения операций.

Функция в языке - это определение свойств структур данных и операций над ними аппликативно или императивно. В первом случае функция специфицируется через комбинацию других функций и базовых операций (через выражения), что соответствует синониму *функциональный*.Во втором случае - значение определяется описанием алгоритма, что соответствует синониму *алгоритмический*. Например, спецификация функции вычисления минимального значения из двух значений $N_1,\ N_2$ в *VDM* имеет вид:

$$\min N_1 N_2 \to N_3,$$
$$\min(x,y) = \text{ if } x < y \text{ then } x \text{ else } y \text{ - алгоритмическое описание.}$$

Объекты языка VDM. Все объекты строятся иерархически. Элементами данных, с которыми оперируют функции, могут быть множества, деревья, последовательности, отображения, а также более сложные структуры, образованные с помощью конструкторов.

Множество может быть конечное и обозначается $X - set$. При работе с множеством используются операции $\in$, $\subseteq$, $\cap$, $\cup$ и др. Язык имеет правила проверки правильности задания этих операций. Пример, $x \in A$ будет корректным только тогда, когда A является подмножеством множества, которому принадлежит X. Пример дистрибутивного объединения дан ниже:

$$union\{(1,2),(0,2),(3,1)\} = (0,1,2,3).$$

Списки (*последовательности*) - это цепочки элементов одинакового типа из множества X. Операция len задает длину списка, а $inds$ - номера элементов списка.

Например, $inds\ lst = (i \in X[f \le i \le len])$.

К списковым операциям относится взятие первого (головы) элемента списка - hd и остатка (хвоста) после удаления первого элемента из списка - tl.

Например, $hd(a,b,c,d) = (a)$, $tl(a,b,c,d) = (b,c,d)$.

Могут использоваться также операция конкатенации (соединение двух списков) и операция дистрибутивной конкатенации.

Дерево - это конструкция mk, позволяющая объединять структуры разной природы (последовательности, множества и отображения). Элементы деревьев могут конструироваться в виде составных объектов, а также применяется деструктор для именования констант, вносимых в ранее определенный составной объект.

Пример. Пусть t - переменная типа Время, значение которой - 10 ч. 30 мин, тогда конструкция $let\ mk - \text{Время}(h,m) = t\ tin$ определяет значение $h = 10$, а $m = 30$.

Отображение - это конструкция map, позволяющая создавать абстрактную таблицу из двух столбцов: ключей и значений. Все объекты таблицы принадлежат одному типу данных - множеству. Операция dom позволяет строить множество ключей, а rng - множество его значений. Кроме того, есть операции исключения строки, слияния двух таблиц и др.

Приведенные конструкции используются для *спецификации программы* и, в частности, начального состояния с инвариантными свойствами, в качестве которого используется inv функция, содержащая описание типов аргументов, результата и операций самой функции. Для проверки правильности *спецификации программы* средствами языка *VDM* задаются пред- и постусловия, аксиомы и утверждения.

Предусловие - это предикат с операцией, к которой обращается программа после получения начального состояния для определения правильности выполнения или фиксации ошибочной ситуации.

Утверждение задает описание операций проверки правильности программы в разных ее точках. Операторы программы изменяют состояние переменных в заданной точке, а операции утверждений анализируют ее (например, после операции работы с БД) в целях определения правильности выполнения этой операции. При возникновении непредвиденной ситуации, аксиомы и утверждения должны предусматривать соответствующие действия.

Постусловие - это предикат, который - истинный после выполнения предусловия, завершения текущих операции в заданных точках при выполнении инвариантных *свойств программ*.

Метод *VDM* предусматривает пошаговую детализацию *спецификации программ*. На первом уровне строится грубая спецификация - модель в языке *VDM*, которая постепенно уточняется, пока не получится окончательный текст описания программы.

Разработка спецификации проводится по следующей схеме:

1. Определение терминов, которыми будет специфицироваться программа.
2. Описание понятий и объектов, для обозначения которых используется *денотат*, идентифицируемый с помощью некоторого имени (или фразы).
3. Описание инвариантных свойств программы.
4. Определение операций над структурами программы (например, ввести объект, удалить и др.), изменяющие ее состояние и сохранение инвариантных свойств.

При переходе от одного шага детализации к другому *модель программы* детализируется и постепенно становится ближе к конечному описанию. Функции – это операции, которые уточняются при детализации структуры программы на каждом шаге спецификации и описания поведения модели.

При реальном выполнении спецификация исполняется итерационно. На первом уровне проверяется только свойства модели программы при заданных ограничениях независимо от среды. Затем используется уточненная и расширенная спецификация с набором формальных утверждений. И так до тех пор, пока окончательно не будет завершен процесс пошагового доказательства спецификации.

Для демонстрации возможностей *VDM* языка рассмотрим задачу поиска ("Поиск") в каталоге ($catalR$) репозитария компонентов имени компонента C и сравнения его с заданным в запросе пользователя. В случае совпадения имен проверяются параметры, и при их совпадении из каталога извлекается код компонента и передается пользователю.

Спецификация переменных программы "Поиск"

$$
\begin{aligned}
repoz \;::=\; & developers : dl \to Init - set \\
& catalR : cat \to Init - set \\
& role : inst \to facet \\
facet \;::=\; & autors : Milk \\
& title : N \\
user \;::=\; & developer : Itn \\
& free : Bool,
\end{aligned}
$$

где $developers$ – сведения о разработчике компонента C ; $facet$ – переменная, в которую посылается код компонента, выбранного из каталога $catalR$ репозитария $repoz$ при совпадении имен в каталоге и запросе; $role$ – переменная, в которой хранится текущий элемент из репозитария, найденный по фасете компонента с номером N для $user$; $autors : Milk$ – имя разработчика компонента; $free : Bool$ – переменная, которая используется для задания признака – компонент не найден или к нему никто не обращался.

Описание инвариантных свойств программы

$$
\begin{aligned}
& type\ inv - repoz : repoz \to Bool \\
& inv - repoz(dev) = \\
& let\ mk\ repoz(cd, c, role) = dev\ in \\
& (\forall i \in dom\ cd) \\
& (\forall i \in cd(i) \\
& ((\exists j \in dom\ cd\ \&\ i \in c)\ \&\ \exists a \in elems\ role(i), autors(a \in domcd) \\
& \&\ free = false \Leftrightarrow developer = catalR\ \&\ facet(N) = role \ldots
\end{aligned}
$$

Операторы программы проверяют список имен компонентов в каталоге, который содержит N элементов типа set . Если они совпадают с именем в запросе, результат сохраняется в $role$.

Доказательство инвариантных *свойств программ* должно проводиться автоматизированным способом с помощью специально созданных инструментальных средств поддержки *VDM* языка.

6.1.2. Спецификация программ средствами RAISE

RAISE-метод и *RSL*-спецификация (RAISE Specification Language) [6.9, 6.10] были разработаны в 80-х годах как результат предварительного исследования формальных методов и их пополнения новыми возможностями. Метод содержит нотации, техники и инструменты для конструирования программ и доказательстве их правильности. Он имеет программную поддержку в виде набора инструментов и методик, которые постоянно развиваются и используются при доказательстве правильности программ, описанных в *RSL* и ЯП (C++ и Паскаль). Язык *RSL* содержит абстрактные параметрические типы данных (алгебраические спецификации) и конкретные типы данных (модельноориентированные), подтипы, операции для задания последовательных и параллельных программ. Он предоставляет аппликативный и императивный стиль спецификации абстрактных программ, а также формальное конструирование программ в других ЯП и доказательство их правильности. Синтаксис этого языка близок к синтаксису языков C++ и Паскаль.

В *RSL*-языке имеются предопределенные абстрактные типы данных и конструкторы *сложных типов данных*, такие как произведение ($product$), множества ($sets$), списки ($list$), отображения (map), записи ($record$) и т.п. Далее рассмотрим некоторые конструкторы *сложных типов данных*.

Произведение типов – это упорядоченная конечная последовательность типов $\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n$ произведения ($product$) $\mathrm{T}_1 \times \mathrm{T}_2 \times, \ldots, \times \mathrm{T}_n$. Представитель типа имеет вид ($v_1, v_2, \ldots, v_n$), где каждое v_i –это значение типа T_i. Компонент произведения можно получить операцией get и переслать set, т.е.

$$get\ component(i, d) = get\ value(i, d),$$
$$set\ component(d, i, val) = d \Rightarrow \nabla(I \rightarrow val).$$

Количество компонентов произведения d находится таким образом:

$$size(d) = id\nabla(null(couterinc(counter))).$$

Конструктор произведения d_1 и d_2 строит произведение $d_1 \times d_2$ вида:

$$product(d_1, d_2) = id\nabla(size(d_1) \Rightarrow couter\ 1)\nabla(null(couter\ 2)inc\ couter\ 2))).$$

Для каждого конкретного типа $product(\mathrm{T}_1 \times \mathrm{T}_2 \times, \ldots, \times \mathrm{T}_n)$ можно построить конструктор значения этого типа из отдельных компонентов произведения таким образом:

$$make\ product(value_1, \ldots, value_n) = (value_i \Rightarrow 1)\nabla \ldots \nabla(value_n \Rightarrow n),$$

где каждое значение $value_i$ имеет тип T_i, а результирующее значение – тип произведения $\mathrm{T}_1 \times \mathrm{T}_2 \times, \ldots, \times \mathrm{T}_n$

Списки типов – это последовательность значений одного типа $list\ \mathrm{T}$, могут быть конечным списком типов T^k и неконечными списком типов T^n. В качестве структур данных типа списка может быть бинарное дерево, в котором есть голова ($head$) и сын ($tail$), который следует за ним в списке, и хвост. К операциям списка относится операция hd – взятия первого элемента списка, т.е. головы, и операция tl – хвоста остальных элементов (аналогично как в *VDM*).

Функция $Caddr(I) = L \Rightarrow tail \Rightarrow tail \Rightarrow Head$ выбирает из списка I -элемент. Индекс элемента помогает выбрать нужный элемент списка:

$$Index(I, idx) = L(idx) = \text{ while } (\neg \text{ is } null(idx)) \text{ do } ((L \Rightarrow tail \Rightarrow L)\nabla dec(idx))L \Rightarrow Head.$$

Для определения количества элементов в списке выполняется функция:

$$len(L) = (ld\ \nabla\ null(result))$$
$$\text{while } (L \Rightarrow) \text{ do } ((L \Rightarrow tail \Rightarrow tail \Rightarrow L)\ \nabla\ inc(result))$$
$$result \Rightarrow .$$

Элемент списка находится так:

$$elem(L) = (ld\ \nabla\ empty(result))$$
$$\text{while } (L \Rightarrow) \text{ do } ((L \Rightarrow tail \Rightarrow L)\ \nabla$$
$$(result \uparrow (L \Rightarrow head \Rightarrow) \Rightarrow elem) \Rightarrow result)$$
$$result \Rightarrow .$$

Аналогично можно представить функции конкатенации, преобразование типов данных, добавления элемента в голову и *хвост списка* и др.

Отображение – это структура (map), которая ставит в соответствие значениям одного типа значение другого типа. Вместе с тем отображение – это бинарное отношение декартова произведения двух множеств как совокупности двухкомпонентных пар, в которых первый компонент – arg содержит элементы аргументов отображения, а второй компонент res – соответствующие элементы значений этого отображения.

В языке имеются разные допустимые операции над отображениями: наложение, объединение, композиция, срез и др. Среди этих видов отношений рассмотрим, например, композицию отображений (m_1, m_2):

$$(ld\ \nabla\ (compose(m_1, m_2) \Rightarrow m))\ apply\ (m, elem)$$
$$apply\ to\ composition\ (m_1, m_2, elem) =$$
$$= (ld\ \nabla\ (image(elem, m1) \Rightarrow s)\ restrict\ (m_2, s) \Rightarrow map$$
$$(ld\ \nabla\ (map\ \text{getname}\ elem \Rightarrow name))\ getvalue\ (name, map).$$

При этом используются функции:

$$\text{Apply } (m, \text{ elem}) = \text{image (elem},m) \text{ elem } \Rightarrow,$$
$$\text{Apply } (m, \text{ elem}) = \text{getvalue (elem},m) \text{ elem } \Rightarrow .$$

Запись – это совокупность именованных полей. Этот тип соответствует типу $record$ в языке Паскаль и $struct$ в языке C++. В языке RAISE для записи определено два конструктора – $record$, $shurtrecord$, описание которых имеет вид

$$type\ record\ id =$$
$$type\ mk\_id\ (short_record\ id) ::=$$
$$destr\_id_1 : type\_expr_1 \leftrightarrow recon\_id$$
$$\ldots$$
$$destr\_id_n : type\_expr_n \leftrightarrow recon\_id.$$

Идентификатор mk_id – это конструктор типа $record$, для которого задается деструктор $destr\_id_n$ как функции получения значения компонентов записи.

Объединение – это конструктор $union$ для объединения типов $typeid = id_1, id_2, \ldots, id_n$, при котором тип id получает одно из значений в списке элементов.

Конструктор типа имеет вид

$$\text{type } id = id\_\text{from}\_id_1(id\_\text{to}\_id_1 : id_1) \mid \ldots \mid id\_\text{from}\_id_n(id\_\text{to}\_idn : id_n).$$

Операции над самим типом не определены в языке RAISE.

Рассмотренные формальные структуры данных языков *VDM* и RAISE предназначены для математического описания программ с помощью утверждений и конструирования новых структур данных, необходимых для проектируемых программ. Средства этих языков фактически – элементы *спецификации программ*, по которым проще проверять правильность программ методами верификации или доказательства, составляя при этом, как и в случае *VDM*, пред, постусловия и утверждения для проведения доказательства программы по ее спецификациям.

6.1.3. Спецификации задач концепторным языком

Для постановки сложных математических задач (суммирование бесконечных рядов, теоретикомножественных операций с бесконечными множествами, гильбертов оператор и др.) и задач искусственного интеллекта (игры, распознавание образов и др.) предложен *общематематический процедурный язык*, так называемый концепторный язык – КЯ [6.17]. В этом языке процесс описания сложной задачи проводится путем обоснования решения задачи с математической точки зрения, затем формального описания постановок задач и, наконец, делается переход к алгоритмическому описанию.

Средства спецификации сложных задач. Основу КЯ составляет теоретикомножественный язык, который содержит декларативные и императивные средства теории множеств ЦермелоФренкеля. Ядро содержит набор элементов (типы, выражения, операторы) и средства определения новых типов, выражений и операторов.

Декларативные средства КЯ – это типизированный, многосортный логикоматематический язык задания *выражений* и структуризации множества значений (*денотат*). Выражения состоят из термов и формул, термы обозначают объекты ПрО, а формулы – утверждения об объектах и отношениях между ними. К конструкторам составных типов и формул относятся *функторы*, предикаты, конекторы и субнекторы.

Функтор – это конструктор, преобразующий термы в термы. *Предикаты* превращают термы в формулы, *конекторы* включают в себя логические связки и кванторы для преобразования одной формулы в другую. Субнектор (дескриптор) – это конструктор построения термов из выражений и формул. Конструкторы термов – это традиционные арифметические и алгебраические операции над числовыми множествами и вещественными функциями. Конструкторы формул включают в себя предикаты, состоящие из предикатных и числовых символов, а также конекторы, состоящие из логических связок, кванторов и конструкторов теории множеств.

Императивные средства КЯ – это операторы и процедуры для описания объектов ПрО с помощью концепторов, состоящих из разделов для определения объектов решаемой задачи и действий над ними. Каждый концептор – это именованный набор определений и действий со следующей структурой описания:

```
концептор К (< список параметров >)
 <список импортных параметров>
 <определение констант, типов, предикатов>
 <описание глобальных переменных>
 <определение процедур>
 начало К
 <тело концептора>
 конец К.
```

Концептор – это декларативное описание объектов и императивное описание операторов вычисления выражений тела. Рассматривается два случая:

1. декларативный концептор состоит из определений параметров и типов;
2. императивный концептор – это тело из операторов задач.

Декларативный концептор задает описание объектов и понятий, связанных с математической постановкой задачи, а описание метода ее решения с помощью императивных концепторов. Концепторное описание – это формальная спецификация задачи, которую можно трансформировать до алгоритмического описания и верификации.

Если полученный концептор неэффективен, то для повышения эффективности строится алгоритм, эквивалентный данному концептору. Он строится аппроксимацией концепторного решения путем замены неконструктивных объектов и неэффективных операций конструктивными и более эффективными аналогами.

Формализация КЯ. Общая схема формализации декларативной и императивной частей КЯ расширяется логикоматематическим языком, традиционными структурными операторами (присваивание,

последовательность, цикл и т.п.), а также теоретикомодельными (денотационными) и аксиоматическими средствами формализации неконструктивной семантики КЯ.

Денотационный подход состоит в определении *семантики* языка путем подстановки каждому выражению соответствующего элемента из множества *денотатов* функции ϕ интерпретации символов сигнатуры языка. Каждой константе $c \in C$, функциональному символу $f \in F$ и *предикатному символу* $p \in P$ сопоставляется объект из множества *денотат*. Этот способ интерпретации семантики выражений и операторов языка аналогичен *денотационной семантики* ЯП. Главное отличие семантики КЯ от семантики программ – это ее неконструктивность. С каждым КЯ можно связать некоторую дедуктивную теорию, которая отражает свойства концепторов.

Формальная дедуктивная теория строится путем выделения из множества всех формул подмножества аксиом и правил вывода. Для каждой пары R_1, R_2 формул дедуктивной теории и каждого оператора I создается операторная формула $\{R_1\}I\{R_2\}$ с утверждением, что если R_1 истинно перед выполнением оператора I, то завершение оператора I обеспечивает истинность R_2, т.е. формула R_1 – предусловие, а R_2 – постусловие оператора I. С помощью неконструктивных объектов и неразрешимых формул этой теории можно адекватно описывать свойства неэффективных процедур.

Аксиоматическое описание КЯ – это аксиомы и утверждения относительно концепторного описания и проведения дедуктивного доказательства и верификации этого описания.

Логико-алгебраические спецификации.При использовании этих спецификаций ПрО представляется в виде *алгебраической системы* с помощью соответствующих носителей, сигнатуры и трех принципов. *Первый принцип* – логико-алгебраическая спецификация ПрО и уточнение понятий ПрО, *второй принцип* – описание свойств ПрО в виде аксиом, которые формулируются в *языке предикатов* первого порядка и *хорновских* атомарных формул, и, наконец, *третий принцип* – это определение термальных моделей из *основных термов* спецификации. Логико-алгебраические спецификации можно ограничить хорновскими формулами из-за простоты аксиом и для упрощения процесса автоматического доказательства теорем. Отношения в сигнатурах спецификаций заменяются булевыми функциями.

Техника доказательного проектирования. Средства концепторной спецификации сложных, алгоритмически не разрешимых задач положены в основу формализованного описания *поведения дискретных систем*. Для описания свойств аппаратнопрограммных средств динамических систем применяются логико-алгебраические спецификации КЯ, техника описания которых включает два этапа.

На первом этапе дискретная система S рассматривается как черный ящик с конечным набором входов, выходов и состояний. Области значений входов и выходов – произвольные, а функционирование системы S – это набор частичных отображений и операций *алгебраической системы*. Они образуют *частичную алгебру*,формальное описание которой выполняется с помощью алгебраических спецификаций и является программой моделирования состояний дискретной системы.

На втором этапе система S детализируется в виде совокупности взаимозависимых подсистем $S_1, \ldots, S_n$, каждая из которой описывается алгебраической спецификацией. В результате получается спецификация системы S из функций переходов и выходов, для которых необходимо доказывать корректность. Процесс детализации выполняется на уровне элементной базы или элементарных программ и сопровождается доказательством их корректности. В конечном итоге получается система S, эквивалентная исходной спецификации. Примеры доказательства систем приведены в [6.17]. Рассмотрим один из них.

Пусть требуется построить спецификацию натуральных чисел из множества этих чисел с *сигнатурой операций* $\sum = (+, \times, \leq)$. При построении используется число 0 и функция следования $s : N \to N'$. Спецификация состоит из следующих аксиом:

1. $\mathrm{x} + 0 = x$,
2. $x + s(y) = s(x + y)$,
3. $x \times 0 = 0$,
4. $x + s(y) = s(x \times y) + x$,
5. $0 \leq x$,
6. $x \leq y \supset s(x) \leq s(y)$
7. $s(x) < s(y) \supset x \leq y$

При этом *алгебраические системы* становятся многоосновными алгебрами, а аксиомы – спецификациями: тождественными и квазитождественными. *Алгебраические* спецификации – наиболее используемые, поскольку для них существуют эффективные алгоритмы выполнения, которые преобразуют спецификации в ЯП высокого уровня. Поэтому такие языки называют языками выполняемых логико-алгебраических спецификаций. Их *операционная семантика* основана на *переписывании термов*, а создаваемая алгебраическая спецификация получает логическую семантику, используемую при доказательстве теорем.

6.2. Методы доказательства правильности программ

Формальные методы тесно связаны с математическими техниками спецификаций, верификацией и доказательством правильности программ. Эти методы содержат математическую символику, формальную нотацию и аппарат вывода. Правила доказательства являются громоздкими и поэтому на практике редко используются рядовыми программистами. Однако с теоретической точки зрения они развивают логику применения математического метода индукции при проверке правильности программ. На основе *спецификации программ* проводится частичное и полное *доказательство* правильности программ [6.4, 6.5].

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

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

Теорема 6.1. Если выполнены все действия метода индуктивных утверждений для программы, то она частично правильна относительно утверждений A, B, C.

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

6.2.1. Характеристика формальных методов доказательства

Наиболее известными формальными методами доказательства программ являются метод рекурсивной индукции или утверждений Флойда, Наура, метод структурной индукции Хоара и др. [6.4, 6.5, 6.18, 6.19].

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

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

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

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

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

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

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

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

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

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

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

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

Процесс формального доказательства правильности программ методом математической индукции зарекомендовал себя как система правил статической проверки правильности программ за столом для обнаружения в них формальных ошибок. С помощью этого метода можно доказать истинность некоторого предположения $\mathrm{P}(n)$ в зависимости от параметра n для всех $n \geq n_0$, и тем самым доказать случай $\mathrm{P}(n_0)$. Исходя из истинности $\mathrm{P}(n)$ для любого значения n, доказывается $\mathrm{P}(n+1)$, что достаточно для доказательства истинности $\mathrm{P}(n)$ для всех $n \geq n_0$.

Путь доказательства следующий. Пусть даны описание некоторой правильной программы (ее логики) и утверждение A относительно этой программы, которая при выполнении достигает некоторой определенной точки. Проходя через эту точку n раз, можно получить справедливость утверждения $\mathrm{A}(n)$, если индуктивно доказать, что:

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

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

6.2.2. Доказательство конкретности с помощью утверждений

Рассмотрим формальное доказательство программы, заданной структурной логической схемой и совокупностью утверждений, задаваемых логическими операторами, комбинациями переменных (true/false), *операциями (конъюнкция*, дизъюнкция и др.) и кванторами всеобщности и существования (табл. 6.1).

Таблица 6.1. Список логических операций

Логические операции

Название	Примеры	Значение
Конъюнкция	$x \,\&\, y$	x и y
Дизъюнкция	$x * y$	x или y
Отрицание	$\neg x$	не x
Импликация	$x \rightarrow y$	если x то y
Эквивалентность	$\mathrm{x} = \mathrm{y}$	x равнозначно y
Квантор всеобщности	$\forall \mathrm{x}\ \mathrm{P}(\mathrm{x})$	для всех x, условие истинно
Квантор существования	$\exists \mathrm{x}\ \mathrm{P}(\mathrm{x})$	существует x, для которого $\mathrm{P}(\mathrm{x})$ истина

Цель алгоритма программы – построение для массива целых чисел T длины $N(array\mathrm{T}[1:N])$ эквивалентного массива T' той же длины N, что и массив T. Элементы в массиве T' должны располагаться в порядке возрастания их значений. Данный алгоритм реализуется сортировкой элементов исходного массива T по их возрастанию. Доказательство правильности алгоритма сортировки элементов массива T проводится с использованием ряда утверждений относительно элементов этого алгоритма, которые описываются пунктами П1– П6.

1. Входное условие алгоритма задается в виде начального утверждения:

   $$\mathrm{A}_{beg} : (\mathrm{T}[1:N] \text{ - массив целых}) \,\&\, (\mathrm{T}'[1:N] \text{ - массив целых })$$

   Выходное утверждение A_{end} – это конъюнкция таких условий:

   $$(\mathrm{T} \text{ - массив целых})(\mathrm{T}' \text{ - массив целых}),$$
   $$(\forall i, \text{ если } i \leq N, \text{ то } \exists j(T'(i) \leq T'(j))),$$
   $$(\forall i, \text{ если } i \leq N, \text{ то } (T'(i) \leq T'(i+1)),$$

   т.е.

   $$\mathrm{A}_{beg} \text{ - это } (\mathrm{T}[1:N] \text{ - массив целых}) \,\&\, (\mathrm{T}'[1:N] \text{ - массив целых })$$
   $$\& \,\forall i, \text{ если } i \leq N, \text{ то } \exists j(T'(i) \leq T'(j)),$$
   $$\& \,\forall i, \text{ если } i \leq N, \text{ то } (T'(i) \leq T'(i+1)).$$

   Расположение элементов массива T в порядке возрастания их величин в массиве T осуществляется алгоритмом *пузырьковой сортировки*, суть которого заключается в предварительном копировании массив T в массив T', а затем проводится сортировка элементов согласно условия их возрастания. Алгоритм сортировки представлен на блоксхеме (рис. 6.2).

   Операторы алгоритма размещены в прямоугольниках, условия выбора альтернативных путей – параллелограммами, точки с начальным A_{beg} и конечным A_{end} условиями и состояниями

алгоритма – кружками. В кружках также заданы: начальное состояние – 0, состояние после обмена местами двух соседних элементов в массиве T – одна звездочка, состояние после обмена местами всех пар за один проход всего массива T – две звездочки.

Кроме уже известных переменных T, T' и N, в алгоритме использованы еще две переменные: i – целое и M – булева переменная, значением которой являются логические константы $true$ и $false$.

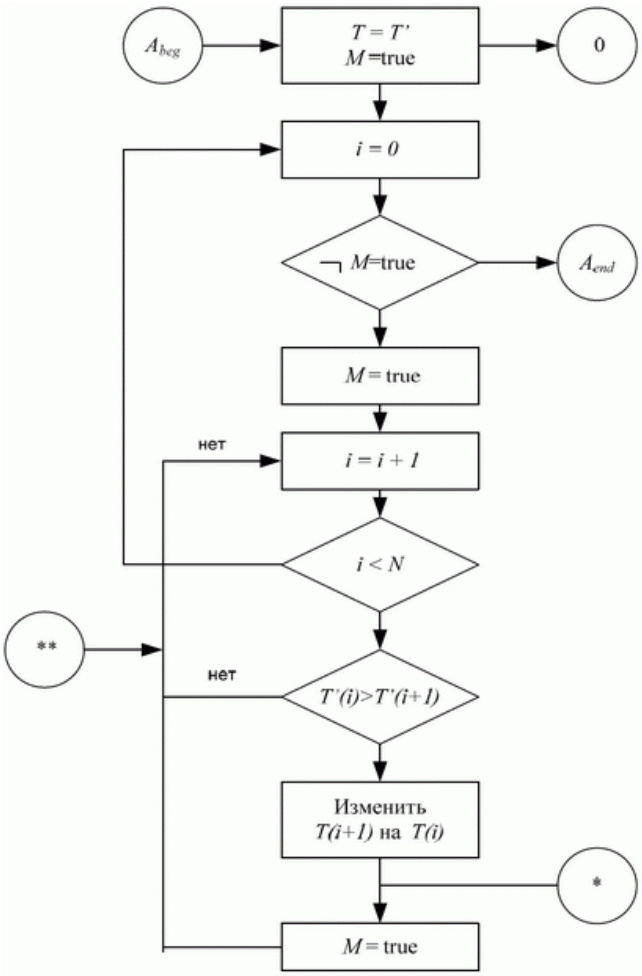
2. Для доказательства того, что алгоритм действительно обеспечивает выполнение исходных условий, рассмотрим динамику их выполнения последовательно в определенных точках алгоритма.

   Заметим, что точки делят алгоритм на соответствующие части, правильность любой из них обосновывается в отдельности.

   Так, оператор присваивания означает, что для всех i ($i \le N i \ge 0$) выполняется ($T'[i] := T[i]$). Результат выполнения алгоритма в точке с нулем может быть выражен утверждением

$$(T[1:N] \text{ - массив целых}) \,\&\, (T'[1:N] \text{ - массив целых})$$
$$\&\, (\forall i, \text{если } i \le N (T[i] = T[i])).$$

   Доказательство очевидно, поскольку за семантикой оператора присваивания (поэлементная пересылка чисел из T в T') сами элементы при этом не изменяются, к тому же в данной точке их порядков T и T' одинаковый. Итак, получили, что выполняется условие б) исходного утверждения.



увеличить изображение

Рис. 6.2. Схема сортировки элементов массива T

Заметим, что первая строка доказанного утверждения совпадает с условием а) исходного утверждения A_{end} и остается справедливой до конца работы алгоритма, поэтому в следующих утверждениях приводиться не будет.

В точке с одной звездочкой выполнен оператор $(i < N)\,(T'(i)) > T'(i+1) \to (T'(i) \text{ и } T'(i+1),$ который меняет местами элементы.

В результате работы оператора будет справедливым такое утверждение:

$$\exists i, \text{если } i < N, \text{то } (T'(i) < T'(i+1)),$$

которое является частью условия в) утверждения A_{end} (для одной конкретной пары смежных элементов массива T'). Очевидно также, что семантика оператора обмена местами не нарушает условие б) выходного утверждения A_{end}.

В точке с двумя звездочками выполнены все возможные обмены местами пар смежных элементов массива T' за один проход через T', т.е. оператор обмена работал один или больше раз. Однако пузырьковая сортировка не дает гарантии, что достигнуто упорядочение за один проход по массиву T', поскольку после очередного обмена индекс i увеличивается на единицу независимо от того, как соотносится новый элемент $T'(i)$ с элементом $T'(i-1)$.

В этой точке также справедливо утверждение

$$\exists i, \text{ если } i < N, \text{ то } T'(i) < T'(i+1).$$

Часть алгоритма, обозначенная точкой с двумя звездочками, выполняется до тех пор, пока не будет упорядочен весь массив, т.е. не будет выполняться условие в) утверждения A_{end} для всех элементов массива $T' :, \text{если} i < N, \text{то} T'(i) < T'(i+1)$.

Итак, выполнение исходных условий обеспечено порядком и соответствующей семантикой операторов преобразования массива.

Доказано, что выполнение алгоритма программы завершено успешно, что означает ее правильность.

3. Этот алгоритм можно представить в виде серии теорем, которые доказываются. Начиная с первого утверждения и переходя от одного преобразования к другому, определяется индуктивный путь вывода. Если одно утверждение - истинно, то истинно и другое. Иными словами, если дано первое утверждение A_1 и первая точка преобразования A_2, то первая теорема - $A_1 \to A_2$.

   Если A_3 - следующая точка преобразования, то второй теоремой будет $A_2 \to A_3$.

   Таким образом, формулируется общая теорема $A_i \to A_j$, где A_i и A_j - смежные точки преобразования. Эта теорема формулируетсятак, что если условие "истинное" в последней точке, то истинно и выходное утверждение $A_k \to A_{end}$.

   Следовательно, можно возвратиться к точке преобразования A_{end} и к предшествующей точке преобразования. Доказав, что $A_k \to A_{end}$ верно, значит, верно и $A_j \to A_{j+1}$ и так далее, пока не получим, что $A_1 \to A_0$.

4. Далее специфицируются утверждения типа if - $then$.
5. Чтобы доказать, что программа корректная, необходимо последовательно расположить все утверждения, начиная с A_1 и заканчивая A_{end}, этим подтверждает истинность входного и выходного условий.
6. Доказательство алгоритма программы завершено.

6.2.3. Валидация сценариев требований

Рассматривается подход к проверке требований, которые заданы в модели требований, построенной с использованием сценариев и акторов, как *внешней сущности* по отношению к разрабатываемой системе [6.21].

Сценарий после трансформации - это последовательность взаимодействий между одним или несколькими акторами и системой, в которой актор выполняет цели сценария при взаимодействии с ней. В модели требований сценарий задает несколько альтернативных событий, заданных на языке диаграмм UML. Они разделяются на функциональные (системные) и внутренние, определяющие поведение системы. На основе описания сценарных требования проводится их валидация.

Валидация требований - это процесс выявления ошибок в представлении сценарных требований, он итерационный и состоит из следующих шагов:

1. Формализованное описание требований в виде сценариев;
2. Создание исполняемой модели требований;
3. Создание специальных сценариев для валидации требований;
4. Применение валидационных сценариев к модели требований;
5. Оценивание результатов поведения модели требований;
6. Проверка условий завершения процесса валидации и при обнаружении каких-либо неточностей проводится повторение шагов, начиная с п. 2.

При выполнении сценариев возникают ошибочные ситуации, при которых поведение системы становится недетерминированным.В этих целях проводится контроль покрытия сценариев в модели требований валидационными сценариями в целях обнаружения рисков или ошибок. Создается модель ошибок, покрывающая модель требований системы и включающая типичные ошибки, используемые при выводе сценариев. Составная часть валидации требований в сценариях - определение классов эквивалентности входных и выходных данных, используемых и для синтеза сценариев.

Входная информация для синтеза сценариев - сценарная модель задается на языке взаимодействия и приведена на рис. 6.3.

Эта информация используется при генерации дополнительных сценариев в целях улучшения процесса валидации, автоматического синтеза сценариев модели и получения модели поведения системы.

Модель проверяется неполноту исходных требований или противоречия в требованиях с помощью тестов и модели ошибок.

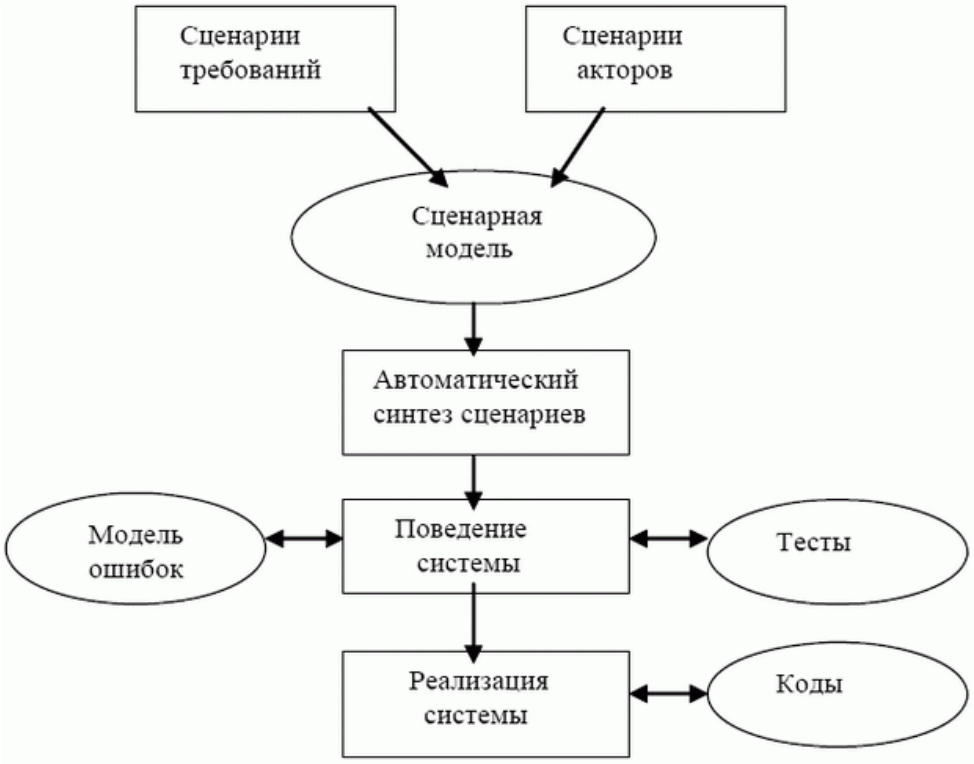


Рис. 6.3. Валидация сценариев требований к системе

Автоматический синтез основан на следующих процедурах:

- валидация требований путем исполнения валидационных сценариев;
- добавление проверенных сценариев к набору валидационных сценариев и их использование как входных данных для синтеза;
- поиск ошибок в сценариях и проверка разных композиций сценариев.Синтез спецификаций сценариев, заданных *диаграммами взаимодействия*, может проводиться в среде системы Rational Rose.

6.2.4. Методы анализа структур программ

Методы анализа структуры программ относятся к доказательству правильности программ [6.20] и состоят в их инспекции независимыми экспертами с участием самих разработчиков. Они проверяют полноту, целостность, однозначность и непротиворечивость определений в программе. Сущность *инспекции* заключается в том, что эксперты пытаются взглянуть на программу "со стороны", подвергнуть ее всестороннему критическому анализу, рассмотреть словесные объяснения разработчиков о способах ее разработки. Цель инспекции – обнаружение ошибок в логике и в исходной программе в статике.

Сквозной контроль может сопровождаться ручной имитацией выполнения программы на выбранных тестах для получения результатов и сравнения их с ожидаемыми. Эти приемы позволяют обнаружить ошибки при многократном просмотре исходного кода, так они не формализованы, их проверка зависит от степени квалификации экспертов группы.

Метод простого структурного анализа ориентирован на анализ графовой структуры программы, в которой каждая вершина – оператор, а дуга – передача управления между операторами. На основе графа определяется достижимость вершин программы и существование выходов всех потоков управления для ее завершения.

Для проведения *анализа потоков данных* граф расширяется специальными указателями переменных, их значениями и ссылками на каждый оператор программы.

При тестировании потоков данных сначала определяются значения предикатов в операторах реализации логических условий, по которым проходили пути выполнения программы. Затем проводится проверка вычислений на арифметических операциях. Для прослеживания путей программы устанавливаются точки, в которых имеются ссылки на переменные до присвоения им значений. Переменной присваивается значение без ее описания либо выполняется повторное описание переменной, к которой нет обращения.

Метод символьной проверки применяется при анализе логики программы и выявлении операторов, по которым не проходит путь вычислений, а также при обнаружении противоречий в описаниилогики программы. Этот метод называют методом анализа потоков управления в символьном виде. Результат проверки – значения переменных, полученных из выражений формул над входными потоками данных.

Метод задается следующими шагами:

1. построение тестового набора данных для заданного пути в виде последовательности операторов символьного выполнения. В случае невозможности построения такого набора делается заключение о нереализуемости (противоречивости) данного пути;
2. определение пути прохождения, при котором выполняются заданные ограничения на входные данные и символьные значения переменных.

Приведем шаги символьной проверки.

Шаг 1. Пусть $\mathrm{P}(\mathrm{X}, Y)$ – программа, выполняемая символически на наборах данных $D = (d_1, d_2, \ldots, d_n)$, где $D \cup \mathrm{X}$ и X – множество входных данных.

Пронумеруем операторы программы $\mathrm{P} = \{P_1, P_2, \ldots, P_n\}$ и обозначим состояние выполнения программы в виде тройки

$$< N, pc, Z >,$$

где N – номер текущего оператора программы P, pc ($partcondition$) – условие выбора пути в программе (вначале $true$) в виде логического выражения над данными D ; Z – множество пар $\{< z_i, e_i > | z_i \in X \cap Y$, в которых z_i – переменная программы, а e_i – ее значение; Y – множество промежуточных и выходных данных.

Семантика символьного выполнения задается базовыми конструкциями ЯП и правилами оперирования символьными значениями, как алгебраическими вычислениями. К базовым конструкциям относятся операторы присваивания, перехода и условные операторы.

В операторе присваивания $Z = e(x, y)$, $x \in X$, $y \in Y$ в выражение $e(x, y)$ подставляются символьные значения переменных x и y, в результате чего получается выражение $e(D)$, которое становится значением переменной $z(z \in \mathrm{X} \cup Y)$. Вхождение в полученное выражение $e(D)$ переменных из Y означает, что их значения, а также значение z не определены.

Оператор перехода – это ссылка к помеченной меткой оператора программы.

Согласно условного оператора "если $\alpha(x, y)$ то $\mathrm{B1}$ иначе $\mathrm{B2}$ " вычисляется выражение $\alpha(x, y)$. Если оно определено и равно $\alpha'(D)$, то формируются логические формулы:

$$\mathrm{pc} \to \alpha'(D), \quad (6.1)$$

$$\mathrm{pc} \to \neg\alpha'(D). \quad (6.2)$$

Если $\mathrm{pc} - false$, то только одна из этих формул может быть выполнимой, а именно если

- формула (6.1) выполнима, то управление передается на оператор $\mathrm{B1}$;
- формула (6.2) выполнима, то управление передается на оператор $\mathrm{B2}$;
- (6.1), (6.2) не выполнимы (т.е. из pc не следует ни $\alpha'(D)$, ни $\neg\alpha'(D)$), тогда один набор данных, который удовлетворяет pc и соответствует части "то" условного оператора и имеется набор данных, соответствующий оператору после "иначе" этого условного оператора.

Таким образом, создаются два пути символьного выполнения в соответствии с формулами: $\mathrm{pc}_1 = \mathrm{pc} \cap \alpha'(D)$ и $\mathrm{pc}_2 = \mathrm{pc} \cap; \alpha'(D)$. Получаем, что $\mathrm{pc} = true$, тогда когда

- для заданного пути формируется pc исходя из семантики условного оператора, преобразующего pc к виду (6.1) или (6.2);
- решаются системы уравнений (6.1) и (6.2) и если решения нет, то это означает невыполнимость пути.

Шаг 2. Определение пути при заданных ограничениях на входные данные проводится так:

- полагаем $\mathrm{pc} = \beta(D)$, где $\beta(D)$ – входная спецификация, когда ее нет, то $\mathrm{pc} = true$;
- производим символьное выполнение операторов, если встречается ветвление, то запоминается состояние в данной точке или выбирается одна из ветвей; выполняется условный оператор, при котором формируется состояние программы с условием pc ;
- в конце пути прохождения по программе определяется функция и набор данных, которым удовлетворяет pc, покрывающий данный путь;
- для промежуточных $\delta(\mathrm{x}, \mathrm{y})$ выходной спецификации $\gamma(\mathrm{x}, \mathrm{y})$ проводится доказательство выполнимости логических формул $\mathrm{pc} \Rightarrow \delta(\mathrm{x}, \mathrm{y})$ и $\mathrm{pc} \Rightarrow \gamma(\mathrm{x}, \mathrm{y})$, где pc – значение текущего условия в данной точке программы.

Доказательства этих формул можно провести верификацией данного участка пути. Выражение формулы декомпозируется на множество неравенств с целью определения несовместимости, мешающей прохождению по данному пути.

При проведении статического анализа программы используются различные инструменты, позволяющие определить ошибки в программе (например, неинициализированные или не использованные переменные). Кроме того, имеются способы автоматизации символьного выполнения программ и контроля в среде языковоориентированной разработки [6.23].

6.3. Верификация и валидация программ

Верификация и *валидация* – это методы анализа, проверки спецификаций и правильности выполнения программ в соответствии с заданными требованиями и формальным описанием программы [6.19, 6.20].

Верификация помогает сделать заключение о корректности созданной программной системы после завершения ее разработки. *Валидация* позволяет установить выполнимость заданных требований путем их просмотра, инспекции и оценки результатов проектирования на этапах ЖЦ для подтверждения того, что проводится корректная *реализация требований*, соблюдение заданных условий и ограничений к системе. *Верификация* и *валидация* обеспечивают проверку полноты, непротиворечивости и однозначности спецификации и правильности выполнения функций системы в соответствии с требованиями.

Верификации и валидации подвергаются:

тесты, *тестовые процедуры* и входные наборы данных.

компоненты системы и их интерфейсы (программные, технические и информационные) и взаимодействия объектов (протоколы, сообщения) в распределенных средах;

описание доступа к БД, средства защиты от несанкционированного доступа к данным разных пользователей;

документация на систему.

Иными словами, основные систематические методы обеспечения правильности программ – *верификация* компонентов и *валидация* требований путем инспектирования для установления соответствия программы заданным спецификациями и требованиям.

6.3.1. Подходы к верификации моделей

Объектная модель и модель распределенного приложения отражают специфику предметной области и принципы взаимодействияобъектов со средой функционирования. Их верификации посвящен ряд работ, в том числе [6.22]. Эта область верификации требует дальнейшего развития и в рамках международного проекта на ближайшие десятилетия будет одним из главных ее направлений.

Верификация объектных моделей основывается на спецификации следующих элементов:

1. Базовых (простых) объектов ОМ, атрибутами которых являются данные и операции объектфункции над этими данными.
2. Проверенных объектов с помощью операций (функции), используемых в качестве теорем, а все операции, которые применяются над их подобъектами, не выводят их из множества состояний объектов.
3. Верифицированных интерфейсов объектов путем доказа тельства правильности передачи типов и количества данных в пара метрах сообщений, заданных в языке IDL. Интерфейс состоит из операций обращения к объекту, который посылает данные другому объекту через сообщение.

Для доказательства правильности спецификации сообщения создается набор утверждений, доказывающий, что для любой пары элементов сообщения, например, A и B, переход от A к B проходит за один шаг.

Действие, выполняемое в промежутке между A и B, приводит к B. При этом часть утверждений проверяет входной параметр и его поступление на вход другого объекта в целях подтверждения его на выходе. Если доказано, что объект, инициированный сообщением, формирует правильный выходной результат – выходной параметр, то сообщение считается правильным.

Доказательство правильности построения ОМ для некоторой ПрО состоит в следующем:

вводятся дополнительные и/или удаляются лишние атрибуты объекта и его интерфейсов в ОМ, доказывается правильность объекта ОМ после изменений спецификации интерфейсов и взаимодействий с другими объектами;

доказывается правильность задания типов для атрибутов объекта, т.е. подтверждения того, что выбранный тип реализует операцию, а множество его значений определено на множестве состояний этого объекта;

доказывается правильность спецификации объектов ОМ и параметров интерфейсов, которые передаются другим объектам.Этим заканчивается заключительное доказательство проверки правильности ОМ.

Верификация модели распределенного приложения – это спецификация процессов SDL (Spesification Description Language), задание модели проверки (model-checking) и индуктивных утверждений. Метод предложен новосибирской школой программирования в [6.12, 6.13]. В нем проверки состоит в редукции системы с бесконечным числом состояний к системе с конечным числом состояний, а также в доказательстве распределенных приложений с помощью индуктивных рассуждений и системы переходов конечного автомата.

Связь между процессами распределенного приложения осуществляется через специальный канал, который передает сообщение с параметрами или без них в качестве сигнала. В него поступает запись после освобождения или чтения очередного сигнала. Процесс задается последовательностью действий, приводящих к изменению переменных, чтению сигнала из канала, записи в канал и очистке канала. Проверка спецификации ограничивается условиями справедливости.

Основные типы данных спецификации в SDL – предопределенные и конструируемые типы данных (массив, последовательность и т.д.). Формулы описываются с помощью предикатов, булевых операций, кванторов, переменных и модальностей. Семантика их определения зависит от последовательных действий (поведений), спецификацией процесса и от момента времени их выполнения.

В предикатах используются локаторы управляющих состояний процессов, контроллеры заполнения каналов (пусто/заполнен канал), а также отношения между переменными и параметрами сигналов. Спецификация процесса состоит из заголовка, контекста, схемы и подспецификации. В заголовке указывается имя и вид процесса, формулы или предикаты.

Контекст – это описание типов, переменных и каналов. Переменная принадлежит процессу, если в ее описании указано место и имя процесса (через точку), которому эта переменная принадлежит. При ее использовании указывается имя переменной с расширением. Если указывается параметр, то в расширенное имя входит имя канала, сигнал и имена параметров, разделенных точками.

В логических условиях используются кванторы всеобщности и существования.

Схема спецификации процесса – это описание условий выполнения и диаграмм процессов. Она инициируется посылкой сообщения во входной канал, который передает сообщение внешней среде для выполнения.

Диаграмма процесса состоит из описаний переходов, состояний, набора операций процесса и перехода на следующее состояние. Набор операций – это действия типа: чтение сообщения из входного канала, запись его в выходной канал, очистка входного и выходного каналов, изменение значений переменных программы Рпроцесса.

Каждая операция определяет поведение процесса и создает некоторое событие. Логическая формула задает модальность поведения спецификации и моменты времени. Процесс, представленный формальной спецификацией, выполняется недетерминировано. Обмен с внешней средой производится через входные и выходные параметры сообщений.

Событие. В каждый момент времени выполнения процесс имеет некоторое состояние, которое может быть отражено в виде снимка, характеризующего это событие, и включает в себя значения переменных, которым соответствуют параметры и характеристики состояний процесса. К событиям процесса относятся:

- отправка сообщения в канал;
- получение сообщения из канала;
- чистка входных и выходных каналов;
- выполнение программ;
- анализ непредвиденного события (взлом канала и др.).

Семантика выполнения процесса определяется в терминах событий и правил с помощью следующего типа утверждения:

любой процесс $\mathrm{P}i \in P$ вызывает событие при чтении или записи сообщения из/в канал, а также при выполнении процесса в узле распределенной системы.

6.3.2. Метод верификации композиции правильных компонентов

Метод верификации композиции компонентов базируется на спецификации функций и временных (*temporal*) свойствах готовых проверенных компонентов (типа reuse) [6.23]. Свойства составного компонента из компонентов повторного использования – reuse проверяются с помощью абстракции и общей компонентной модели (ОКМ). Эта модель состоит из совокупности проверенных компонентов, спецификаций их временных свойств и условий функционирования, которые проверяются с помощью аппарата асинхроннойпередачи сообщений (АПС). Его основу составляет модель проверки (Model Checking) [6.16, 6.23] временных свойств и обнаружения ошибок взаимодействия, возникающих при композиции компонентов.

Модель проверки включает в себя идентификацию правильных компонентов; композицию повторных компонентов по их спецификациям; формирование общей спецификации компонентной системы, составленной из правильных компонентов и др. При этом выполняются следующие условия:

- спецификация компонентов задается в языке диалекта UML [6.23] и содержит описание временных свойств;
- reuse-компоненты задают функции, спецификации интерфейса и временные свойства;
- композиционный аппарат проверяет свойства составных компонентов.

Модель ОКМ – это совокупность специфицированных компонентов и их временных свойств для обеспечения верификации. Свойство компонента определяется исходя из условий среды. Когда компонент многократно используется в составе составного компонента эти свойства должны учитывать возможности среды и связей с другими компонентами композиции. ОКМ проверяется на модели вычислений АПС.

Представители ОКМ-модели могут быть примитивными и составными. Описание свойств примитивных элементов модели проверяется непосредственно с помощью модели проверки, а свойство составного компонента – на абстракции компонента, составленной из примитива и проверенных свойств в интегрированной среде.

Если абстракция слишком громоздкая для проверки, то применяется композиционный подход для проверки сгруппированных свойств компонентов и включения проверенных свойств в абстракцию.

Данный подход может использоваться в распределенных приложениях, функционирующих на платформах CORBA, DCOM и EJB.

Формально каждый компонент C в ОКМ-модели задается в виде $C = (E, I, V, P)$, где E – исходный код компонента; I – интерфейс этого компонента с другими компонентами через передачу сообщений или вызовов процедур; V – множество переменных, определенных в E и связанных со *свойствами множества* временных свойств P, отражающими особенности среды компонента.

Каждое свойство – это пара $(p, \mathrm{A}(p))$, проверяемая на множестве E, где p – свойство компонента C в E, $\mathrm{A}(p)$ – множество временных формул из свойств, определенных на множествах I и V. Свойства компонента C включается в абстракцию P только тогда, когда оно проверено в среде этого компонента.

Композиция компонентов – это совокупность более простых компонентов: $(E_0, I_0, V_0, P_0), \ldots, (E_{n-i}, I_{n-i}, V_{n-i}, P_{n-i})$, определенных на модели компонента C следующим образом. E создается из множества представлений $E_0, E_1, \ldots, E_{n-1}$, связанных между собой интерфейсами из набора интерфейсов $I = \{I_0, \ldots, I_{n-1}\}$, операций связи $Ih(0 < h < n)$ для взаимодействия с другими компонентами;

P – множество временных свойств, определенных на I и V, и проверенных на компонентах E с использованием отдельных свойств $\mathrm{P}_0, \ldots, P_{n-1}$;

V – подмножество $\bigcup_{i=0}^{n-1} V_i$, где V_i – ссылка на свойство i –компонента из C, заданное в P.

Модель вычислений АПС – это *вычислительная модель* системы, заданная на конечном множестве взаимодействующих процессов представленных кортежами:

$$\mathrm{P} = (\mathrm{X}, \sum, Q, \nabla),$$

где X – множество переменных с типом; $\sum$ – *расширенная модель* состояния; Q – очередь сообщений в порядке их поступления; ∇ – множество начальных значений для каждой переменной из X, E и пустое для Q.

При выполнении в вычислительной среде создается модель состояния в виде кортежа (Φ, M, T), где Φ – множество состояний, каждое из которых связано с ассоциативным действием; M – множество типов сообщений; T – набор переходов, определенных на множествах Φ и M

Каждое из состояний переходов – кортеж (r, t, m), где r и t – состояния в Φ и m – тип сообщения во множестве сообщений M.

Семантически каждое действие определяется сегментом программы, составленным из операторов: пустой оператор, присваивания, передачи сообщений, условный и составной операторы и др.

Асинхронная передача сообщений АПС вызывает чередование переходов состояний и действий процессов. Для двух процессов P_1 и P_2 передача сообщения от P_1 к P_2 включает в себя: тип сообщения m из множества M для P_2 и соответствующие параметры. Когда оператор действия выполняется, сообщение m с параметрами ставится вочередь к процессу P_2. Более подробные сведения о верификации компонентов приведены в [6.23].

6.3.3. Перспективные направления верификации программ

По данным, опубликованным в [6.15], ежегодно ошибки в ПО США обходятся в 60 млрд. долларов. Для преодоления этих проблем американские специалисты и специалисты из европейских стран по формальным методам и спецификациям программ приняли решение поставить теоретические достижения в этой области на производственную основу [6.16]. Этому решению предшествовали результаты исследований по формальной *верификации моделей* ПрО и обращений к функциям на языке API проекта SDV фирмы Microsoft, а также проверка безопасности и целостности БД и др. Кроме того, начали активно применяться формальные языки спецификации (RAISE, Z, *VDM* и др.) для разработки приложений. В 2005 г. был сформулирован международный проект по формальной верификации ПО.

Идея создания этого проекта принадлежит Т.Хоару, она обсуждалась на симпозиуме по верифицированному ПО в феврале 2005 г. в Калифорнии. Затем в октябре того же года на конференции IFIP в Цюрихе был принят международный проект сроком на 15 лет по разработке "целостного автоматизированного набора инструментов для проверки корректности ПС".

В нем сформулированы следующие основные задачи:

- разработка единой теории построения и анализа программ;
- построение всеобъемлющего интегрированного набора инструментов верификации для всех производственных этапов, включая разработку спецификаций и их проверку, генерацию тестовых примеров, уточнение, анализ и верификацию программ;
- создание репозитария формальных спецификаций и верифицированных программных объектов разных видов и типов.

В данном проекте предполагается, что верификация будет охватывать все аспекты создания и проверки правильности ПО и, таким образом, она станет главной альтернативой обнаружения ошибок в создаваемых программах.

В связи с тем, что комитет ISO/IEC в рамках стандарта ISO/IEC 12207:2002 провел стандартизацию процессов верификации и валидации ПО и, учитывая цель проекта, проблема создания автоматизированного набора инструментов и репозитария для проверки корректности разных объектов программирования является перспективной.

Репозитарий станет хранилищем программ, спецификаций и инструментов, применяемых при разработках и испытаниях, оценках готовых компонентов, инструментов и заготовок разных методов. В его функции входит:

- накопление верифицированных спецификаций, методов доказательства, программных объектов и реализаций кодов для разных применений;
- накопление всевозможных методов верификации, их оформление в виде, пригодном для поиска и отбора реализованной теоретической концепции для дальнейшего применения;
- разработка стандартных форм для задания и обмена формальными спецификациями разных объектов, инструментов и готовых систем;
- разработка механизмов интероперабельности и взаимодействия для переноса готовых верифицированных продуктов из репозитария в новые распределенные и сетевые среды для использования в новых ПС.

Данный проект предполагается развивать в течение 50 лет. Известно, что более ранние проекты ставили подобные цели: улучшение качества ПО, формализация моделей сервисных услуг, снижение сложности за счет использования ПИК, создание отладочного инструментария для визуальной диагностики ошибок и их устранения и др. Однако эти направления еще не получили должной реализации и предполагаемого коренного изменения в программировании пока не произошло.

Повторное обращение к технике формальной *спецификации программ* еще не означает, что в программе будут отсутствовать ошибки. Остаются для исследований проблемы обнаружения ошибок в программных проектах, в интерпретаторах спецификаций языков программирования, в агентных и аспектных программах и др. Реализация международного проекта по верификации ПО дает перспективу для решения многих проблем обеспечения правильности программ и систем.

Контрольные вопросы и задания

1. Дайте определение формальной спецификации.
2. Назовите категории классификации спецификаций.
3. Определите основные понятия формальной спецификации *VDM*.
4. Определите основные базовые элементы спецификации RAISE.
5. Сравните математические понятия методов *VDM* и RAISE.
6. Определите цель и структуру концепторного языка.
7. Назовите формальные методы доказательства и приведите. их короткую аннотацию.
8. Определите понятия пред- и постусловий, аксиом и утверждений.
9. Опишите, как проходит процесс доказательства программы, заданной спецификацией.
10. В чем проблемы проведения доказательства программ с по мощью формальных методов?
11. Приведите отличие техники формального доказательства от символьного выполнения программ?
12. Дайте определение верификации и валидации.
13. В чем суть композиции верифицированных программ?
14. Расскажите о международном проекте по верификации.