**Ю. Б. Колесов**
**Ю. Б. Сениченков**

# МОДЕЛИРОВАНИЕ СИСТЕМ

## ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

Рекомендовано Учебно-методическим объединением
по университетскому политехническому образованию в качестве
учебного пособия для студентов высших учебных заведений, обучающихся
по направлению подготовки 220100 – «Системный анализ и управление»

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681. 3.06(075.8)
ББК 32. 973.26-018.2я73
К60

**Колесов, Ю. Б.**

К60 Моделирование систем. Объектно-ориентированный подход. Учебное пособие / Ю. Б. Колесов, Ю. Б. Сениченков. — СПб.: БХВ-Петербург, 2012. — 192 с.: ил.

ISBN 5-94157-579-3

Учебное пособие содержит: краткое изложение языка UML — той его части, которая может быть использована как основа языка моделирования сложных динамических систем; описание и возможности предлагаемого авторами нового языка моделирования на базе гибридных автоматов, являющегося расширением UML; исторический обзор и примеры различных подходов к конструированию инструментов моделирования; объектно-ориентированный анализ сложных динамических систем. Книга является второй из трех книг, объединенных общим названием МОДЕЛИРОВАНИЕ СИСТЕМ.

*Для студентов вычислительных специальностей технических вузов*

УДК 681.3.06(075.8)
ББК 32.973.26-018.2я73

**Группа подготовки издания:**

| | |
|---|---|
| Главный редактор | *Екатерина Кондукова* |
| Зам. главного редактора | *Людмила Еремеевская* |
| Зав. редакцией | *Григорий Добин* |
| Редактор | *Анна Кузьмина* |
| Компьютерная верстка | *Ольги Сергиенко* |
| Корректор | *Зинаида Дмитриева* |
| Дизайн серии | *Игоря Цырульникова* |
| Оформление обложки | *Елены Беляевой* |
| Зав. производством | *Николай Тверских* |

Рецензенты:

*Евгенев Г. Б.*, д. т. н., профессор кафедры «Компьютерные системы автоматизации производств» МГТУ им. Баумана

*Ивановский Р. И.*, д. т. н., профессор кафедры «Распределенные вычисления и компьютерные сети» СПбГПУ

Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 15,48.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-579-3

# Оглавление

# Предисловие

Вы держите в руках *том II* книги "Моделирование систем" — объектно-ориентированный подход. Объектно-ориентированное моделирование стало в последнее время основным инструментом проектирования сложных динамических систем.

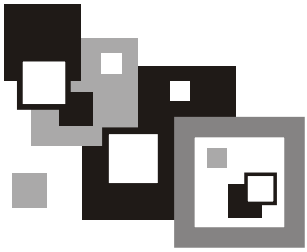
*Глава 1* книги посвящена рассмотрению UML, точнее того его подмножества, что непосредственно может быть использовано на ранней стадии проектирования сложных динамических систем. Язык UML де-факто стал стандартом моделирования сложных программных комплексов, и в частности может быть использован для построения моделей многокомпонентных иерархических систем.

*Глава 2* представляет собой введение в объектно-ориентированный язык моделирования, построенный на гибридных автоматах — графическом представлении гибридных систем. Этот язык лежит в основе программного комплекса MvStudium (**http://www.mvstudium.com**).

*Глава 3* содержит краткое описание существующих объектно-ориентированных языков моделирования и реализующих их программных комплексов.

В *главах 4* и *5* рассматриваются особенности проектирования многокомпонентных систем и обсуждаются возможности объектно-ориентированного анализа сложных систем.

# ГЛАВА 1



# Объектно-ориентированный подход к моделированию

"...сложность является неотъемлемой чертой универсального приложения, предназначенного для решения повседневных задач в реальном мире..."
*Г. Буч, А. Якобсон, Дж. Рамбо [2]*

Объектно-ориентированный подход к моделированию сложных динамических систем был предложен авторами языка SIMULA-67 [6]. Именно они стали использовать специальные модели — классы, описывающие сразу все множество близких по своим свойствам объектов, обладающих внутренней структурой и поведением. Имея такое обобщенное описание, можно уже выбирать конкретный элемент этого множества, необходимый для моделирования конкретного устройства, создавая экземпляр класса и наделяя его конкретными значениями параметров. Так возник новый подход к моделированию сложных многокомпонентных систем, названный *объектно-ориентированным подходом* (ООП), который, как оказалось, может быть применен и для проектирования программных комплексов. Он активно совершенствовался и применялся, прежде всего, при проектировании сложных программных комплексов и дискретных систем. Разрабатываемая программная система сначала моделируется с помощью специального объектно-ориентированного языка моделирования, а только затем превращается в реальный программный комплекс. Наиболее известным является *унифицированный язык моделирования* (Unified Modeling Language или сокращенно UML). Модификации объектно-ориентированной технологии применительно к моделированию сложных многокомпонентных динамических систем возникли сравнительно недавно и называются *объектно-ориентированным моделированием* (ООМ). Они представлены языком моделирования Modelica, реализованы в явном виде в пакетах AnyLogic, MvStudium, Ptolemy II, неявно — в Simulink, Stateflow и других компонентах пакета Matlab.

# Необходимость в унифицированном языке описания моделей

Традиционная технология проектирования сложных систем, не предполагающая предварительного компьютерного моделирования, включает в себя следующие основные этапы: формулировку требований к будущей системе, разработку на их основе проектной документации, создание опытного образца, его испытание на соответствие требованиям и сопровождение промышленного образца.

В идеальном случае полная функциональная спецификация, разработанная системными аналитиками или автоматически с помощью компьютерных технологий, уже является проектным решением, остается малое — построить физическую модель по этой спецификации, которая и будет проектируемой системой. К сожалению, это не так. Создание реальной системы, соответствующей разработанным функциональным спецификациям, требует творческих, не формализуемых инженерных решений и ручного труда.

Даже, казалось бы, полностью формализованная "мягкая часть" современных технических систем — встроенное программное обеспечение — чаще всего выполняется на специальных встроенных ЭВМ или микропроцессорах с особыми каналами обмена, специальными операционными системами и другими особенностями, предполагающими "ручную" доводку разработанного программного обеспечения. Без этого "ручного", плохо формализуемого, этапа трудно обеспечить специальные требования надежной работы системы при заданных температурах, вибрациях, перегрузках, уровнях излучения, ограничениях на объем и вес. В *главе 5* обсуждаются условия, при которых возможна автоматическая генерация встроенного программного обеспечения по функциональной спецификации, и эти условия оказываются еще очень далекими от сегодняшних реалий.

В процессе разработки сложной технической системы наряду с моделями высокого уровня используются так же и модели низкого уровня абстракции, трудно воспринимаемые человеком, трудно формализуемые и требующие ручного труда, при переходе от модели к реальной системе.

На этапе стендовой отработки макета системы часто используется так называемое полунатурное моделирование, когда отдельные блоки системы представлены либо реальными образцами, либо их имитаторами. Теоретически в качестве имитаторов нужно использовать соответствующие функциональные спецификации, однако это не всегда удается, и приходится разрабатывать для них специальные модели.

Наконец, на этапе сопровождения после внесения изменений моделирование должно полностью повторяться, начиная с того уровня, на котором были внесены изменения. На практике, конечно, это выполняется далеко не всегда.

Благодаря возможностям современной вычислительной техники переходить к изготовлению опытного образца стали только после создания и испытания на соответствие предъявляемым требованиям компьютерной модели. Такую технологию можно назвать проектированием на основе моделирования, и она позволяет использовать преимущества компьютерного моделирования уже на самых ранних этапах разработки. Вычислительные эксперименты с моделью еще не существующей системы позволяют проводить более тщательный анализ предлагаемых решений, чем при традиционном проектировании, когда большинство ошибок и несоответствий требованиям обнаруживается на этапе экспериментов с опытным образцом. Описание модели на языке моделирования — это строгое описание ее функциональных свойств. Модель становится единой формальной функциональной спецификацией проектируемой системы, частью проектной документации, доступной всем разработчикам. В цепочке технологических операций появляется новый этап — построение и исследование формальных спецификаций. Этап построения и исследования функциональных спецификаций на основе объектно-ориентированной технологии называется *объектно-ориентированным анализом* и рассматривается в *главе 5*. Сегодня предпочтение отдают проектированию на основе моделирования и средам моделирования, автоматизирующим как можно больше этапов разработки.

Программных сред, поддерживающих технологию проектирования, основанную на моделировании, и способных по заданному описанию системы строить модель, существует много. Отсутствие в них возможностей поддерживать тот или иной этап разработки принимается как данное. Разработчики используют конкретную среду моделирования и дополнительно устанавливают свои правила совместной работы отдельных коллективов. В результате получается своеобразная, во многом неформальная, надстройка над средой моделирования.

В то же время почти для всех существующих сред моделирования характерны следующие недостатки:

- изолированность, не позволяющая использовать компоненты и библиотеки различных сред в одном проекте, приводящая к многократному дублированию одних и тех же моделей на различных языках моделирования. И дело здесь не столько в конкуренции и борьбе за потребителя, в результате чего нужные компоненты оказываются недоступными, сколько в трудности правильно воспроизвести поведение "чужого" компонента в

"своей" среде. Проблема "обмена" моделями между средами до сих пор еще не нашла удовлетворительного решения;

- отсутствие полного набора средств исследования модели и, как следствие, необходимость строить, исследовать и визуализировать поведение отдельных компонентов или всей модели в других средах, где нужные инструменты анализа существуют, или дублировать их.

В этих условиях ценной является любая возможность унификации, поэтому представляется, что в качестве основы для современного объектно-ориентированного языка моделирования сложных динамических систем должен использоваться унифицированный язык моделирования. Этим языком может быть UML — объектно-ориентированный язык прототипирования сложных программных комплексов и вычислительных систем [2]. Язык UML заслуживает внимания по нескольким причинам.

Во-первых, авторы UML являются ведущими специалистами и язык зафиксировал все основные достижения объектно-ориентированного подхода к проектированию программных систем, став фактическим стандартом. Поэтому создателям специальных языков ООМ имеет смысл использовать UML как своего рода метаязык.

Во-вторых, часть конструкций UML может быть использована непосредственно для описания структуры проекта, а часть — в качестве основы для расширения.

По мере введения различных понятий и конструкций UML, будет показано, как можно использовать их непосредственно в языках моделирования сложных динамических систем.

В то же время авторы UML считают, что "UML — язык дискретного моделирования, и он не предназначен для разработки непрерывных систем, встречающихся в физике и механике". Это справедливо, если под непрерывной физической системой понимать традиционную математическую модель в виде изолированной системы, описываемой обыкновенными дифференциальными уравнениями или уравнениями в частных производных, не меняющей своей формы во времени. В случае же многокомпонентных систем, меняющихся под воздействием дискретных событий, многие конструкции языка UML могут быть использованы для описания моделей таких систем.

# Классы, экземпляры и многокомпонентные системы

> "При работе с задачами и системами, состоящими из большого количества элементов, возможность разложения на составные части приобретает первостепенное значение. Внимание человека должно сосредотачиваться; чтобы мыслить точно и связно, необходимо по возможности сокращать количество одновременно охватываемых понятий. Разлагая большую задачу на части, можно получить задачи-компоненты с таким небольшим количеством деталей, чтобы каждую из этих задач можно было обдумывать и анализировать в целом. Разложение на составные части абсолютно необходимо, если в анализе и программировании задачи принимает участие несколько человек".
>
> *У.-И. Дал, Б. Мюрхауг, К. Июгорд [6]*

Понятие *класса* как множества сущностей (объектов), имеющих одинаковое функциональное назначение, структуру и поведение и отличающихся значением параметров, является ключевым для проектирования многокомпонентных систем. С понятием класса неразрывно связано понятие *экземпляра класса*, т. е. конкретного объекта из множества всех объектов того же самого класса с уникальными значениями параметров.

Разбиение реальной или проектируемой системы на компоненты, построение классов, соответствующих этим компонентам, установление связей между компонентами, аналогичных связям между компонентами моделируемой реальной или проектируемой системы, и построение модели из экземпляров классов с учетом существующих связей — это ключевые моменты технологии объектно-ориентированного моделирования. При объектно-ориентированном подходе структура модели соответствует структуре реального объекта.

Использование *экземпляров класса* позволяет собирать новую модель из типовых компонентов, подстраивая их каждый раз под конкретные условия функционирования, а также использовать построенную модель в качестве компонента других моделей (рис. 1.1). Так создаются модели и новые библиотечные компоненты в пакете Simulink. Поэтому любой инструмент моделирования, позволяющий строить модели из стандартных компонентов, явно или неявно использует технологию ООМ. В то же время инструменты, использующие все возможности ООМ, позволяют пользователю не только самому определять новые элементарные компоненты (создавать новые классы, не являющиеся механической композицией уже существующих), но и применять наследование и полиморфизм для модификации уже разработанных классов.

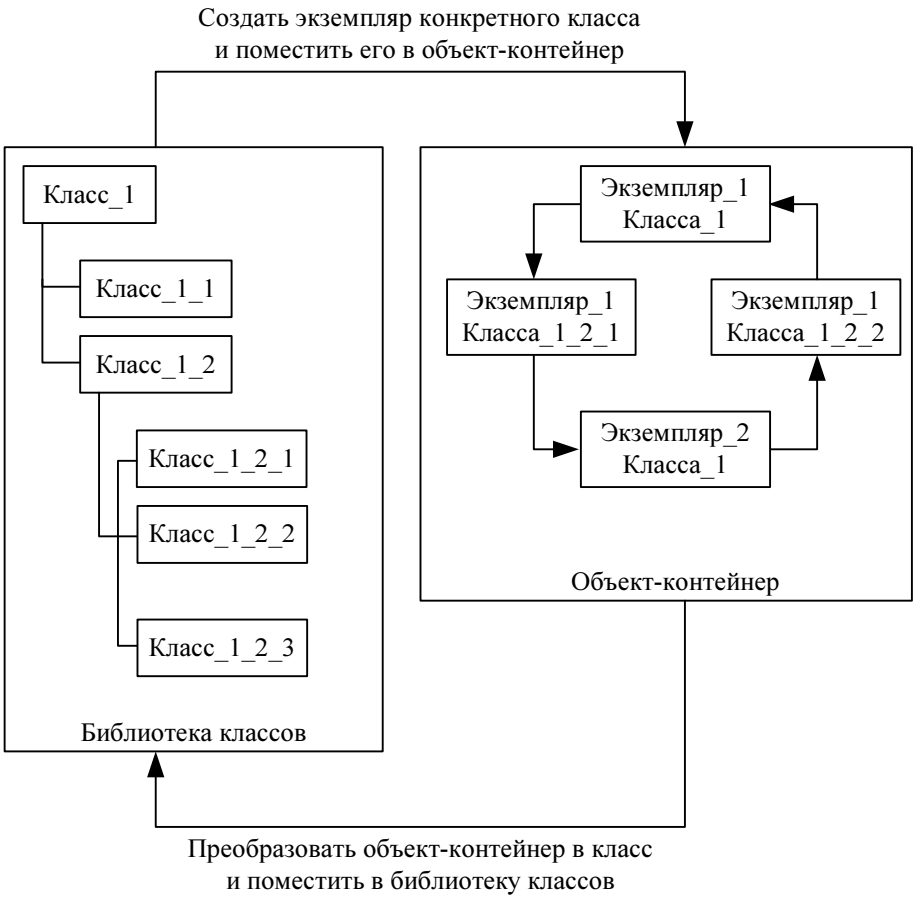



**Рис. 1.1.** Создание нового компонента с помощью готовых компонентов

В UML выделяются "активные" и "пассивные" объекты и соответственно "активные" и "пассивные" классы, экземплярами которых они являются. *Активным объектом*, согласно UML, называется "объект, который владеет потоком управления и может инициировать управляющие воздействия" [2]. *Пассивным объектом* называется "объект, у которого нет собственного потока управления. Все его операции выполняются под управлением потока, прикрепленного к активному объекту" [2]. Применительно к дискретным системам эти определения вполне достаточны, однако применительно к ООМ сложных динамических систем они представляются несколько узкими. Поэтому заменим "поток управления" на "процесс, развивающийся во времени" и "управляющие воздействия" — на "воздействия на другие объекты". В *гла-*

*ве 2* для компонентов сложных динамических систем (СДС) вводится специальное понятие "активного динамического объекта".

Компоненты модели СДС могут быть как активными, так и пассивными объектами. Пассивным объектом удобно считать совокупность данных и чисто алгоритмических операций (например, объект "список" с набором операций добавления, удаления и модификации).

Объектно-ориентированный подход позволяет скрывать детали реализации компонента и делать "видимыми" только те переменные и операции, которые нужны другим компонентам (инкапсуляция данных). В активных объектах данные и поведение — законы, определяющие изменение переменных состояния во времени, скрыты от наблюдателя, видны обычно только отдельные переменные состояния, значения которых изменяются во времени, и операции, управляющие поведением. В ОПП мы последовательно вызываем операции (методы) других программных объектов, для того чтобы изменить данные этих объектов или получить нужный результат, необходимый нам как промежуточный в процессе "своих" вычислений, а в ОММ мы преимущественно используем результаты параллельных действий, проводимых другими объектами, или управляем их поведением с помощью сигналов.

Объектно-ориентированный подход позволяет использовать ранее созданные классы для проектирования новых классов, дополняя и видоизменяя отдельные свойства уже разработанных классов. В результате возникает иерархия классов, в которой новые классы автоматически включают в себя данные и операции своих предшественников и отражают в своем описании в явном виде только новую информацию. Отношение между классами в такой иерархии называется *обобщением*. Новые классы обобщают свойства уже существующих классов, ранее созданных. Механизм, с помощью которого создаются новые классы, включающие в себя свойства предшественников, называется *наследованием*.

Под обобщением также понимают и правила использования экземпляров одних классов (созданных позже) вместо экземпляров других классов (созданных ранее), если новые классы наследуют свойства старых, дополняя их. Действительно, в этом случае все новые классы включают без изменений все свойства своих предшественников и умеют делать все то, что умели делать их предки. Таким образом, экземпляры новых классов всегда можно использовать вместо экземпляров их предшественников.

Обобщение может быть связано и с переопределением свойств предшественников. В случае переопределения операций классов появляется возможность получить так называемые *полиморфные операции*. Упрощенно говоря, каждый новый класс в иерархии видоизменяет некоторое общее для всех классов

действие (как минимум, общим для этих действий является имя), реализуя его по-своему. И вертолет, и самолет с вертикальной посадкой умеют летать и садиться в произвольном месте, но реализуют это разными способами. И тот, и другой можно определить как потомок класса "летательный аппарат" с операциями "взлет", "полет", "посадка". И если иерархия выстроена так:

"летательный аппарат" ← вертолет ← самолет с вертикальной посадкой,

то конкретный самолет всегда можно использовать вместо конкретного вертолета. С помощью полиморфных операций можно передать специфические черты общих для родственных классов операций, приписав каждому классу в иерархии собственную модификацию общей для всех операции. При этом мы можем не уточнять, что "взлет", "полет", "посадка" осуществляется вертолетом или самолетом — это и так становится ясным из контекста, где указывается вид используемого летательного аппарата.

Экземпляры классов можно создавать в начале вычислительного эксперимента и использовать их на протяжении всего процесса моделирования, а можно создавать, когда в них возникает потребность, и уничтожать, как только они становятся ненужными. Возможность динамически создавать и уничтожать экземпляры классов позволяет создавать *модели с переменным числом компонентов*.

Модели с переменным во времени числом компонентов возникают во многих приложениях, и в частности широко используются при моделировании систем массового обслуживания.

ООМ целесообразно использовать, когда:

- среди компонентов модели много однотипных и характерных для данной прикладной области, отличающихся только значениями параметров;
- имеется возможность так разбить моделируемый объект на компоненты, что классы, им соответствующие, образуют библиотеки, имеющие древовидную структуру;
- система имеет переменную структуру, в которой число компонентов неизвестно заранее, и они могут появляться и исчезать в зависимости от наступления различных событий;
- библиотеки классов, создаваемые для одного проекта, могут быть многократно использованы непосредственно или после модификации в других проектах.

# Использование UML на начальной стадии проектирования

> "UML должен быть достаточно выразительным, чтобы работать со всеми концепциями, возникающими в современной системе, такими, например, как параллелизм и распределенность, а также с такими механизмами создания программного обеспечения, как инкапсуляция и компоненты".
>
> "UML запутан, неточен и сложен".
>
> *Г. Буч, А. Якобсон, Дж. Рамбо [2]*

UML предоставляет пользователю большой набор средств графического описания структуры и поведения сложных систем. Выбор конкретного способа представления модели с помощью графических элементов языка определяется пользователем.

Предполагается, что модель проектируемой системы создается сначала в самом общем виде и постепенно уточняется. На первом этапе обсуждаются список возможных вариантов использования будущей системы, статическая структура модели (классы и их отношения), распределяются виды работ между участниками разработки. На втором этапе детализируется структура классов, уточняется, из каких компонентов они состоят и как эти компоненты связаны между собой, конкретизируется динамическое поведение системы. На третьем этапе выбирается конкретная конфигурация вычислительной системы, на которой будет реализована модель, и выделяются ресурсы отдельным компонентам. Нас в дальнейшем будут интересовать только два первых этапа.

Для описания моделей сложных динамических объектов, в отличие от моделей сложных программных комплексов, достаточно небольшого подмножества из всего арсенала средств, предлагаемых UML, и уточнения некоторых понятий, касающихся поведенческих аспектов моделируемой системы.

# Диаграммы классов

*Класс* — это языковая конструкция, служащая для описания множества объектов, обладающих общими свойствами. Такие конструкции языка создатели UML называют классификаторами, т. к. они наделяют создаваемый элемент модели свойствами, по которым его можно отличить от других элементов.

С классом в UML связаны понятия *атрибута, операции, метода и поведения* — общих свойств, присущих моделируемому объекту. Связи между классами и свойства классов отображаются на диаграммах классов.

## Атрибуты

Под *атрибутом* понимается описание поименованного элемента заданного типа, для которого указана область значений. Какие именно типы можно использовать, определяется конкретным языком моделирования. При создании экземпляра класса его значение из указанного диапазона должно быть конкретизировано. Исключение составляют атрибуты, общие для всего класса в целом, их значения остаются постоянными для любого экземпляра класса и присваиваются им при инициализации самого класса (иногда их называют статическими или атрибутами класса). Описание атрибутов подчиняется определенным синтаксическим правилам и размещается в секции описания атрибутов. В частности, при описании статических атрибутов соответствующая строка в секции атрибутов подчеркивается.

Обязательным элементом описания атрибута является только его имя. Имя должно быть уникальным внутри класса и может иметь различные области видимости: + (публичная), # (защищенная), - (приватная), ~ (пакетная) видимость — приведенные символы используются как элементы языка описания атрибутов. В качестве значения видимости по умолчанию выбирается приватная видимость, делающая переменные класса невидимыми извне. В общем случае имя атрибута представляется составным именем, указывающим класс, в котором он объявляется, или "путь" к этому атрибуту в общем "пространстве" имен модели, что учитывает вложенность описаний. Любой атрибут может иметь конкретное предопределенное значение, которое он получит в момент создания экземпляра класса.

## Поведение

*Поведение* отражает динамические свойства классов и подразделяется на *взаимодействие*, *смену состояний* и *деятельность*. И соответственно, поведение может быть представлено тремя способами: в виде диаграмм последовательности и коммуникации, диаграмм состояний и диаграмм деятельности.

*Взаимодействие*, представляемое диаграммами последовательности и коммуникации, — это обмен сообщений между компонентами модели. Диаграммы последовательности и коммуникации описывают участников взаимодействия (источники передачи и получателей сообщений), последовательности обмена сообщениями между ними, линии жизни участников (экземпляры классов могут создаваться и уничтожаться в любой момент модельного времени), а также накладывают временные ограничения на время передачи и приема сообщений.

Назначение диаграмм последовательности и коммуникации — указать участников и возможные сценарии обмена сообщениями между ними. Достаточно

очевидно, что такие диаграммы нужны разработчику на самом раннем этапе проектирования функциональной схемы, а в дальнейшем могут служить только для проверки правильности разработанных алгоритмов посылки и обработки полученных сообщений.

*Смена состояний* является важным признаком проявления динамики. Под состоянием в UML понимаются "условия или ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет собственную деятельность или ожидает какого-либо события [2]".

Напомним, что при моделировании сложных динамических систем лучше употреблять термин "гиперсостояния", т. к. речь может идти об объединении множества состояний, удовлетворяющих некоторому условию.

О гиперсостояниях идет речь в картах состояний Харела. В гибридных системах гиперсостояния системы ассоциируются с длительными процессами, происходящими в моделируемых объектах. Например, мы говорим, что система отопления находится в нормальном состоянии (явное гиперсостояние, т. к. речь идет о длительной работе системы управления), если в комнате поддерживается заданная температура. Если система отопления не работает и ожидает сигнала, чтобы начать выполнение предписанных заданий: начать или прекратить автоматическую поддержку температуры, перейти к аварийному режиму поддержания температуры — это тоже состояние системы, состояние ожидания. Наконец, система отопления может выполнять предписанные длительные действия (проверять оборудование перед началом отопления), только после завершения которых она может перейти в состояние ожидания. В гибридных системах гиперсостояниям обычно предписываются уравнения, параметры и форма которых неизменны.

*Деятельность* в UML трактуется как управление всей системой. Управление в динамической системе может осуществляться непрерывно (например, управление непрерывной системой по положению и скорости с помощью ПИД-регулятора) и дискретно (с помощью встроенных цифровых вычислителей в контуре управления непрерывным устройством, предполагающее квантование непрерывного входного и выходного управляющих сигналов). С точки зрения современных сложных динамических систем, это достаточно простые виды управления. На управление можно смотреть и шире. В этом смысле деятельность связана с алгоритмами управления, или с анализирующими и поддерживающими характеристики системы в заданных границах, или с оптимизирующими ее поведение в соответствии с заданными критериями, или приводящими систему к выбранной цели. В последнем случае управление выражается в предписании, какие действия необходимо выполнить для достижения заданной цели. Как описать процессы управления, являются ли эти процессы последовательными или параллельными, в какой по-

следовательности их следует выполнять. Именно это и указывается на диаграммах деятельности.

Непосредственно диаграммы деятельности не нашли применения в современных языках моделирования — исключение составляют языки моделирования, использующие сети Петри.

И деятельность, как разновидность глобального поведения всей системы, и смена состояний могут быть представлены одним и тем же графическим способом — диаграммой состояний. Такая диаграмма по существу является "дискретной", т. е. классической картой состояния, отражающей, что и в какой последовательности следует выполнять для реализации управления, с указанием того, какие процессы можно выполнять параллельно. Язык UML допускает использование диаграмм состояний для описания деятельности, и мы воспользуемся этим, чтобы не описывать подробно собственно диаграмму деятельности.

## Операции и методы

Помимо секции атрибутов, описание любого класса содержит секцию операций. Авторы UML отождествляют операцию с любой "вычислительной услугой", которую один экземпляр класса предоставляет другим экземплярам, или использует сам в процессе работы. Эта услуга, так же как атрибуты класса, может иметь различную видимость. Имя операции является единственным синтаксическим элементом описания операции, хотя в описании может использоваться и список входных и выходных параметров, с указанием их типов.

Описание операций класса помещается в секцию операций класса. Конкретная реализация операции называется *методом*. При описании операций им можно и не сопоставлять метод, тогда они называются *абстрактными*. Имена абстрактных операций выделяются курсивом.

Обычно операции осуществляются над конкретными данными, приписанными экземпляру, однако операции могут использоваться для преобразования данных, приписанных всему классу. К операциям, свойственным классу в целом, можно отнести операции над статическими данными класса, выполняемыми в момент инициализации класса.

## Абстрактные и конкретные классы. Интерфейсы

Класс изображается прямоугольником, имеющим три секции — секцию имени, секцию атрибутов и секцию операций (рис. 1.2, *а*, *б*). Обязательной среди этих секций является секция имени.

С помощью классов могут порождаться новые сущности — экземпляры классов, в которых конкретизируются значения их атрибутов (прямые экземпляры). Такие классы называются *конкретными* (рис. 1.2, *б*). Классы бывают и *абстрактными* (рис. 1.2, *а*). Их используют для описания общей структуры всех его потомков — конкретных классов. Абстрактные классы не могут иметь прямых экземпляров. Прямые экземпляры потомков абстрактного класса называются *косвенными экземплярами абстрактного класса.*

На раннем этапе проектирования иногда достаточно знать, какие операции класса будут видны извне (рис. 1.2, *в*), иными словами, какими операциями данного класса могут воспользоваться другие классы. Для описания таких операций в UML предусмотрены интерфейсы. *Интерфейс* — это описатель видимых снаружи операций класса, или по-другому описатель услуг, предоставляемых классом. Один и тот же класс может иметь несколько интерфейсов, описывающих различные услуги. Интерфейсы бывают обеспечиваемыми и требуемыми. *Обеспечиваемым* называется интерфейс, который предоставляет класс; *требуемый интерфейс* — это тот набор операций, который необходим классу от других классов для выполнения своих задач. Интерфейс можно рассматривать как абстрактный класс, содержащий только абстрактные операции. Интерфейс, как и класс, является классификатором. Отношение обобщения применимо к интерфейсу, таким образом строя интерфейсы, можно применять механизм наследования.

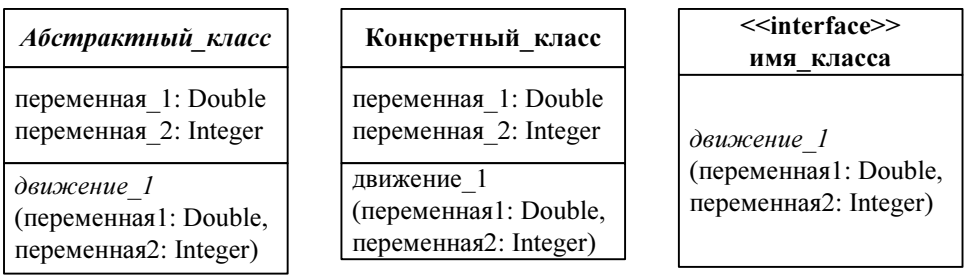



**Рис. 1.2.** Языковые конструкции: *а* — абстрактный класс; *б* — конкретный класс; *в* — интерфейс

Как уже отмечалось, классы могут быть как активными, так и пассивными. *Активные классы* изображаются особым образом (рис. 1.3). В этой главе не столь существенно, является ли класс пассивным или активным, поэтому мы не будем усложнять рисунки, подчеркивая графически активность или пассивность рассматриваемого класса.

На рис. 1.4 показаны последовательные этапы разработки: интерфейса, абстрактного и конкретных классов для построения некоторой механической


**Рис. 1.3.** Изображение активного класса

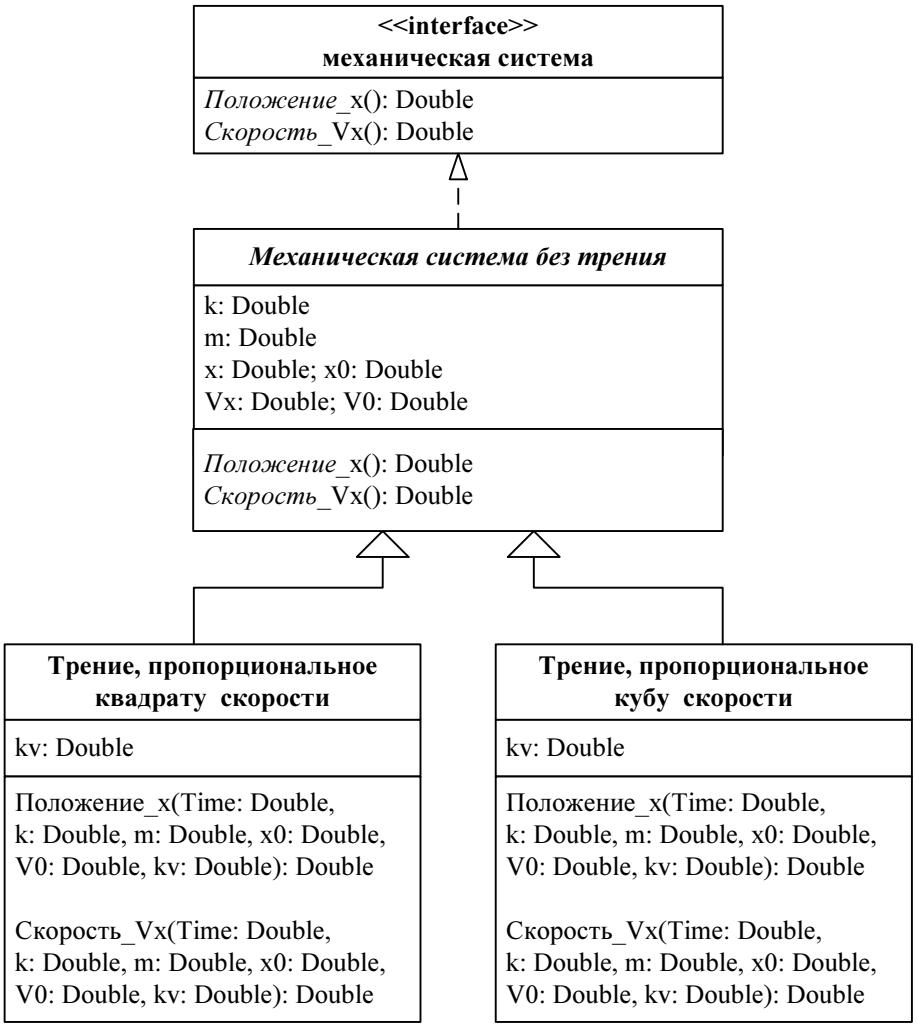



**Рис. 1.4.** Построение модели с помощью интерфейса и различного вида классов

модели, у которой законы движения различным образом учитывают трение. Сам интерфейс указывает только на то, что конкретный класс может предоставить законы, по которым вычисляются положение и скорость объекта. У класса "механическая система" имеются две абстрактные операции. Классы "трение, пропорциональное квадрату скорости" и "трение, пропорциональное кубу скорости" наследуют свойства класса "механическая система без трения" и по-разному реализуют требуемые операции. Таким образом, оба класса предоставляют требуемый интерфейс, например, для класса, который в дальнейшем будет строить графики зависимости положения и скорости от времени.

На рис. 1.4 использована стрелка, нарисованная пунктиром. Она указывает на то, что оба конкретных класса реализуют один и тот же интерфейс. Можно сказать, что конструирование абстрактных и конкретных классов зависит от предложенного интерфейса. Пунктирная стрелка на рис. 1.4, восходящая к интерфейсу, как раз и указывает на отношение зависимости между интерфейсом и классом.

# Классы и отношения

Классификаторы могут находиться между собой в различных отношениях— ассоциации, зависимости, обобщения, реализации, использования. Основными отношениями для классов можно считать отношения ассоциации, обобщения, агрегации и композиции. Интерфейсы могут находиться в отношении обобщения. Таким образом, для создания и классов, и интерфейсов можно использовать механизм наследования.

## Ассоциация

*Ассоциация классов* — это отношение, указывающее, что между экземплярами классов существует логическая связь. Ассоциации являются важным элементом диаграмм классов. У ассоциации может быть имя, помогающее понять, что подразумевается под логической связью. Наиболее простой вид ассоциации — бинарная ассоциация, или упорядоченная пара (Имя класса_1, Имя класса_2). Например, ассоциация (Станция слежения, Объект слежения) с именем `следит` указывает на ведущую роль и назначение станции слежения. В данном случае, помещенное на первое место имя класса "Станция слежения" указывает, что при рассмотрении этой пары главную роль в конкретной модели играет именно этот класс. Бинарная ассоциация изображается сплошной линией (рис. 1.5), соединяющей два изображения классов. У концов этой линии (полюсов ассоциации) можно указать множественность экземпляров, участвующих в ассоциации. В нашем примере участвует один экземпляр

"Станции слежения", но с ним может быть связано произвольное число экземпляров класса "Объект слежения". На возможное отсутствие объектов слежения указывает значение ноль для нижней границы числа экземпляров, за которыми может следить станция слежения, и произвольное число — для верхней границы. Связь между экземплярами не обязательно должна быть постоянной — она может возникать и исчезать в процессе работы модели.

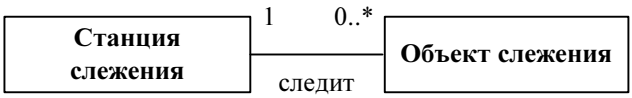



**Рис. 1.5.** Ассоциация с указанием множественности

С помощью различных видов ассоциации можно указать на различные виды связей между экземплярами: например, если элемент связан сразу с двумя экземплярами, то можно уточнить, может ли он быть связан одновременно с одним экземпляром, или только с одним из них.

Построение ассоциаций между классами чрезвычайно важно на первоначальном этапе построения будущей модели и может найти отражение в комментариях в таких реально используемых графических элементах языков моделирования, как функциональные схемы.

## Обобщение

> "Миф о том, что наследование — это всегда хорошо, является проклятием объектно-ориентированных технологий с момента их зарождения".
>
> *Г. Буч, А. Якобсон, Дж. Рамбо [2]*

Отношение обобщения, как мы уже видели, является отношением, с помощью которого строится, например, дерево классов. В корень дерева помещается предок, или *суперкласс*, а далее следуют его потомки — *подклассы*. Напомним, что если класс в некоторой иерархии является абстрактным, то его имя записывается курсивом (рис. 1.6, *а*, *б* показывают два возможных способа представления дерева классов, связанных отношением обобщения). Дерево классов может использоваться и на этапе предварительного рассмотрения модели, и как графический элемент языка моделирования. Вспомните, как подобные графические конструкции используются для описания иерархически организованных файлов.

Отношение обобщения может быть применено и к другим компонентам, имеющим иерархическую структуру, к уже рассматривавшимся интерфейсам.

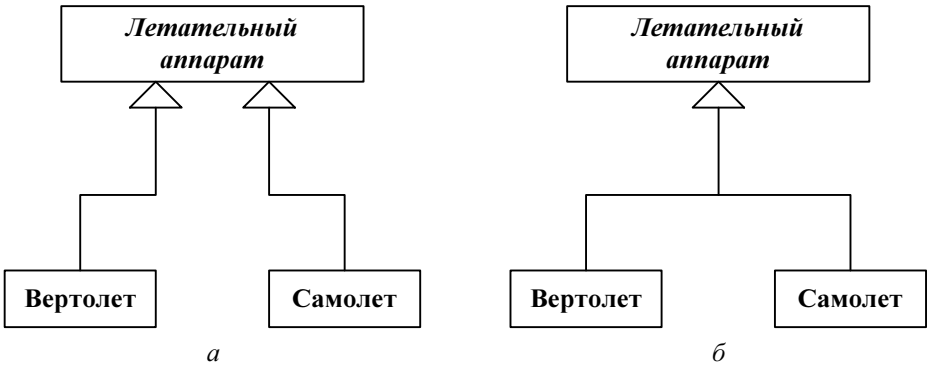



**Рис. 1.6.** Обобщение. Два допустимых способа представления дерева классов: *а* — соединительные линии независимы; *б* — соединительные линии объединены

На диаграмме классов можно дополнительно указать, что данное дерево классов полностью разработано и не будет в дальнейшем расширяться — {complete}, или требует доработки {incomplete}.

## Агрегация

Отношение агрегации между классами является частным случаем отношения ассоциации (рис. 1.7), и указывает на то, что один класс в качестве составляющих может содержать другие классы. Это отношение "часть — целое", и оно используется для того, чтобы показать, что класс имеет внутренние компоненты и, возможно, иерархическую структуру.

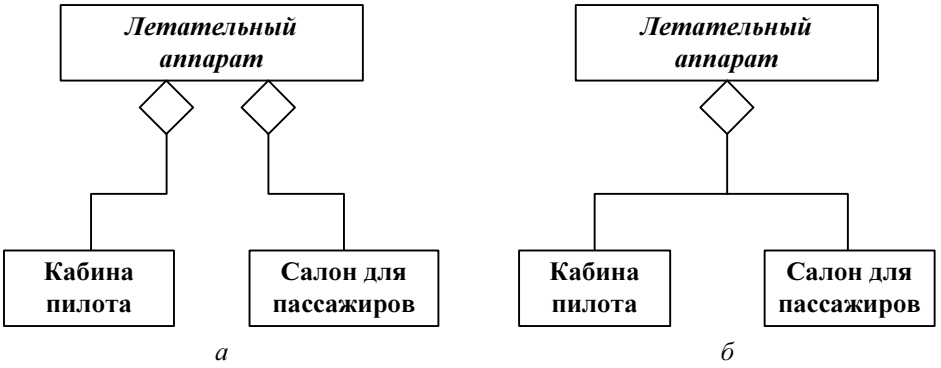



**Рис. 1.7.** Агрегация. Два допустимых способа представления структуры класса: *а* — соединительные линии независимы; *б* — соединительные линии объединены

Частным случаем агрегации является отношение композиции (рис. 1.8), указывающее, что класс не только имеет структуру, но и представляет собой

единое целое. Создавая такой класс, мы создаем все его компоненты одновременно, уничтожая — уничтожаем все его составляющие.

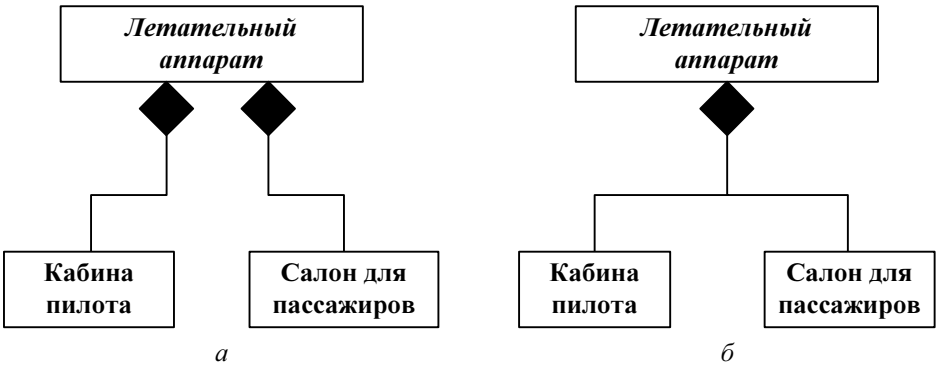



**Рис. 1.8.** Композиция. Два допустимых способа представления композиции: *а* — соединительные линии независимы; *б* — соединительные линии объединены

## Наследование

Отношение обобщения непосредственно связано с различными способами построения новых классов на базе существующих.

*Наследование* — это механизм, позволяющий на базе имеющихся классов строить новые, используя уже готовые классы, как часть новых. Новые классы обобщают существующие, дополняя их новыми свойствами. Строить новые классы можно различными путями.

- **Построение новых классов с помощью композиции.**

  Весьма распространенным является прием, когда пользователю доступен только экземпляр класса-контейнера, куда можно помещать экземпляры существующих классов, настраивать их параметры и связывать между собой (см. рис. 1.1). При этом существует возможность сохранить созданный компонент в виде элемента пользовательской библиотеки (нового класса). Помещая экземпляр вновь созданного класса в новый контейнер и объединяя его с экземплярами других классов, можно построить новый компонент и иерархию классов. Этот подход чрезвычайно удобен, если заранее известно, что новое устройство можно собрать, используя только существующие компоненты.

- **Наследование путем добавления новых свойств.**

  Можно строить новые классы, наделяя существующие классы новыми свойствами (добавляя переменные, структурные компоненты и операции,

т. е. все те элементы, что составляют класс в конкретном языке программирования или моделирования) и сохраняя все уже имеющиеся. Предшественник нового класса просто встраивается в него как готовый элемент, без каких-либо изменений (рис. 1.9).

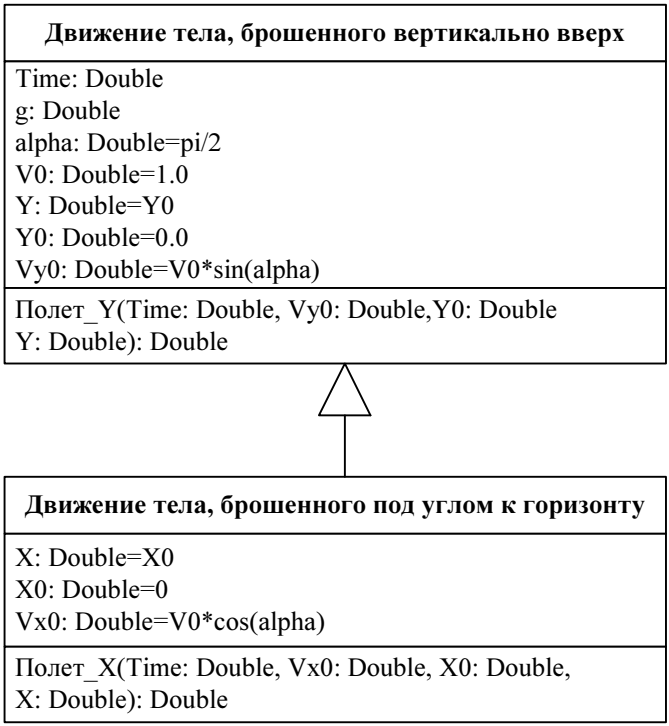
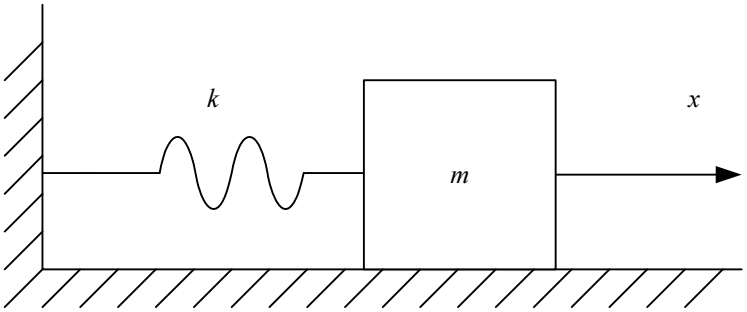



**Рис. 1.9.** Наследование добавлением новых свойств

На рис. 1.9 показаны два класса: "Движение тела, брошенного вертикально вверх" и "Движение тела, брошенного под углом к горизонту". Второй класс наследует операцию вычисления высоты тела и все переменные состояния первого класса и добавляет только операцию вычисления дальности полета как функции от времени и начальных условий. Столь же просто добавить в эту иерархию и еще более сложную модель. Сейчас выбранная нами система координат неподвижна. Свяжем эту систему координат с движущимся прямолинейно и с постоянной скоростью новым объектом, например, поместим наш шарик в самолет. Используя модель "Движение тела, брошенного под углом к горизонту" как базовую, не представляет труда описать полет шарика в самолете в системе координат, связанной с поверхностью Земли.

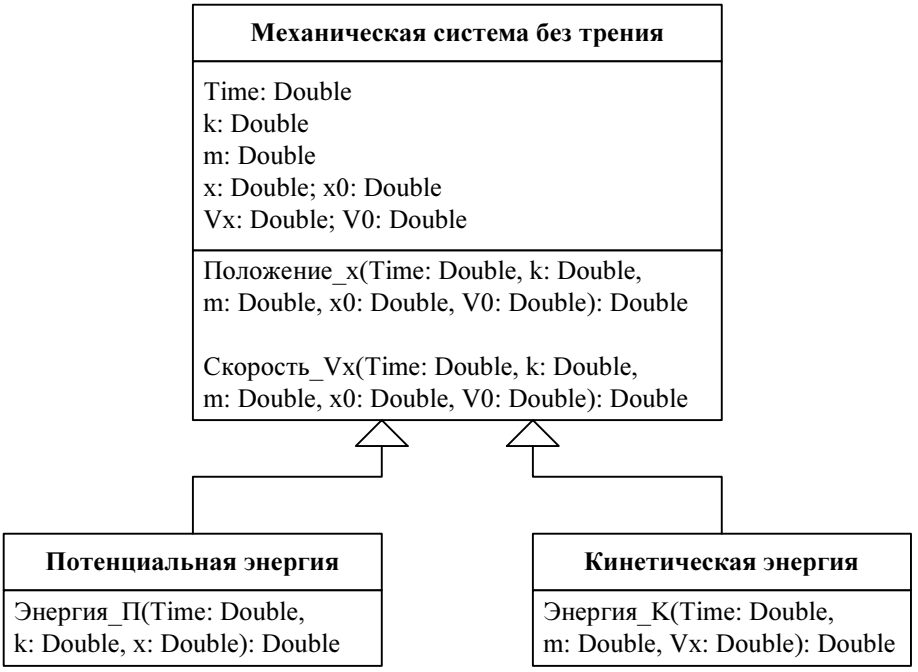



*а*

$$m\frac{d^2x}{dt^2} + k \cdot x = 0$$

$$T = \frac{1}{2} \cdot \left(\frac{dx}{dt}\right)^2$$

$$V = \int_0^x k \cdot x \cdot dx = \frac{1}{2} \cdot k \cdot x^2$$



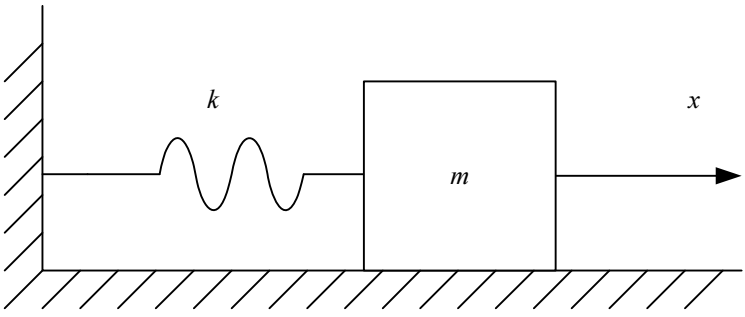

*б*

**Рис. 1.10.** Добавление новых свойств в исходный класс: *а* — механическая система; *б* — представление математической модели в виде дерева классов

Если все классы от предка до последнего потомка оказываются вложенными друг в друга, тогда они образуют одну-единственную ветвь дерева (рис. 1.9). Если новые классы, порожденные от одного и того же предшественника, отличаются между собой различными элементами, каждому из них добавлено свое, новое уникальное свойство, то возникает дерево классов. На рис. 1.10 показана простейшая механическая система, в которой класс-предок вычисляет только положение и скорость груза, движущегося за счет сжатия пружины, а его потомки могут вычислить дополнительно кинетическую и потенциальную энергию груза.

В ООМ дерево классов можно строить так, чтобы возник набор последовательно усложняющихся моделей одного и того же реального объекта. Надо признаться, что наборы последовательно усложняющихся моделей встречаются реже, чем наборы, в которых используют уже существующий класс просто как заготовку для нового класса. В последнем случае свойства предков и потомков могут существенно различаться. Но даже этот последний прием используется весьма ограниченно — деревья чаще оказываются "кустами" с небольшим числом уровней.

Рассмотренный вид наследования называется *обычным*, или *одиночным*, *наследованием*. В то же время можно представить себе ситуацию, когда потомок может наследовать свойства нескольких классов, принадлежащих




*а*

$$m\frac{d^2x}{dt^2} + k \cdot x = 0$$

$$m\frac{d^2x}{dt^2} + k \cdot x + kv \cdot \left(\frac{dx}{dt}\right)^2 = 0$$

$$m\frac{d^2x}{dt^2} + k \cdot x + kv \cdot \left(\frac{dx}{dt}\right)^3 = 0$$

**Рис. 1.11.** *(Часть 1 из 2.)* Переопределение операций исходного класса:
*а* — механическая система

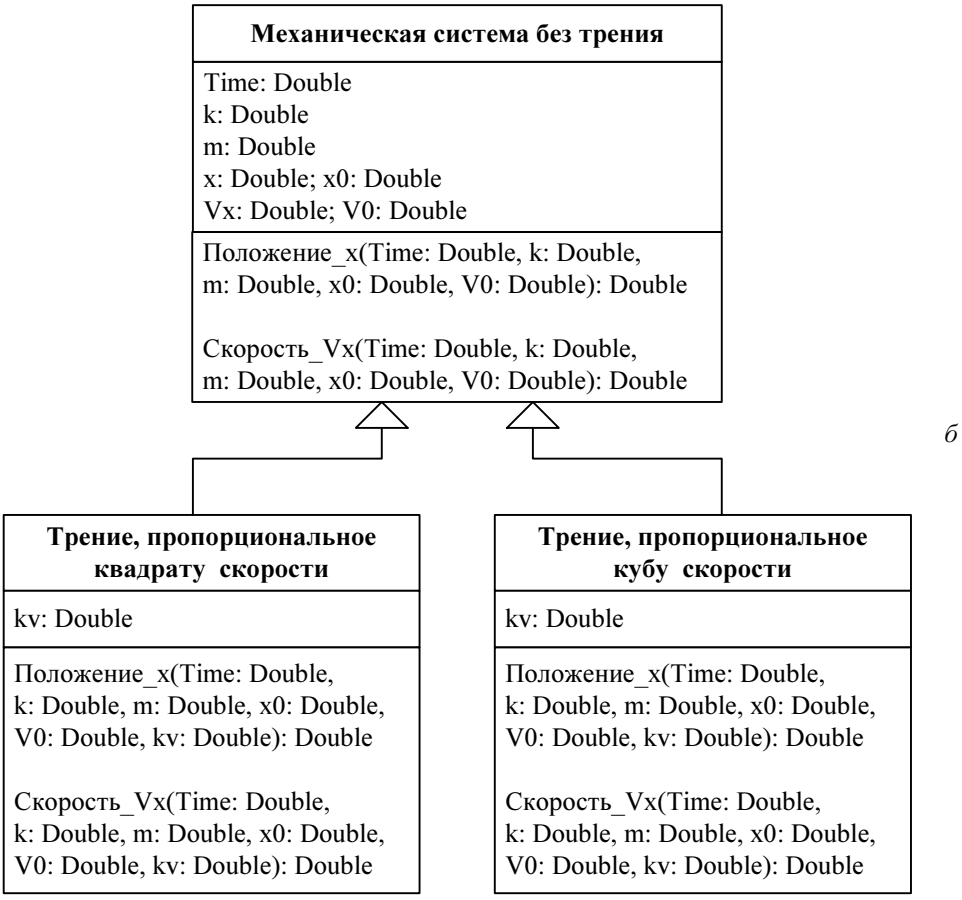



**Рис. 1.11.** *(Часть 2 из 2.)* Переопределение операций исходного класса: *б* — дерево классов-моделей

разным деревьям, и тогда наследование называется *множественным*. Множественное наследование — потенциально опасный механизм создания новых сущностей, т. к. у классов-родителей могут оказаться одинаковые по сигнатуре, но разные по семантике функции.

- **Наследование путем переопределения существующих свойств.**

  Во многих случаях потомки отличаются от предков тем, что они изменяют (переопределяют) свойства своих родителей. Обычно речь идет о переопределении операций или структуры. Например, в нашем примере простой механической системы предком может быть класс, моделирующий колебания груза в среде без трения, а его потомки могут учитывать различные виды трения (рис. 1.11).

## Полиморфизм

Как уже отмечалось, *полиморфизм* связан с правилом использования (подстановки) одних классов вместо других, если классы связаны отношением обобщения. В дереве классов все потомки умеют делать все, что умеют делать предки, и в этом смысле они взаимозаменяемы. В случае если в корне дерева лежит абстрактный класс, имеющий несколько абстрактных операций, то его конкретные потомки должны заменить абстрактные операции конкретными (имеющими методы), а их потомки могут переопределить конкретные операции предков собственными. Это позволяет не вникать в детали реализации уже существующей библиотеки и дополнить ее новыми классами с нужными операциями. Такое переопределение также позволяет переложить на исполняющую систему задачу выбора нужного метода, когда осуществляется подстановка.

# Поведение. Диаграммы состояний

В UML диаграмма состояний используется для описания динамических свойств системы и показывает, в каких состояниях может находиться система, при каких условиях эти состояния сменяют друг друга, по каким законам меняются переменные состояния в конкретных состояниях. Диаграммы состояний также показывают, как протекают параллельные процессы в моделируемом объекте, если они существуют.

На рис. 1.12 представлена диаграмма состояний, описывающая финальный забег двух спортсменов, которым разрешены технические остановки. "Финальный забег" — это имя состояния. В забеге участвуют два спортсмена, бегущие независимо друг от друга, т. е. мы наблюдаем два параллельных процесса. Это отражено на карте состояний — в основное состояние вложены два внутренних состояния "Спортсмен_1" и "Спортсмен_2". Параллельные состояния разделены пунктирной линией. Внутри каждого параллельного состояния спортсмен совершает последовательные действия, меняя состояние "Движение" на состояние "Остановка". У состояния "Финальный забег" могут быть свои входные и выходные действия (например, entry/action1, action2 и exit/action3, action4), точно так же, как входные и выходные действия могут быть у каждого вложенного состояния (entry/action11, action12 и exit/action21, action22). Финальный забег заканчивается, только когда оба спортсмена достигают финиша.

На диаграмме состояний прямоугольники со скругленными углами обозначают состояния. Каждое состояние имеет имя. Стрелки обозначают переходы, над которыми можно указывать имена событий, приводящих к смене состояния. Такие переходы называются *триггерными*. *Нетриггерные* переходы

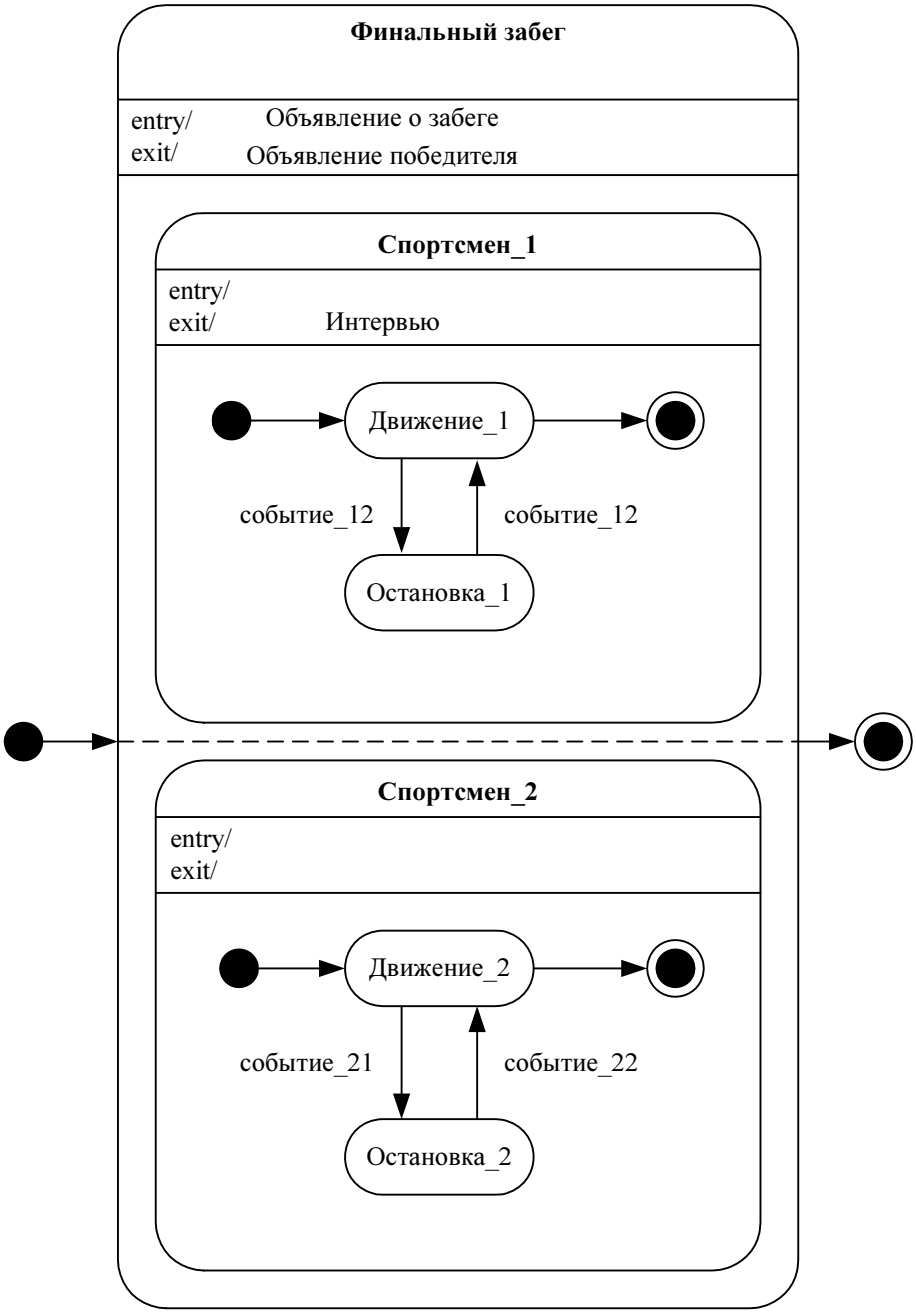



**Рис. 1.12.** Диаграмма состояний с параллельными процессами

связаны с окончанием деятельности в состоянии. В нашем случае — это переход в финальное состояние, означающее конец забега, или переходы, указывающие на достижение финиша каждым спортсменом. Описание события может сопровождаться сторожевым условием. Если событие произошло, то сторожевое условие, в зависимости его истинности или ложности, разрешает либо запрещает переход в новое состояние. И наконец, с переходом связаны еще и имена операций, которые следует выполнить, когда происходит смена состояния.

С переходами связаны не только условия покидания состояния, но и действия перехода. Как входные и выходные действия, так и действия перехода считаются мгновенными.

От диаграмм состояний UML требует следующее.

- Диаграмма состояний обязана обрабатывать событие за событием, последовательно, и перед обработкой следующего завершать все, что касается предыдущего. Таким образом, события не взаимодействуют друг с другом. В этом смысле события асинхронны и не должны происходить одновременно. Если же события оказываются одновременными, то требуется, чтобы их можно обрабатывать в любом порядке, и результат обработки в любом случае будет одинаковым. Параллельное вычисление в распределенной системе требует независимости.
- Диаграмма состояний должна быть детерминирована. Если у активного (текущего) состояния могут сработать одновременно несколько исходящих переходов, сработает только один, и какой именно — предсказать невозможно, т. к. это зависит от программной реализации.
- Во время обработки очередного события могут быть активны несколько параллельных состояний. Диаграмма состояний должна передать копию такого события всем активным состояниям.
- Параллельные состояния не могут взаимодействовать посредством общей разделяемой памяти. Любые взаимодействия между параллельными процессами должны моделироваться явно, используя механизм передачи сигналов.

Приведенная диаграмма чрезвычайно проста, но она уже показывает достоинства и недостатки диаграмм состояний UML. Диаграммы состояний унаследовали синтаксис и семантику карт состояний Харела. Одной из целей создания карт состояния была возможность одновременно, на одной диаграмме, описывать параллельные процессы и их взаимодействие. Внутри отдельно параллельного процесса при этом оставались только последовательно меняющиеся состояния, которым предписано конкретное длительное состояние. В то же время наличие в одной диаграмме, предписанной классу, нескольких

параллельных процессов говорит о том, что класс устроен достаточно сложно. Критерием "простоты" в этом случае является отсутствие параллельных процессов. Иными словами, можно построить несколько классов, диаграммы состояний которых будут содержать только диаграммы состояний с последовательными переходами.

# Структурированные классификаторы

*Структурированный классификатор* предназначен для описания объектов, имеющих внутреннюю структуру. В нашем случае речь пойдет о классах и, следовательно, о классах, имеющих иерархическую структуру. Структурированный классификатор описывает не только части класса, но и связи между ними, называемые в этом случае *соединителями*. Соединители отличаются от связей ассоциаций. Ассоциации указывают лишь на логическую, концептуальную связь между классами, в то время как соединители предназначены для передачи информации между независимыми частями во время исполнения. Структурированный класс может быть описан различными способами, в частности с использованием ассоциаций (рис. 1.13). В данном случае указывается только, что на аэродроме есть одна служба слежения и одна транспортная служба (отсутствие значения множественности у полюса означает, что используется один экземпляр), а с ними может взаимодействовать заданное число летательных аппаратов и машин. Такие описания удобно применять на ранней стадии проектирования.

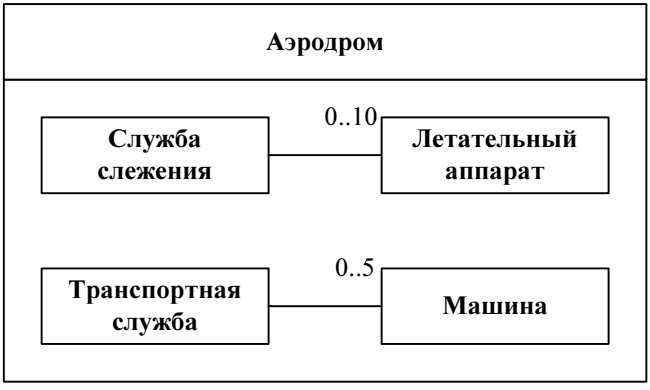



**Рис. 1.13.** Описание структурированного класса с помощью ассоциации

Еще одним высокоуровневым способом описания является описание части класса с точки зрения предоставляемого и требуемого интерфейсов с указанием портов, которые и обеспечивают заданный интерфейс (рис. 1.14).

На рис. 1.14 показан класс, который имеет четыре порта, каждый из которых "знает", куда передавать поступающую на него информацию (какой части структурированного класса). Связанный с портом интерфейс говорит о том, какие услуги предоставляет или требует заданный порт. Открытые (внешние) порты классов можно соединять между собой, при условии, что их интерфейс совпадает.

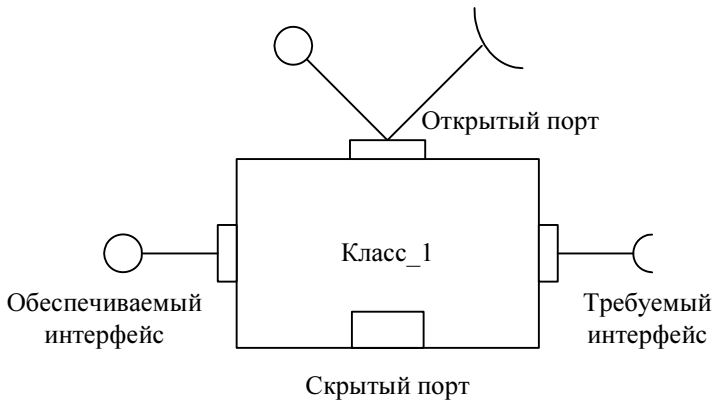



**Рис. 1.14.** Порты и интерфейсы класса

# Компоненты

> "Модели UML создаются как для логического анализа, так и для проектирования, обеспечивающего реализацию системы.
>
> Некоторые конструкции в модели представляют собой проектные элементы. Класс может инкапсулировать свою внутреннюю структуру за внешне видимыми портами.
>
> Компонент — это замещаемая часть системы, которая соответствует некоторому набору интерфейсов..."
>
> *Г. Буч, А. Якобсон, Дж. Рамбо [2]*

*Компоненты* (как графический символ в UML) являются разновидностью структурированного классификатора и их используют, когда функциональная схема слишком громоздка. Компонент указывает на то, что он имеет сложную структуру (для этой цели служит специальный значок), описывает все обеспечиваемые и требуемые интерфейсы (рис. 1.15, *а*) и, если нужно, свою внутреннюю структуру, как показано на рис. 1.15, *б*.

Структурированный компонент подсказывает (рис. 1.15), что компоненты с одинаковыми обеспечиваемыми и требуемыми интерфейсами взаимозаменяемы. В этом смысле, такие взаимозаменяемые компоненты можно назвать полиморфными, хотя они не связаны отношением обобщения.

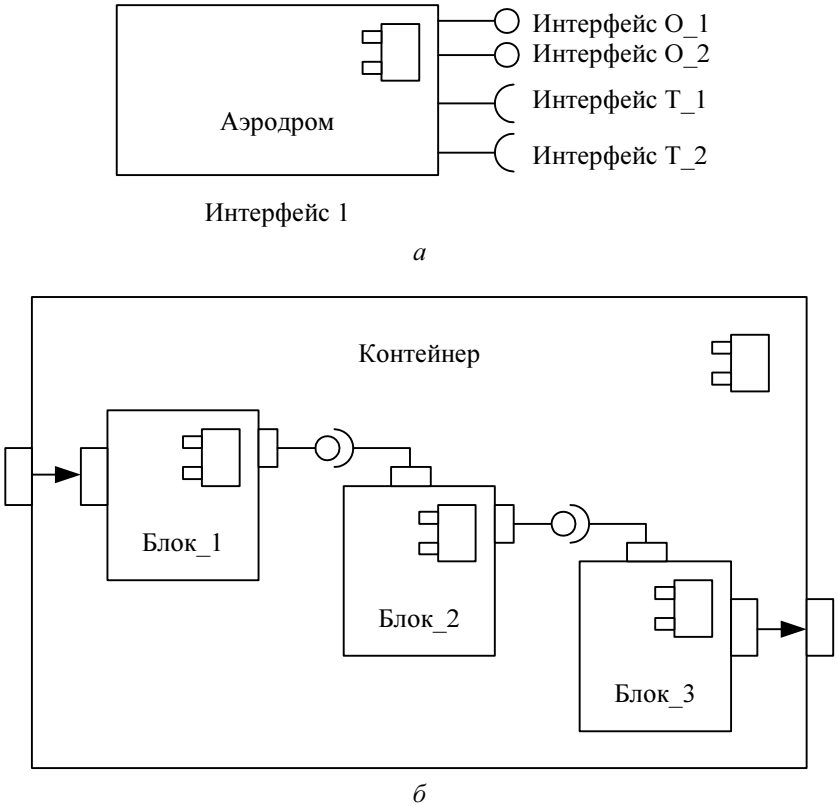



**Рис. 1.15.** Описание структурированных классов с помощью компонентов: *а* — компонент; *б* — структурированный компонент

# События и сигналы

В первом томе [4] под событием понимался факт достижения выбранной исследователем точки в пространстве состояний. Это могут быть точки, в которых происходит качественное изменение поведения модели. Точки, в которых можно сравнивать поведение модели и реального объекта. Точки, в которых переменные состояния выходят за предопределенные границы, что свидетельствует о неправильном поведении модели.

Авторы языка UML предлагают под событием понимать все важное с точки зрения исследователя, что может произойти во время работы моделируемой системы. При таком толковании событий в языке моделирования должны быть предусмотрены средства их описания, так как определение события уже не сводится к заданию времени его возникновения и (или) значения перемен-

ной состояния в этой временной точке. Информация о произошедших событиях в некотором компоненте модели может быть нужна исполняющей системе, другим компонентам, пользователю, наблюдающему за поведением модели, среде, в которой функционирует модель. Эта информация передается с помощью сообщений.

Сообщение, с точки зрения авторов UML, — это единичная операция передачи информации между отправителем и получателем. Сообщения отражают еще одно проявление динамики системы — информационное взаимодействие между отправителем и получателем сообщений. Возможна широковещательная передача сообщений (в этом случае сообщение передается всем возможным получателям) и направленная (конкретные получатели сообщений явно или неявно указаны).

Сообщение о событии может быть передано явно, с помощью посылки сигналов (переменных особого типа или экземпляров классов особого вида), или неявно, когда экземпляр определенного класса вызывает операцию экземпляра другого класса. В последнем случае речь также идет о взаимодействии — налицо и отправитель, и получатель информации. Взаимодействие может быть асинхронным — отправитель не рассчитывает на получение ответа и не прекращает своей работы после посылки сообщения, и синхронным, когда от получения ответа зависит дальнейшее поведение отправителя. Синхронное взаимодействие предполагает явный ответ получателя на посланное сообщение.

Рассмотрим пример. После успешного взлета самолета диспетчер узнает о неблагоприятных погодных условиях в аэропорту приземления (событие) и посылает сообщение пилоту в виде сигнала "Внимание! Сообщение об изменении аэропорта назначения", сопровождая его параметром — именем нового аэропорта назначения. Пилот подтверждает получение сигнала и меняет курс. Это пример синхронного взаимодействия в виде обмена сообщениями между одним отправителем и одним получателем. Это же сообщение может быть широковещательным, если по каким-то соображениям исходное сообщение посылается всем самолетам, находящимся в воздухе. Оно заставляет пилотов, летящих в этот город, менять курс (синхронизация и возникновение параллельных действий в системе), а для остальных является "информацией, которую нужно только принять к сведенью" (асинхронное взаимодействие). Предположим, что диспетчер может автоматически контролировать количество горючего в баках самолета, а также на основании этой информации принимать решение о новом аэропорте назначения. В этом случае он может вызвать процедуру определения количества топлива нужного самолета, и после принятия ответа послать сообщение об изменении курса. Для системы управления самолетом такой запрос является событием, его надо "идентифицировать" и правильно на него отреагировать.

В современных языках моделирования пользователь часто не имеет возможности явно описывать события (не существует переменных типа "событие" и или класса со стереотипом "событие", и не существует возможности написать свой пользовательский обработчик событий). Большинство событий являются предопределенными, и их возникновение умеет обнаруживать исполняющая система и приписывать параметрам, связанным с данным типом события, конкретные значения. Передача этих параметров возможна при посылке сигнала. Таким образом, значения этих параметров становятся доступными не только исполняющей системе, но и пользователям. В качестве неявно заданного параметра события всегда выступает время его возникновения.

В то же время события могут быть инициированы пользователем. К их числу, например, относятся события, приводящие к смене поведения.

В UML выделяются следующие события, связанные со сменой поведения:

- получение запроса на вызов операции (call event);
- изменение значения логического выражения (или предиката) в результате изменения значений переменных, входящих в его состав (change event);
- разновидность предыдущего события, приписанного к конкретному переходу и приводящего к смене поведения в картах поведения при изменении его значения и истинности сторожевого условия (change trigger);
- получение сигнала (signal event);
- истечение заданного отрезка времени (time event).

Первый тип событий говорит о возникновении нового вида деятельности, и в этом смысле связан со сменой поведения системы. Последние четыре типа событий пользователь может использовать при конструировании карты поведения, в качестве условия срабатывания перехода, наряду с охраняющими (сторожевыми) условиями (дополнительный предикат, устанавливаемый пользователем). Охраняющие условия позволяют уточнить условия срабатывания перехода при возникновении события выбранного типа. Эти события инициирует пользователь. В любой модели существуют и события, видимые только исполняющей системе — события, связанные с возникновением и уничтожением объектов, появлением особых чисел, таких как NaN. Реакция на эти внутренние события предопределена исполняющей системой, в частности она может позволить обрабатывать эти события пользователю.

Информация о событиях передается с помощью сообщений, реализованных или в виде посылки сигналов, либо запроса на вызов операции. Передача сигнала

`send` *`имя_сигнала`*`(`*`список_действительных_параметров`*`)`

или вызов операции

`call` *`имя_операции`*`(`*`список_действительных_параметров`*`)`

являются сообщениями.

Сигнал, с точки зрения UML, является именованным классификатором. У сигнала есть список параметров, представленный как его атрибуты. Таким образом, определяя сигнал и соответствующий ему список параметров (`имя_сигнала(список_формальных_параметров): signal`), пользователь явно определяет некоторое важное для поведения системы событие, отличное от перечисленных выше предопределенных событий.

Так как сигнал является классификатором, то он может иметь предков, от которых наследует свои атрибуты *(см. разд. "Наследование" ранее в этой главе)*. К числу атрибутов сигнала могут относиться и операции, дающие доступ к атрибутам или изменяющие их. Эти операции могут использоваться при создании экземпляра сигнала и для открытия доступа к его атрибутам. Определяя класс или интерфейс, пользователь может объявить в них сигналы, которые они готовы отрабатывать. Для графического изображения сигнала используется такой же прямоугольник, как для изображения класса, с такими же секциями, но перед именем сигнала ставится слово "Signal", взятое в кавычки.

Отправив сигнал, отправитель продолжает свою работу. Для получателя получение сигнала является событием, на которое он должен отреагировать, если это предусмотрено картой состояний, т. е. речь идет асинхронной передаче информации. Если отправитель и получатель должны взаимодействовать, то необходимо явно послать сигнал от объекта, получившего сигнал, к объекту, его пославшему. В этом случае можно говорить о синхронизации процессов отправителя и получателя.

# Пакеты

> "Пакетами в UML называются иерархически организованные блоки моделей".
>
> *Г. Буч, А. Якобсон, Дж. Рамбо [2]*

Описание всей модели может содержать документацию, собственно модель или несколько ее вариантов на различных уровнях абстракции, результаты натурных и вычислительных экспериментов, т. е. весьма разнородную информацию, которую лучше систематизировать и хранить отдельно. Даже сама UML-модель содержит элементы и диаграммы различного типа. Пакеты позволяют упорядочить и систематизировать все элементы модели, присвоив

им уникальные имена и расположив их в определенном месте. Естественно, что пакеты представляют собой иерархические структуры, элементы пакетов могут ссылаться на другие пакеты. Таким образом, их можно представить в виде графа.

Вложенный пакет имеет доступ ко всем элементам окружающих его пакетов. Сам пакет может ограничить доступ к своим элементам, если это необходимо. Пакет может объявить свои элементы видимыми (публичными) или невидимыми (приватными). Пакет-контейнер, содержащий подпакеты, не видит содержимого своих пакетов и вынужден импортировать нужные ему элементы.

# Модель

"Модель — это пакет, в котором содержится полное описание системы, сделанное с определенной точки зрения".

*Г. Буч, А. Якобсон, Дж. Рамбо [2]*

"Модель должна иметь внутреннюю организацию, которая позволила бы работать с ней сразу нескольким рабочим группам. Это требование исходит вовсе не из семантики, т. к. большая монолитная модель системы была бы не менее точной, нежели набор моделей, распределенных в связанные между собой пакеты. Может быть, цельная модель была бы более точной, но с такой моделью не смогут одновременно работать несколько групп разработчиков, т. к. при этом они будут постоянно мешать друг другу".

*Г. Буч, А. Якобсон, Дж. Рамбо [2]*

*Модель* является разновидностью пакета (рис. 1.16). С точки зрения авторов UML, разделение модели на части и распределение всей необходимой информации по пакетам производится, в первую очередь, для обеспечения возможности работать над созданием модели нескольким коллективам одновременно. В то же время они отмечают, что "язык UML не налагает жестких правил в части разбиения модели на пакеты, однако чем лучше вы это сделаете, тем проще вам будет поддерживать вашу модель". Самым простым разбиением на пакеты может быть разбиение модулей модели по принадлежности к используемым библиотекам, создание специальных пакетов, хранящих документацию, версии, результаты натурных и вычислительных экспериментов.

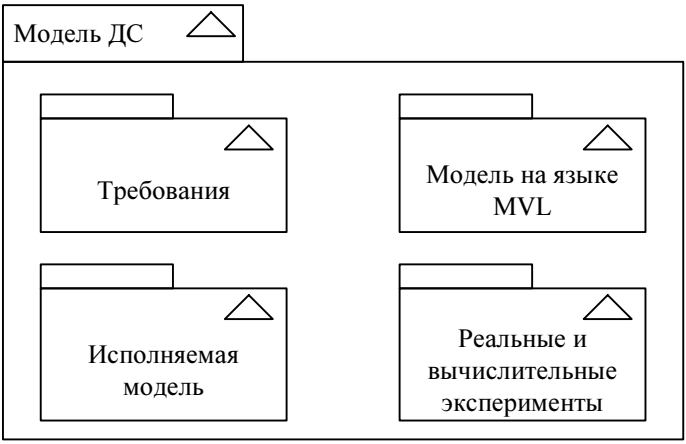
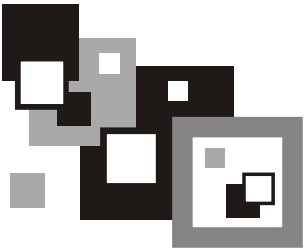



**Рис. 1.16.** Модель

ГЛАВА 2



# Объектно-ориентированное моделирование сложных динамических систем на основе формализма гибридного автомата

В данной главе рассматриваются принципы построения языка объектно-ориентированного моделирования сложных динамических систем (СДС) на базе математической модели гибридного автомата, рассмотренной в книге [4]. На наш взгляд, формализм гибридного автомата позволяет создать мощный и естественный для человеческого восприятия язык моделирования СДС, который в то же время максимально соответствует синтаксису и духу языка UML [2]. На основе формализма гибридного автомата уже построены входные языки пакетов моделирования AnyLogic [11], MvStudium [4], Ptolemy [10], Shift [9].

Предлагаемый в этой главе язык моделирования рассматривается нами как идеал, к которому мы стремимся при создании пакета моделирования MvStudium 4. К сожалению, не все возможности "идеального" языка удалось реализовать в существующей версии пакета (**http://www.mvstudium.com**). Мы надеемся, что более полная и совершенная версия пакета станет доступной читателям книги Колесов Ю. Б., Сениченков Ю. Б. "Моделирование систем. Примеры и задачи" уже в 2006 году.

Другие подходы к ООМ СДС, использующие иные базовые математические модели, рассматриваются в *главе 3*.

## Активный класс и активный динамический объект

Основным элементом языка ООМ СДС является активный класс, экземпляр которого будем называть *активным динамическим объектом* (АДО). Актив-

ные классы предназначены для описания компонентов моделей СДС. В языке UML также существуют активные классы, и их экземпляры также называются *активными объектами*, т. к. им присуща собственная внутренняя деятельность, независимая от поведения других объектов.

С точки зрения описания, активному классу соответствует некоторая совокупность переменных и поведения — непрерывного, дискретного или гибридного [4].

Атрибутам активного класса в рассматриваемом языке моделирования соответствуют переменные, имеющие уникальное имя, вид, тип, и явно или неявно заданное начальное значение. Напомним, что каждая переменная может быть: константой, параметром, переменной состояния, входом, выходом, контактом или потоком, а принадлежность к одному из перечисленных множеств и определяет ее вид. Таким образом, секция атрибутов в описании активного класса с точки зрения синтаксиса отличается от секции атрибутов классов языка UML. Она более детализирована.

Поведение экземпляра активного класса UML задается картой состояний (statechart), а поведение АДО — гибридным автоматом. Гибридный автомат, или карта поведения (behavior chart, или B-chart), является обобщением карты состояния. И здесь описание активного класса предлагаемого языка отличается от описания активного класса UML — ссылка на карту поведения включена в описание класса.

Карты поведения в свою очередь также можно классифицировать. Как уже упоминалось, изолированные динамические системы с непрерывным временем можно представить только одними уравнениями, не связывая их с примитивным гибридным автоматом, имеющим одно состояние и единственный переход в особое конечное состояние по заданному условию. Для пользователей, имеющих дело в основном с изолированными динамическими системами, целесообразно выделить в языке моделирования особое "непрерывное поведение", как частный случай карты с состоянием, которому приписана система уравнений (рис. 2.1).

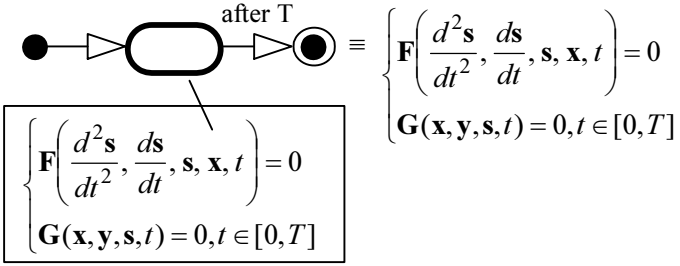



**Рис. 2.1.** Непрерывное поведение как частный случай гибридного поведения

На рис. 2.1 приведена система алгебро-дифференциальных уравнений в том виде, как она может быть задана пользователем во входном языке. Напомним, что $\mathbf{s}$, $\mathbf{x}$, $\mathbf{y}$ — это векторы, соответствующие переменным состояния, входам и выходам компонента. Присутствие вторых производных $\frac{d^2\mathbf{s}}{dt^2}$ служит напоминанием, что входной язык допускает уравнения второго порядка без их сведения к уравнениям первого порядка. Для графического языка эта запись слишком громоздка, поэтому в дальнейшем вместо нее будет использоваться символическое изображение решаемой системы $\left\{ F(\frac{d^2X}{dt^2}, \frac{dX}{dt}, X, t) = 0 \right.$, где переменные не разделяются по видам, могут быть и скалярами и векторами, а сами уравнения могут быть алгебраическими, обыкновенными дифференциальными, или алгебро-дифференциальными.

Таким образом, присущая АДО деятельность может, в отличие от объекта UML, быть непрерывной и не связанной ни с каким "потоком управления". Единственным "движителем" непрерывной деятельности выступает независимый и глобальный поток непрерывного времени. Это и позволяет называть экземпляры активных классов активными динамическими объектами. Другим отличием АДО от объекта UML является способ взаимодействия с внешним окружением. Объекты UML могут взаимодействовать либо посредством посылки другому объекту сообщения, либо посредством прямого вызова видимой извне операции другого объекта. В первом случае в принимающем объекте может сработать переход карты состояний, ожидающий это сообщение, во втором случае операторы метода вызываемого объекта просто вставляются в последовательность действий, выполняемую в карте состояния вызывающего объекта. Первый случай взаимодействия — передача сообщений или сигналов — имеет место и для дискретных компонентов моделей СДС. Второй случай — вызов методов — характерен только для программных объектов и компонентами моделей СДС не используется. В то же время, компоненты моделей СДС могут взаимодействовать между собой непрерывным образом непосредственно через внешние переменные.

Описание активного класса включает в себя описание внешних и внутренних переменных и описание поведения. Понятие внешних и внутренних переменных непосредственно связано с понятием видимости переменных и инкапсуляцией данных. Внешние переменные являются видимой частью АДО (интерфейсом), а внутренние переменные и поведение инкапсулированы внутри АДО (рис. 2.2). Значения внутренних переменных могут изменяться только изнутри АДО. Значения внешних переменных могут изменяться как вне объекта, так и внутри него. Начальные значения переменных и значения пара-

метров указываются при создании АДО. Значения параметров остаются неизменными на протяжении всего времени существования экземпляра активного класса. С каждой непрерывной переменной $X$, имеющей тип: скалярный вещественный, векторный или матричный, автоматически связывается набор дополнительных переменных с именами $\frac{dX}{dt}$, $\frac{d^2X}{dt^2}$, ..., соответствующих их производным, не требующих специального описания и вычисляющихся автоматически.


**Рис. 2.2.** Описание активного класса

Поведение АДО в общем случае является суперпозицией собственного поведения и совокупного поведения локальных объектов с учетом связей (рис. 2.3). По отношению к локальным объектам содержащий их объект является объектом-контейнером. В синтаксисе UML локальные объекты — это атрибуты с семантикой указателя. Собственное поведение может задаваться либо системой дифференциально-алгебраических уравнений общего вида, либо картой поведения (рис. 2.3). Заметим, что в предлагаемом языке активный класс может содержать только одну карту поведения, что облегчает понимание поведения компонента модели и упрощает синхронизацию параллельно работающих гибридных автоматов [4]. Для задания собственного поведения могут также потребоваться локальные алгоритмические функции или процедуры, а также локальные классы. Следует отметить, что эти алгоритмические функции и процедуры являются элементами описания собственного поведения и инкапсулированы внутри описания класса, поэтому к ним нельзя обратиться извне.

В *главе 1 книги* [4] были выделены три "ортогональных" направления одной из возможных классификаций компонентов моделей СДС:

- по типу взаимодействия с внешним миром: открытые и изолированные компоненты;

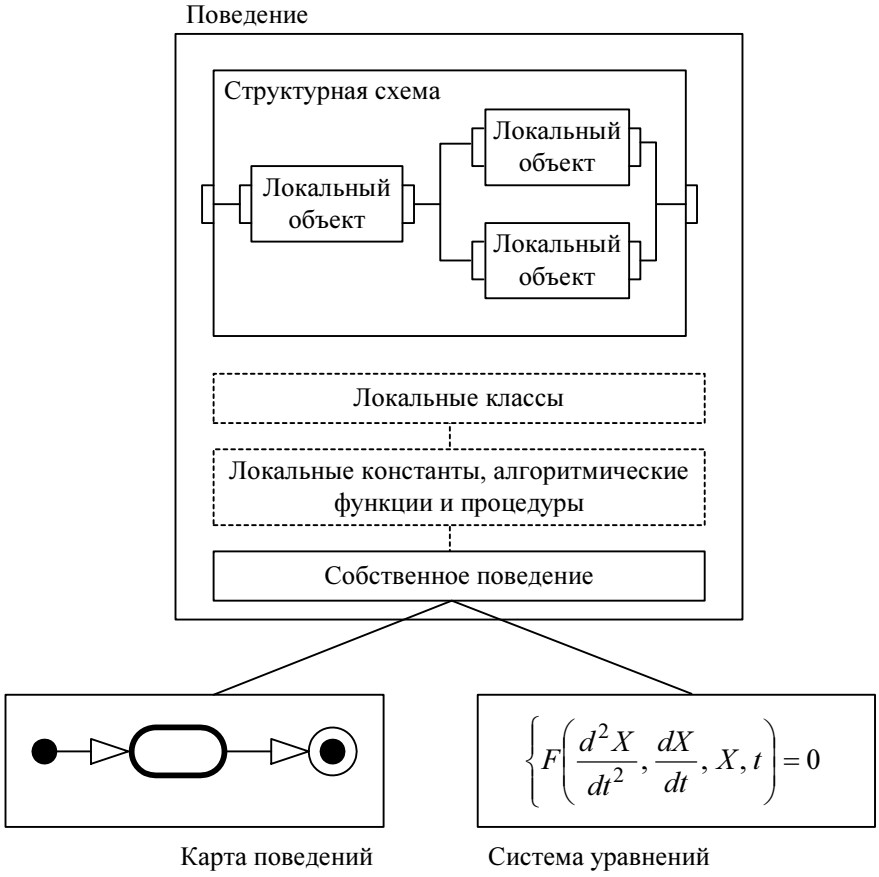



**Рис. 2.3.** Составляющие поведения активного класса

- ❒ по типу внутренней структуры: элементарные и составные компоненты;
- ❒ по типу собственного поведения: непрерывное поведение, гибридное поведение, нет собственного поведения.

Для подобного рода семантической типизации в языке UML существует специальная конструкция — *стереотип* (stereotype). Будем считать, что стереотип для активных классов, порождающих АДО, — это одна из 12-ти возможных комбинаций перечисленных выше классификационных признаков. Первый классификатор определяет наличие или отсутствие внешних переменных. Второй задает наличие или отсутствие структурной схемы. Третий определяет, есть ли у объекта собственное поведение, и если есть, то что это — система уравнений или карта поведения. Указание стереотипа полезно при работе в интегрированной среде пакета моделирования, т. к. позволяет

рационально конфигурировать редактор описания класса, убирая ненужные для данного стереотипа поля и окна и показывая только необходимые.

Возможные преобразования стереотипа:

- любой класс с непрерывным поведением может быть преобразован в эквивалентный класс, поведение которого задано картой поведения (см. рис. 2.1);
- класс с поведением, заданным в виде карты поведения, показанной на рис. 2.1, может быть преобразован в эквивалентный класс с непрерывным поведением (предполагается, что если состоянию не приписано никакой деятельности, то это эквивалентно непрерывной деятельности с "пустой" системой уравнений);
- любой класс, соответствующий изолированной системе, может быть преобразован в класс, соответствующий открытой системе с пустым набором внешних переменных, и наоборот;
- любой класс, имеющий элементарную структуру, может быть преобразован в класс с составной структурой с пустым набором локальных объектов и связей, и наоборот.

На практике наиболее часто используются четыре основных стереотипа:

- класс, порождающий непрерывный элементарный объект;
- класс, порождающий дискретный элементарный объект;
- класс, порождающий гибридный элементарный объект;
- класс, порождающий "схему".

Первому случаю соответствует активный класс, показанный на рис. 2.4. Собственное поведение порождаемого объекта задается системой уравнений. В системе уравнений, а также в выражениях для начальных значений переменных могут использоваться локальные функции.

Второму случаю соответствует активный класс, показанный на рис. 2.5. Собственное поведение порождаемого объекта задается частным случаем карты поведения, эквивалентной карте состояний без вложенных параллельных состояний. В условиях срабатывания, мгновенных действиях и в выражениях для действительных параметров деятельности в состоянии, а также в выражениях для начальных значений переменных могут использоваться локальные функции. Состояниям могут приписываться в качестве деятельностей экземпляры локальных классов.

Третьему случаю соответствует активный класс, показанный на рис. 2.6. Собственное поведение порождаемого объекта задается общим случаем карты поведения. Использование локальных констант, функций и процедур, а также локальных классов аналогично предыдущему случаю.

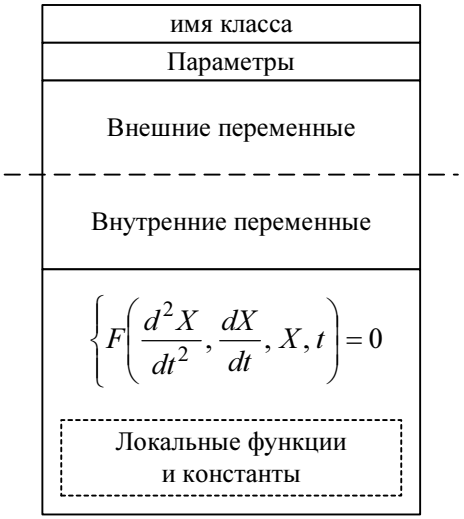



**Рис. 2.4.** Класс с непрерывным поведением без внутренней структуры

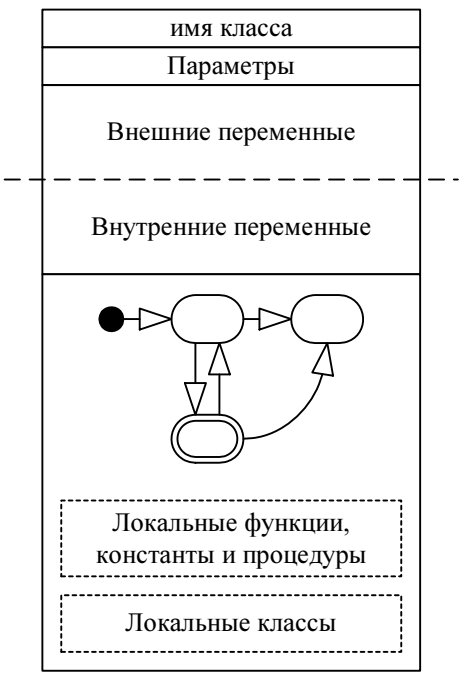



**Рис. 2.5.** Класс с дискретным поведением без внутренней структуры

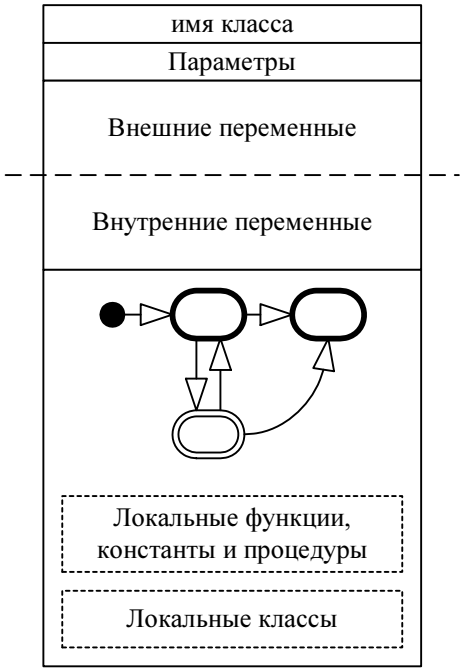



**Рис. 2.6.** Класс с гибридным поведением без внутренней структуры

Четвертому случаю соответствует активный класс ("схема"), показанный на рис. 2.7. Порождаемый объект не имеет собственного поведения, его функционирование определяется суперпозицией поведений локальных объектов и связей. При отсутствии собственного поведения нет необходимости в собственных внутренних переменных и в локальных классах. Локальные константы функции могут использоваться при инициализации внешних переменных, а также при задании действительных параметров для локальных объектов.

Экземпляр активного класса может быть создан или уничтожен:

- ❒ явно с помощью специальных операторов, выполняемых в последовательности мгновенных дискретных действий;
- ❒ неявно при создании и уничтожении объекта-контейнера, в состав которого данный объект входит в качестве локального;
- ❒ неявно при входе в состояние и выходе из состояния, которому приписан данный объект в качестве деятельности.

Экземпляр модели в целом создается исполняющей системой пакета моделирования в начале вычислительного эксперимента и уничтожается по его завершении.

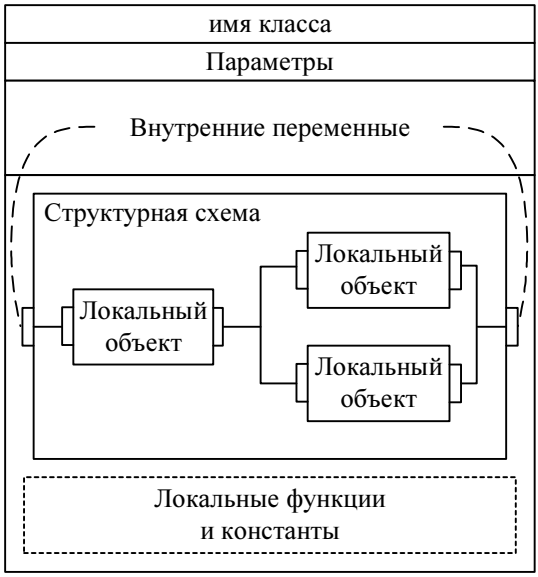



**Рис. 2.7.** Класс — "схема"

При создании экземпляра активного класса могут быть указаны действительные значения его параметров, отличные от значений "по умолчанию", указанных в определении класса. Весьма удобной при построении сложных моделей является возможность указывать при создании экземпляра активного класса также и действительные начальные значения его переменных.

Для любого АДО определена функция `time`, возвращающая значение локального времени этого объекта.

# Пакеты и модель

*Пакет* в предлагаемом языке — это контейнер для группы элементов, ограничивающий область их видимости. Элементы, объявленные как экспортируемые, видимы извне под составным именем, включающим в качестве префикса имя пакета, например, `P.E`, где `P` — имя пакета, а `E` — имя элемента в этом пакете. Остальные элементы видимы только внутри данного пакета. В описании элемента пакета видимы все остальные элементы этого же пакета. Пакет образует собственную область видимости и все элементы пакета должны иметь несовпадающие имена. Элементами пакета могут являться другие пакеты. В отличие от языка UML, где элементами пакета являются только классы, естественными элементами пакета в языке ООМ также явля-

ются константы, алгоритмические функции и процедуры и определения типов переменных (рис. 2.8).

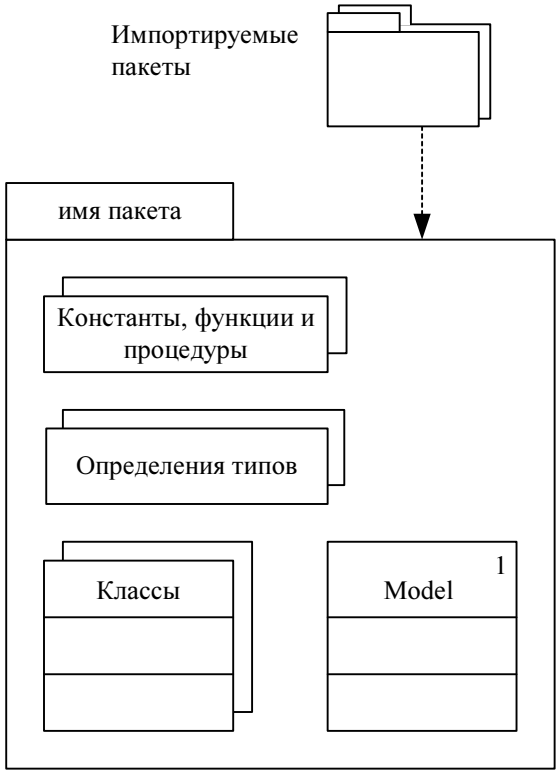



**Рис. 2.8.** Структура пакета

Для того чтобы использовать экспортируемые компоненты пакета `A` в пакете `B`, необходимо в писании пакета `B` поместить указание импортирования.

В пакете языка OOM имеется предопределенный класс `Model`, который является по терминологии UML "синглетным", т. е. может иметь только один экземпляр. Этот единственный экземпляр с именем `model` и является выполняемой моделью, с которой проводится вычислительный эксперимент. Значение функции `time` для этого объекта соответствует значению глобального времени вычислительного эксперимента. Опыт показывает, что на практике большинство моделей являются элементарными, изолированными непрерывными системами. Для создания такой простейшей модели достаточно только определить переменные модели и задать ее систему уравнений, не используя никаких знаний об объектно-ориентированном моделировании.

# Использование пассивных объектов

В сложных моделях обычно возникает необходимость в более мощных средствах манипулирования данными, чем простые математические выражения. Кроме того, для описания логики мгновенных действий требуются алгоритмические операторы, такие как условный оператор, оператор цикла и т. д. В последовательных действиях также удобно использовать процедуры, а в выражениях — функции. Наконец, в языке ООМ было бы правильно иметь возможность использовать помимо активных динамических объектов еще и "обычные" — пассивные алгоритмические.

Разработчики инструментов моделирования используют при решении этой проблемы два подхода:

- язык моделирования создается как расширение какого-нибудь языка программирования (Fortran, Simula, C, Java и т. п.);
- язык моделирования поддерживает относительно небольшое множество алгоритмических конструкций, а для более сложных действий необходимо использовать внешние программные компоненты (DLL, серверы COM, сборки .NET и т. п.), написанные непосредственно на языках программирования.

Примерами первого подхода являются практически все ранние инструменты моделирования. Примером языка ООМ, "надстроенного" над объектно-ориентированным языком программирования, служит входной язык пакета AnyLogic [11], который является расширением языка Java. Примерами второго подхода являются такие пакеты, как Omola, Dymola, а также пакеты семейства MvStudium [4]. Например, в пакете MvStudium в качестве "внутреннего" алгоритмического языка используется небольшое подмножество языка программирования Ada, включающее в себя оператор присваивания, оператор вызова процедуры, условный оператор, оператор варианта, оператор цикла, оператор выхода из цикла и оператор возврата. Кроме того, "внутренний" алгоритмический язык может предусматривать и небольшой набор предопределенных пассивных объектов, например, список.

Первый подход помимо очевидных достоинств имеет и серьезные недостатки, которые являются оборотной стороной достоинств. Во-первых, сложный и мощный базовый язык программирования чрезвычайно затрудняет создание интерактивного инкрементального транслятора и заставляет ориентироваться на доступные пакетные компиляторы (например, `javac` для языка Java). Это означает, что контроль правильности модели откладывается до полной компиляции и возможны проблемы с диагностикой ошибок. Во-вторых, при использовании стандартного компилятора затруднен сам контроль семантики, и некоторые действия приходится делать на стадии выпол-

нения модели. Наконец, первый подход требует определенного уровня знания базового языка программирования у пользователя. Поэтому второй подход представляется для языков ООМ более перспективным. Процедуры и функции должны быть элементами описания поведения динамического объекта и не могут вызываться извне другими объектами, т. к. динамический объект взаимодействует с окружением только через внешние переменные. Внутри процедур и функций динамического объекта видимы переменные объекта (в теле процедуры могут быть также изменены их значения). Для определения внутренних процедур и функций динамического объекта вполне достаточно относительно небольшого подмножества какого-нибудь известного языка программирования (например, Java). Современные подходы к компонентному программированию (например, в MS .NET [5]) позволяют использовать в описании модели только определения классов пассивных объектов, программный код которых находится в независимо разработанных с использованием любого удобного языка программирования "сборках".

# Переменные

Переменные в модели могут являться атрибутами активного класса, атрибутами локального класса, локальными переменными в алгоритмической функции или процедуре. Переменные имеют семантику значения, если в качестве типа переменной указан тип данных, и семантику указателя, если в качестве типа переменной указан класс объекта. Все переменные — атрибуты — видимы в описании локальных классов, а также в алгоритмических функциях и процедурах данного класса. Определение переменной в общем случае включает в себя:

- стереотип переменной (вид переменной);
- идентификатор переменной;
- указание типа переменной;
- начальное значение переменной.

В определении рабочей переменной отсутствует указание стереотипа.

*Стереотип* переменной отражает семантические правила ее использования:

- "состояние" (`state`) — внутренняя переменная, используемая только в описании собственного поведения объекта;
- "вход" (`input`) — внешняя переменная, значение которой может быть изменено только извне;
- "выход" (`output`) — внешняя переменная, значение которой может быть изменено только внутри объекта;

- "контакт" — внешняя переменная, значение которой может быть изменено как извне, так и внутри объекта;
- "поток" (`flow`) — аналогичен стереотипу "контакт", но подразумевает наличие дополнительных уравнений связи при соединении;
- "разъем" (`connector`) — внешняя переменная может участвовать в связях.

Стереотип может также указываться в определении типа данных.

*Идентификатор* переменной представляет собой строку, состоящую из латинских и русских букв, цифр и знака подчеркивания и начинающуюся с буквы.

*Указание типа* переменной представляет собой имя типа (предопределенного, определяемого в пакете или импортируемого) или определение анонимного типа и задает тип значения переменной и, возможно, ее стереотип.

*Начальное значение* переменной представляет собой выражение, в котором могут использоваться константы, функции, а также начальные значения других переменных (при этом только не должны возникать алгебраические циклы). При создании экземпляра данного класса все переменные приобретают указанные начальные значения или значение "не присвоено", если в определении класса не указано начальное значение. Значения параметров и начальные значения переменных конкретного экземпляра могут быть явно указаны при вызове конструктора, например:

```
Маятник(L=2, Alpha=-pi/2, Omega=1);
```

В этом случае значения, указанные в определении класса, игнорируются.

# Типы данных

Для моделирования непрерывных систем необходим минимальный набор типов данных: скалярный вещественный тип, типы "вектор" и "матрица" со своими традиционными операциями, а также целые числа для вычисления индексов векторов и матриц. Для моделирования дискретных и гибридных систем необходимо также иметь более широкий спектр целых типов (байт, короткое целое, длинное целое), перечислимые, булевы, символьные и строковые типы, а также одномерные и двумерные массивы с элементами любого скалярного типа. Кроме того, для описания явной синхронизации параллельных процессов нужны специфические переменные-сигналы. Для систем со сложной структурой крайне желательно наличие типа "запись". С помощью этого типа очень удобно передавать в компактной форме набор взаимосвязанных данных (возможно, различных типов) между структурными компонентами. Структура типов данных показана на рис. 2.9.

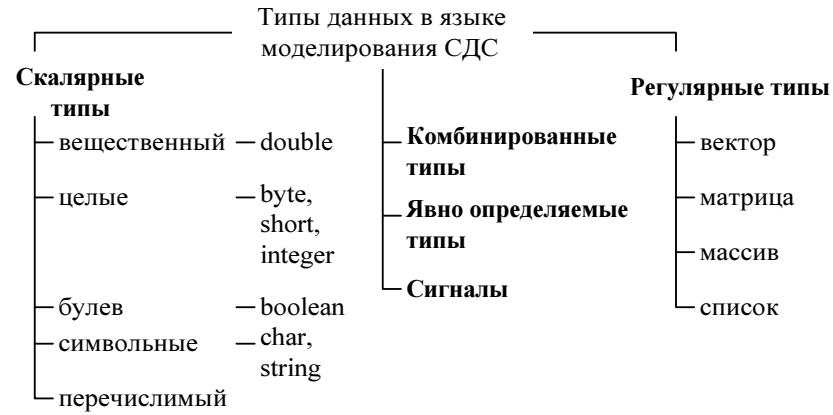



**Рис. 2.9.** Типы данных

# Скалярные типы

К скалярным типам относятся вещественный, целые, булев, перечислимый и символьные типы.

## Вещественный тип

Для приближенного представления вещественных чисел в языке моделирования используется тип `double`, соответствующий стандарту вычислений с плавающей точкой [8]. С помощью этого типа, имеющего внутреннее представление длиной в 8 байтов, могут быть представлены вещественные числа со знаком в диапазоне $5.0 \times 10^{-324}$ — $1.7 \times 10^{308}$ с точностью 15—16 значащих десятичных цифр в мантиссе. Для типа `double` определены следующие операции и отношения: сложение, вычитание, умножение, деление, возведение в целую и вещественную степень, равенство, неравенство, больше, больше или равно, меньше, меньше или равно.

Примеры вещественных литералов:

```
-3.5  +5.67  1.5E3  -3.4E12  1.76E-2  4 2e3
```

## Целые типы

В языке моделирования необходимы следующие целые типы:

- `byte` (8 битов без знака, диапазон чисел 0—255);
- `short` (16 битов со знаком, диапазон чисел –32 768 — 32 767);

- `integer` (32 бита со знаком, диапазон чисел –2 147 483 648 — 2 147 483 647);
- `int64` (64 бита со знаком, диапазон чисел $-2^{63}$ — $2^{63} - 1$).

Для целых типов определены следующие операции и отношения: сложение, вычитание, умножение, целое деление, сравнение по модулю, возведение в целую степень, побитовое "ИЛИ", побитовое "И", побитовое "НЕ", равенство, неравенство, больше, больше или равно, меньше, меньше или равно.

Примеры целых литералов:

```
1  34  -4567
```

## Булев тип

Тип `boolean` имеет множество значений {`false`, `true`}. Для булева типа определены следующие операции и отношения: логическое "ИЛИ", логическое "И", логическое "НЕ", равенство, неравенство.

## Перечислимые типы

Перечислимые типы определяются путем явного задания (перечисления) конечного множества значений как упорядоченной совокупности не совпадающих по именам литералов вида ($e_0, e_1, e_2, ..., e_n$), где $e_i$ — идентификатор литерала с номером $i$. Два перечислимых типа являются одинаковыми, если их множества значений совпадают. Для литералов перечислимого типа определены следующие отношения: равенство, неравенство, больше, больше или равно, меньше, меньше или равно. Результат этих отношений равен результату соответствующих отношений между номерами значений. Значения различных перечислимых типов несравнимы. Перечислимые литералы задаются своими идентификаторами. Литералы различных перечислимых типов могут иметь совпадающие идентификаторы.

Пример определения перечислимого типа:

```
(Alpha, Beta, Gamma)
```

## Символьные типы

К символьным типам относятся собственно символьный тип `char` и строковый тип `string`. Тип `char` включает в себя упорядоченное множество символов. Тип `string` — это строка произвольной длины. Множество символов следует рассматривать как специальный случай перечислимого типа с соответствующими отношениями. Для строк определены следующие операции и отношения: конкатенация, равенство, неравенство. Символьные литералы задаются соответствующим символом, заключенным в кавычки. Примеры

символов: "A", "1", "a". Строковые литералы задаются соответствующей строкой, заключенной в кавычки. Примеры строк: "abcd", "1234".

## Регулярные типы

К регулярным типам относятся векторы, матрицы, массивы и списки.

### Векторы

Определение типа `vector[N]` задает вектор-столбец фиксированного размера `N` с элементами типа `double`. Определение типа `vector` задает вектор-столбец переменного размера с элементами типа `double`. Элементы вектора всегда нумеруются с 1. Контроль правильности использования вектора с переменного размера возможен только во время исполнения. Текущий размер вектора всегда можно определить с помощью функции `size()`. Для векторов определены следующие операции и отношения: умножение на скаляр, сложение, вычитание, транспонирование, равенство, неравенство.

Примеры векторных литералов:

```
[1;2;3;4]  [0; 0; 2.3; 5.67; 1E2]
```

`[for i in 1..10 | i**2 ]` — вектор размера 10, содержащий значения 1, 2, 9, ..., 100 (итеративный векторный литерал).

### Матрицы

Определение типа `matrix[N,M]` задает прямоугольную матрицу фиксированного размера с `N` строками и `M` столбцами с элементами типа `double`. Определение типа `matrix` задает прямоугольную матрицу переменного размера. Элементы матрицы всегда нумеруются с 1 по обоим измерениям. Вектор всегда можно рассматривать как матрицу `[N,1]`. Контроль правильности использования матрицы переменного размера возможен только во время исполнения. Текущий размер матрицы всегда можно определить с помощью функции `size()`. Для матриц предусмотрены следующие операции и отношения: умножение матрицы на скаляр, умножение матриц, сложение, вычитание, транспонирование, равенство, неравенство.

Примеры матричных литералов:

- `[1, 2, 3, 4; 1, 4, 5, 6]` — матрица размера 2×4;
- `[0,3.4,5;7,8,0.67;0.8,6,2.3]` — матрица размера 3×3.
- `[for i in 1..3, j in 1..5 | i*j-1]` — матрица размера 3×5 (итеративный матричный литерал).

Итеративные матричные литералы можно использовать в качестве начального значения переменных, а также в качестве правой части формулы или оператора присваивания.

## Массивы

В языке моделирования поддерживаются одномерные и двумерные массивы с элементами какого-либо видимого в данной точке описания типа. Границы индексации элементов либо указываются явно в виде диапазона, либо не указываются совсем. Примеры определений массивов:

```
A1: array [1..3] of boolean;
A2: array [0..4, 1..2] of integer;
A3: array of double;
A4: array [0..5] of double;
```

Для массивов определены отношения равенства и неравенства. Два массива с совместимыми типами элементов равны, если их размерности одинаковы и соответствующие элементы равны. Для логических массивов имеются две предопределенные логические функции: `any` и `all`, результат которых соответствует операциям "ИЛИ" и "И" над элементами массива.

Для массивов с неопределенной размерностью текущие границы индексации доступны через предопределенные функции `low` и `high`. При присваивании массивов значения копируются, например, после выполнения оператора

```
A3:=A4;
```

будут верны соотношения

`low(A3)=0`, `high(A3)=5`, `A=A4`

Примеры литералов-массивов:

```
{true,false,true}
{0,1,2,3,4,
0,0,0,1,1}
```

В операциях с массивами могут также использоваться итеративные литералы:

```
A3:={for i in 1..4 | 2**i }
```

## Списки

Список является предопределенным пассивным классом. Список аналогичен одномерному массиву соответствующего типа с нижней границей индексации 0 и неопределенной верхней границей индексации. У списка имеются операции добавления (`add`), удаления (`delete`, `remove`) и вставки (`insert`) элементов. В список может быть добавлен также другой список или массив соответствующего типа. Списку может быть присвоен другой список или мас-

сив соответствующего типа, в этом случае осуществляется копирование элементов. Литералы-списки соответствуют литералам — одномерным массивам. Для списков определены отношения равенства и неравенства, а также операция доступа к элементу по индексу.

Примеры определений списков и операций над списками:

```
L1: list of integer:={1,2,3};
L2: list of double;
...
L1.add(5);       -- L1={1,2,3,5}
L1.insert(1,10); -- L1={1,10,2,3,5}
L2:=A3;          -- L2={2,4,8,16}
L1[2]:=0;        -- L1={1,10,0,3,5}
```

## Комбинированный тип (запись)

Переменная комбинированного типа есть последовательность поименованных компонентов. Компоненты записи могут принадлежать к различным типам (рекурсии в определении записи не допускаются). Примеры определений записей:

```
record
  A: integer;
  B: boolean;
  C: matrix[2,3];
end record;
```

Примеры комбинированных литералов:

```
{A=>2, B=>true, C=>[1,2,3; 0,0,1]}
```

Для записей определены отношения равенства и неравенства (покомпонентно).

## Явно определяемые типы

Декларация типа позволяет связать идентификатор типа с некоторым определением типа и в дальнейшем использовать этот идентификатор для задания типа констант, переменных и формальных параметров. Примеры деклараций типа:

```
type T1 is integer;
type T2 is matrix[2,2];
type T3 is (A,B,C);
type T1_1 is T1;
```

```
type T4 is record
             X: T2;
             Y: double;
             Z: array [0..2] of boolean;
             W: T3;
           end record;
```

Литералы определяемого типа соответствуют его определению:

```
V1: T1 := 2;
V2: T2 := [1,2;2,4];
V3: T3 := B;
V4: T4 := (X=>[0,1;1,0], Y=>4.67, Z=>(false,false,true), W=>A);
```

В определении типа может также быть указан стереотип переменных, имеющих значения данного типа, например:

```
type Voltage is double;
type Current is double;
connector type Pin is
  record
    V: Voltage;
    flow I: Current;
  end record;
```

Переменная, декларированная как

```
N: Pin;
```

автоматически приобретает стереотип `connector`.

## Сигналы

Как отмечалось выше, переменные типа `signal` — это сообщения (возможно, с параметрами), передаваемые между параллельно выполняемыми процессами с целью их явной синхронизации. Формальные параметры сигнала должны декларироваться в определении переменной или типа, например:

```
T: signal (V: double; Teta: double);
type Throw is signal (V: double; Teta: double);
T1: Throw;
SignalA: signal;
```

С переменной-сигналом можно выполнить только одно действие: послать сигнал с помощью оператора `send`, указав фактическое значение параметров, например:

```
send T1 (V:=100; Teta:=rad(45));
```

Фактические значения параметров сигнала доступны только для чтения в мгновенных действиях перехода с условием срабатывания `when T1`, который принял данный сигнал, через составные имена с именем сигнала в качестве префикса, например:

```
Vx := T1.V*cos(T1.Teta);
```

При использовании регулярных структур компонентов возможно появление массивов сигналов. Для массивов сигналов имеются две предопределенные логические функции: `any` и `all`. Первая функция возвращает значение `true`, если в данный момент гибридного времени послан хотя бы один сигнал из массива, вторая функция возвращает значение `true`, если посланы все сигналы из массива.

## Автоматическое приведение типов

Автоматическое приведение типов производится при использовании в операциях или отношениях операндов различных типов в следующих случаях:

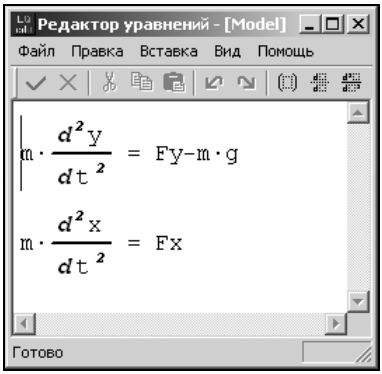
- при использовании различных целых типов операнд меньшей разрядности приводится к операнду большей разрядности;
- при использовании целого и вещественного типов целое значение приводится к вещественному типу;
- при использовании символьного и строкового типов символьное значение приводится к соответствующей строке длиной 1;
- типы `vector[1]` и `matrix[1,1]` могут трактоваться как `double`, и наоборот.

# Система уравнений

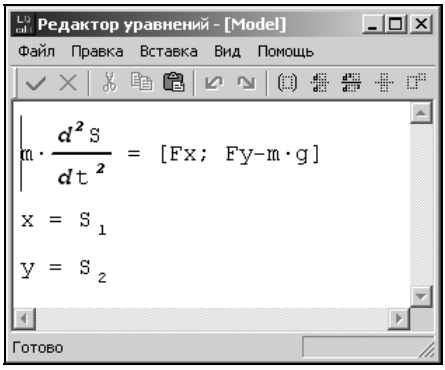
Собственное поведение непрерывного объекта задается системой дифференциально-алгебраических уравнений общего вида, т. е. допускающей использование производных порядка выше первого и не разрешенной относительно производных. Уравнения могут задаваться как в скалярной, так и в матричной форме. Современный инструмент моделирования СДС должен позволять вводить и редактировать уравнения в естественной математической форме. На рис. 2.10, *а* показано представление системы уравнений тела, брошенного под углом к горизонту, в скалярном виде, а на рис. 2.10, *б* — матричном виде на входном языке пакета MvStudium 4.

Язык моделирования должен также позволять явно указывать набор искомых переменных. Например, в пакете MvStudium транслятор сам определяет допустимый набор искомых переменных, однако в ряде случаев этот набор может быть не единственным. Кроме того, для систем уравнений с производ-

ными порядка выше первой необходимо иметь возможность явно указывать начальные значения "младших" производных (по умолчанию им присваивается нулевое начальное значение).




*а*




*б*

**Рис. 2.10.** Система уравнений на входном языке пакета MvStudium 4: *а* — в скалярном виде; *б* — матричном виде

В общем случае может потребоваться автоматическое преобразование исходных уравнений к форме, пригодной для численного решения [4]. Например, исходная система уравнений, показанная на рис. 2.10, *а*, будет преобразована к следующему виду (перед уравнением через двоеточие показана переменная, определяемая из этого уравнения при численном решении):

$$\begin{cases} y : \dfrac{dy}{dt} = y' \\ y' : \dfrac{dy'}{dt} = y'' \\ x : \dfrac{dx}{dt} = x' \\ x' : \dfrac{dx'}{dt} = x'' \\ y'' : m \cdot y'' = Fy - m \cdot g \\ x'' : m \cdot x'' = Fx \end{cases}$$

В процессе выполнения этого преобразования может возникнуть необходимость в символьном дифференцировании, символьном разрешении некоторых алгебраических уравнений, а также проверка наличия "алгебраических циклов" в наборе формул и разрыв таких циклов [4]. Например, в приведен-

ной выше системе уравнений последние два алгебраических уравнения для ускорения решения полезно было бы путем символьных преобразований разрешить относительно искомых переменных.

При моделировании гибридных систем чрезвычайно важно подсказать исполняющей системе и пользователю, что решаемая система уравнений имеет особенности, представляющие трудности для построения численного решения. Предположим, что классические динамические системы с гладкими правыми частями рассматриваются как самые "простые" для построения решения современными численными методами. Очевидно, что кусочно-непрерывные системы решать уже сложнее, так как численный метод должен самостоятельно суметь локализовать и преодолеть разрыв, например, уменьшая шаг интегрирования в его окрестности. Однако проще подсказать методу, что разрыв существует и рассматривать две задачи: до и после разрыва. В этом случае во входном языке должны быть конструкции, явно указывающие исполняющей системе на эти особые точки.

Рассмотрим, например, систему уравнений для компонента "усилитель с насыщением", статическая характеристика которого показана на рис. 2.11.

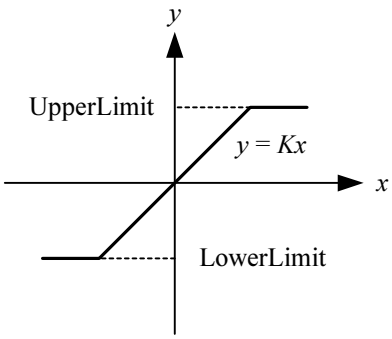



**Рис. 2.11.** График функции "усилитель с насыщением"

Простейшим решением будет задать систему уравнений в виде

$$\{Y = \text{Saturation}(Y, K, \text{LowerLimit}, \text{UpperLimit}),$$

где Saturation является алгоритмической функцией, задаваемой, например, следующим образом:

```
double Saturation(double x, double K, double LowerLimit,
                  double UpperLimit) {
  double y = K*x;
  if (y>UpperLimit) return UpperLimit;
```

```
  else if (y<LowerLimit) return LowerLimit;
  else return y;
}
```

Многие модельеры так и поступают, но это именно тот случай, когда решаемая система уравнений имеет разрывную производную. В данном случае очень удобно использовать в качестве подсказки так называемые условные уравнения:

$$\begin{cases} z = K \cdot X; \\ Y = \text{if } z > \text{UpperLimit} \text{ then } \text{UpperLimit} \\ \qquad\qquad\qquad \text{else if } z < \text{LowerLimit} \text{ then } \text{LowerLimit} \text{ else } z. \end{cases}$$

Условное уравнение отнюдь не сводится к простому переносу кода из тела алгоритмической функции в правую часть уравнения. В современных пакетах моделирования СДС исполняющая система умеет генерировать дискретное событие при переключении условных выражений с одной ветви на другую. Таким образом, скачок значения производной произойдет во "временной щели" и численное решение будет корректным.

В ряде случаев бывает удобным использовать "функциональный" стиль при составлении системы уравнений. Пусть, например, функционирование моделируемой системы задается уравнениями

$$\begin{cases} \dfrac{dy}{dt} = \text{Saturation}(x, 2, -1, 1); \\ x = A \cdot \sin(\omega \cdot t). \end{cases}$$

Конечно, эту систему уравнений можно задать структурной схемой, показанной на рис. 2.12.

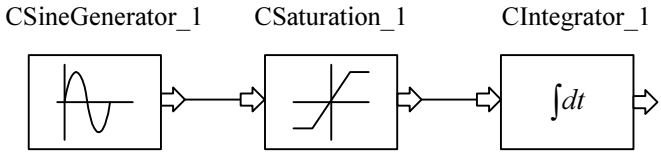



**Рис. 2.12.** Эквивалентная структурная схема

Однако это может быть неудобным, если у моделируемой системы нет естественной структуры. Поэтому язык моделирования должен позволять использовать в уравнениях некоторый набор стандартных функций с разрывами (звено с насыщением, релейное звено, зона нечувствительности, генератор "пилы", генератор импульсов и т. д., а также некоторые общематематические функции, такие как `abs`, `sign`). Но обращения к этим функциям должны ав-

томатически заменяться на экземпляры соответствующих стандартных компонентов, как показано на рис. 2.13.

```
S1: CSaturation(K=2,LowerLimit=-1,UpperLimit=1)
```

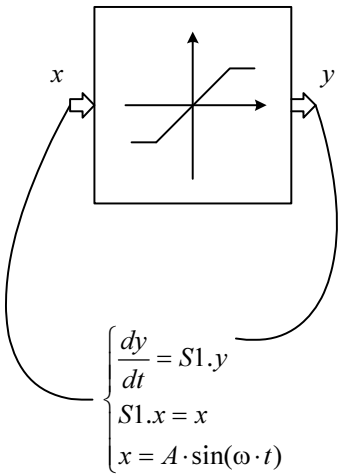



**Рис. 2.13.** Использование компонентов-функций

Часто бывает также удобно включать в систему уравнений формулы целого, булева или строкового типа. Примером может служить описание функционирования аналого-цифрового преобразователя в модели системы автоматического регулирования:

$$Y = \text{round}\left(\frac{X}{D}\right),$$

где переменная $Y$ имеет целый тип. Язык моделирования должен позволять использовать формулы (но не уравнения) "невещественного типа". Однако при анализе текущей совокупной системы уравнений всей модели в целом (такой анализ в зависимости от типа компонента может проводиться на этапе ввода уравнений, генерации кода или во время выполнения модели) необходимо выяснить, используется ли переменная, стоящая в левой части такой формулы, в каких-либо уравнениях. Если нет, то достаточно вычислить эту формулу в правильной последовательности с другими формулами. Если да, то эту формулу необходимо автоматически преобразовать в эквивалентный гибридный автомат, показанный на рис. 2.14.

В этом случае в момент изменения значения переменной будет возникать дискретное событие, в результате чего правые части обычных уравнений останутся кусочно-непрерывными.

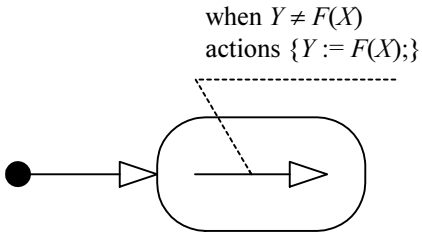



**Рис. 2.14.** Гибридный автомат, эквивалентный "невещественной" формуле

# Карта поведения

*Карта поведения* задает собственное поведение гибридного или дискретного АДО. Карта поведения является расширением "карты состояний" или "машины состояний" UML [2] в части трактовки "деятельности", приписываемой состоянию. Однако такая расширенная трактовка деятельности вынуждает, как будет показано ниже, одновременно вводить некоторые ограничения на конструкции, касающиеся чисто дискретных аспектов.

Карта поведения, как и карта состояний, представляет собой совокупность состояний и переходов. В любой момент времени только одно из состояний является текущим. В начальный момент времени существования данного объекта текущим является начальное состояние.

## Состояния

Состояние может быть обычным или особым. К *особым состояниям* относятся начальное состояние, конечное состояние и точка ветвления.

*Обычное состояние* соответствует некоторому качественному состоянию моделируемой системы и изображается прямоугольником с закругленными углами. На рис. 2.15 показана карта состояний отрывающегося маятника. В данном случае моделируемая система имеет два естественных качественных состояния:

- колебания до обрыва стержня;
- свободный полет точечной массы после обрыва стержня.

В общем случае в обычном состоянии могут быть определены входные действия, выходные действия, а также локальная деятельность (activity) (рис. 2.16). Последовательность входных действий выполняется мгновенно при каждом входе в данное состояние, а последовательность выходных действий — мгновенно при каждом выходе из данного состояния. Локальная деятельность протекает непрерывно, пока данное состояние является текущим.

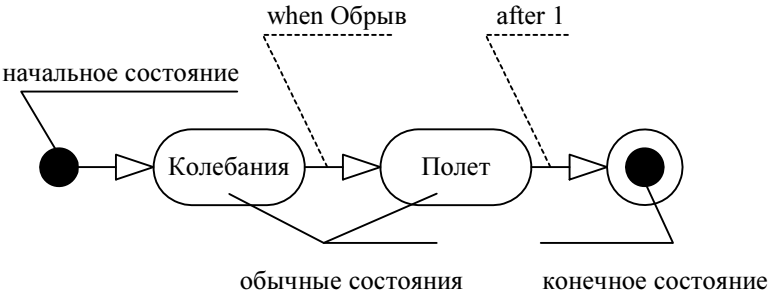



**Рис. 2.15.** Карта состояний отрывающегося маятника

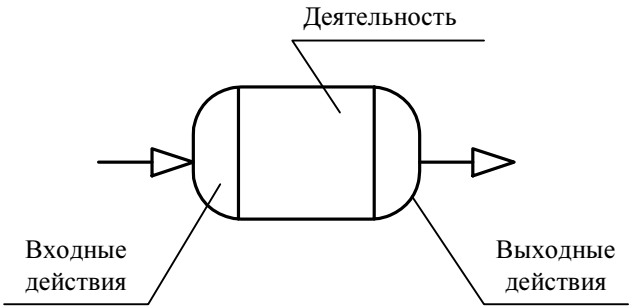



**Рис. 2.16.** Обычное состояние

*Начальное состояние* изображается черным кружком (рис. 2.17). Оно всегда становится текущим при инициализации карты поведений. В начальное состояние не может входить ни один переход. Начальное состояние не может иметь входных или выходных действий, а также локальной деятельности.

*Конечное состояние* изображается кружком с черным кружком меньшего размера в центре (рис. 2.17). Из конечного состояния не может исходить ни одного перехода. Конечное состояние не может иметь входных или выходных действий, а также локальной деятельности. Переход в конечное состояние означает, что функционирование карты поведений завершилось. Если это карта поведений модели, то это означает завершение вычислительного эксперимента.

Предопределенная функция `finalized` возвращает значение `true`, если:

- собственное поведение объекта — аргумента этой функции является картой поведений;
- текущим в этой карте поведений является конечное состояние.

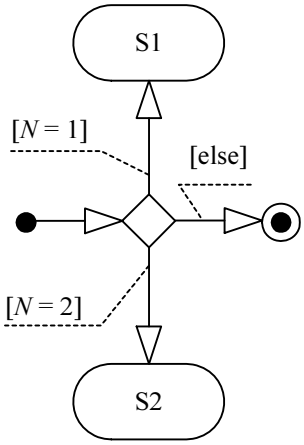



**Рис. 2.17.** Простая точка ветвления

*Точка ветвления* изображается ромбом (см. рис. 2.17). Точка ветвления не может иметь входных или выходных действий, а также локальной деятельности. Исходящие из точки ветвления переходы должны быть только нетриггерными *(см. далее)*. При попадании в точку ветвления вычисляются охраняющие условия исходящих из нее переходов и выполняется тот переход, охраняющее условие которого истинно. Охраняющие условия (включая альтернативу `else`) исходящих переходов должны охватывать все возможные варианты, то есть при попадании в точку ветвления обязательно должен сработать один из исходящих переходов. Например, в карте поведений на рис. 2.17 в момент $t = 0$ в зависимости от значения переменной $N$ происходит либо переход в обычные состояния S1 и S2, либо переход в конечное состояние.

В недетерминированных моделях весьма удобна *вероятностная точка ветвления*, которая изображается двойным ромбом (рис. 2.18). Условия исходящих из нее переходов трактуются как значение вероятности переходов, а для альтернативного перехода его вероятность предполагается равной $1 - S_P$, где $S_P$ — сумма вероятностей остальных исходящих переходов (в примере на рис. 2.18 вероятность альтернативного перехода равна $1 - p1 - p2$). На рис. 2.18 приведена карта поведений модели, в которой $N$ раз разыгрывается вероятностное ветвление и подсчитывается реальное число срабатываний трех альтернативных переходов.

В UML под *деятельностью в состоянии* понимается однократно или циклически выполняемая последовательность дискретных действий. Такая трактовка деятельности вполне достаточна для создания прототипов вычислительных систем, но недостаточна для создания моделей СДС. В карте поведений под *деятельностью в состоянии* понимается экземпляр $X$ некоторого

АДО, который динамически создается при входе в данное состояние и уничтожается при выходе из этого состояния. В моменты, когда данное состояние является текущим, собственное поведение объекта, которому принадлежит карта поведений, совпадает с поведением объекта $X$.

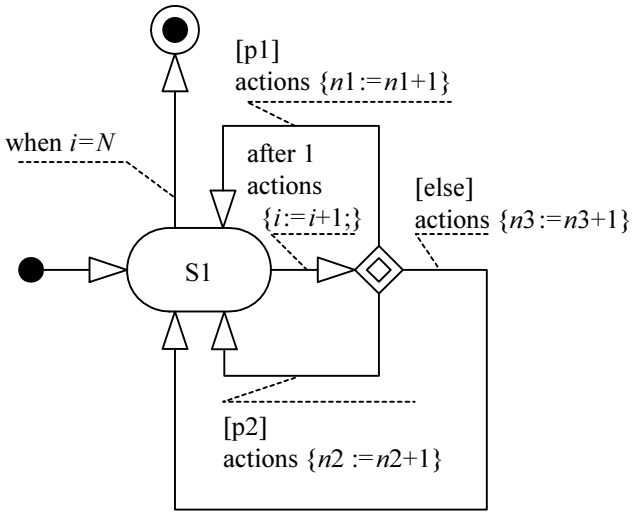



**Рис. 2.18.** Вероятностная точка ветвления

Такая трактовка деятельности и обусловила использование специального термина "карта поведения", поскольку термин "гибридная карта состояний" обычно служит для обозначения карты состояний, в которой состояниям приписываются уравнения. Кроме того, такое определение деятельности накладывает ряд ограничений на базовую конструкцию — карту состояний. Прежде всего, это связано с трактовкой подсостояний. Многоуровневая внешне карта состояний UML, по существу, является плоской одноуровневой, т. к. разрешается задавать "прямые" переходы извне непосредственно на вложенное подсостояние и наоборот, из подсостояния на состояние верхнего уровня иерархии. Иерархическая же карта поведений получается простым использованием дискретных или гибридных компонентов в качестве деятельностей. В этом случае мы имеем дело с действительно иерархической вложенностью. Очевидно, что никакие "прямые" переходы здесь невозможны, поскольку поведение инкапсулировано внутри объекта $X$. При создании экземпляра локальной карты поведений ее текущим состоянием всегда является начальное. Соответственно, невозможны и переходы в так называемое "историческое" состояние, поскольку вложенная карта поведений просто уничтожается при выходе из охватывающего состояния. Еще две конструкции классической карты состояний UML — параллельные подсостояния и

соединение/разъединение переходов — не могут использоваться в карте поведений, т. к. противоречат принципу синхронной композиции гибридных автоматов [4].

Карта поведений позволяет строить сложные поведения путем последовательной (а не параллельной, как в случае структурной схемы) композиции отдельных компонентов — объектов-деятельностей — взаимодействующих через начальные условия. Обычное состояние, которому приписана непрерывная деятельность, будем изображать толстой линией, а состояние, которому приписана дискретная или гибридная деятельность — двойной тонкой линией. Так, например, сложное поведение отрывающегося маятника может строиться как последовательная композиция двух простых непрерывных объектов — модели колебаний маятника и модели свободного двумерного движения материальной точки (рис. 2.19).

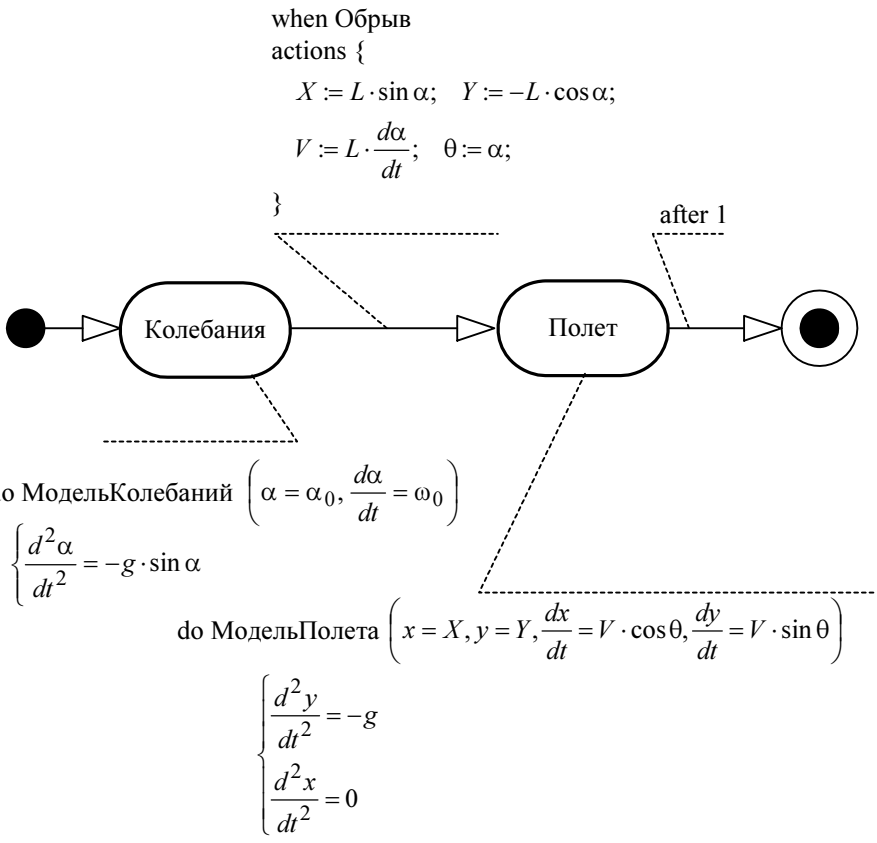



**Рис. 2.19.** Карта поведений обрывающегося маятника

В начальный момент текущим становится состояние Колебания и создается экземпляр класса МодельКолебаний с действительными значениями начальных условий. В момент отрыва в дискретных действиях перехода вычисляются декартовы координаты маятника, а также его линейная скорость и угол наклона траектории. Заметим, что экземпляр деятельности продолжает существовать до завершения действий исходящего перехода. На основе этих данных вычисляются действительные начальные значения при создании экземпляра класса МодельПолета в состоянии *Полет*.

Отметим, что карта поведений, показанная на рис. 2.19, соответствует случаю, когда деятельности являются экземплярами локальных классов и все переменные декларированы как атрибуты класса. В случае, когда классы МодельКолебаний и МодельПолета являются глобальными или импортируемыми классами, обращение к переменным деятельности возможно только через префикс состояния, например, Колебания.$\alpha$, Полет.x.

Следует также отметить, что для практического удобства полезно несколько ослабить правила видимости переменных для классов одного и того же пакета. Вполне разумным представляется доступность всех переменных локальной деятельности, а не только внешних, в условиях и действиях карты поведений: например, на рис. 2.19 в действиях перехода видима внутренняя переменная $\alpha$ и ее первая производная объекта Колебания.

## Переходы

*Переход* срабатывает, если исходное состояние перехода является текущим и выполняется некоторое условие срабатывания. При срабатывании выполняется мгновенная последовательность действий перехода, если она есть. Условие срабатывания перехода в общем случае включает в себя указание запускающего события (триггер) и охраняющее условие (guard). В карте поведений одновременно (в гибридном времени) может срабатывать только один переход. Переходы делятся на внешние и внутренние, а также триггерные и нетриггерные.

*Внешний переход* переводит систему из одного состояния в другое (возможно, то же самое). *Внутренний переход* не приводит к выходу из текущего состояния.

На рис. 2.20 показана карта поведений, в которой после срабатывания как внешнего, так и внутреннего переходов текущим все равно останется состояние S1. Предположим, что действия внешнего и внутреннего переходов одинаковы. Однако конечные результаты срабатывания этих переходов могут быть различными, поскольку при срабатывании внутреннего перехода выполняются только мгновенные действия перехода, а при срабатывании внеш-

него перехода выполняются: выходные действия состояния S1, мгновенные действия перехода, уничтожение объекта-деятельности в состоянии S1, входные действия состояния S1, создание объекта-деятельности в состоянии S1.

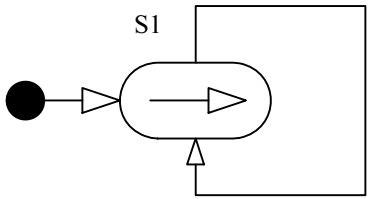



**Рис. 2.20.** Внешний и внутренний переходы

Проиллюстрируем различие между внешним и внутренним переходами на конкретном примере. Рассмотрим модель "прыгающего мячика" — класс `BBall`. Предположим, что свободное движение мячика задается непрерывным объектом класса `Flight` с системой уравнений, показанной на рис. 2.10, *а*, и при упругом соударении с горизонтальной плоскостью вертикальная составляющая скорости мгновенно изменяет знак. Это "классическая" модель, иллюстрирующая тип гибридных моделей, в которых состав переменных и набор уравнений остаются неизменными, а значения некоторых переменных могут изменяться скачками.

На рис. 2.21 показана карта поведений модели прыгающего мячика с использованием внешнего перехода. В этом случае мы имеем модель с переменным составом. При каждом отскоке текущий мячик (экземпляр класса `Flight`) исчезает и появляется новый экземпляр мячика, согласованный со старым по начальным условиям. "Время жизни" одного экземпляра мячика ограничено очередным участком свободного полета. Чтобы подчеркнуть это, в примере на компакт-диске, в окне 3D-анимации после каждого отскока мячик изменяет цвет.

На рис. 2.22 показана карта поведений модели прыгающего мячика с использованием внутреннего перехода. В этом случае один и тот же экземпляр класса `Flight` существует от начала до конца вычислительного эксперимента, а при отскоке лишь скачком меняется значение производной $\frac{dy}{dt}$.

Результаты моделирования для обеих моделей одинаковы, однако использование внутреннего перехода для данного типа гибридных моделей представляется более естественным. Заметим, что в карте поведений отрывающегося маятника (см. рис. 2.19) может быть использован только внешний переход, т. к. эта модель относится к тому типу гибридных моделей, в которых изменяется и состав переменных, и набор уравнений.

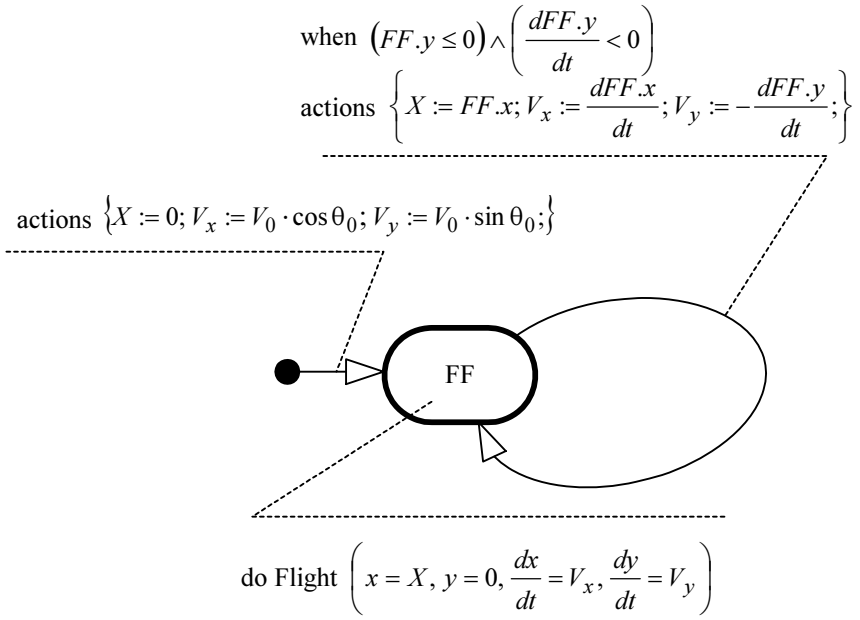



**Рис. 2.21.** Карта поведений прыгающего мячика с внешним переходом

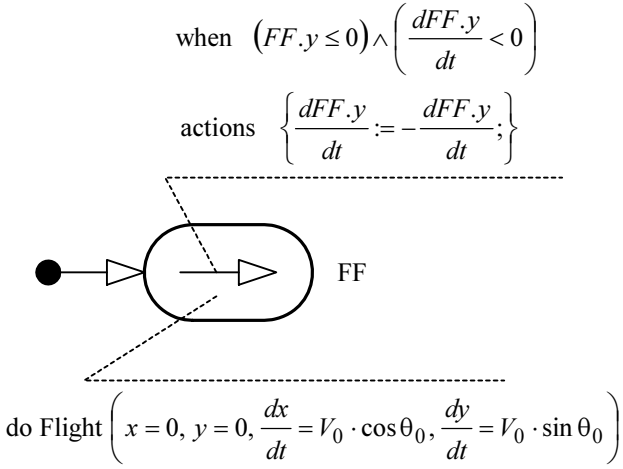



**Рис. 2.22.** Карта поведений прыгающего мячика с внутренним переходом

*Триггерный переход* срабатывает, если происходит запускающее событие и выполняется охраняющее условие (или оно отсутствует). В случае, если происходит запускающее событие, но охраняющее условие не выполняется, пе-

реход не срабатывает. Запускающее событие может задаваться тремя способами:

- `when` *<логическое_выражение>* — событие происходит, когда логическое выражение становится истинным;
- `when` *<сигнал>* — событие происходит, когда появляется указанный сигнал;
- `after` *<интервал>* — событие происходит, когда истекает указанный интервал от момента входа в текущее состояние.

В карте поведений отрывающегося маятника переход между состояниями `Колебания` и `Полет` является триггерным с запускающим событием второго типа, а переход между состоянием `Полет` и конечным состоянием — триггерным с запускающим событием третьего типа. В карте поведения прыгающего мячика переход является триггерным с запускающим событием первого типа.

*Нетриггерный переход* срабатывает немедленно при входе в исходное состояние, если выполняется охраняющее условие или оно отсутствует. Нетриггерный переход, у которого отсутствует охраняющее условие, называется *безусловным*. Обычно именно безусловный переход переводит карту поведений из начального в первое обычное состояние. Часто такой безусловный переход используется для инициализации значений переменных (см. рис. 2.21). С помощью нетриггерных переходов осуществляется выбор нового состояния в точке ветвления. Для переходов, исходящих из точки ветвления, охраняющее условие указывается в квадратных скобках, и может быть приведено условие `else`.

Использовать нетриггерный переход, исходящий из обычного состояния, имеет смысл в следующих двух случаях:

- состоянию приписана непрерывная деятельность, и нахождение в этом состоянии применяется для решения системы алгебраических уравнений с целью согласования значений переменных;
- состоянию приписана гибридная или дискретная деятельность, в этом случае исходящий нетриггерный (в частности, безусловный) переход сработает только тогда, когда карта поведений объекта-деятельности достигнет своего конечного состояния.

Использование нетриггерных переходов и точек ветвления проиллюстрируем на модели статистических испытаний. Предположим, что мы хотим провести статистический эксперимент — бросить тело $N$ раз, задавая случайные значения начальной скорости и угла бросания, и получить $N$ значений дальности падения. Пусть значения начальной скорости и угла бросания распределены по нормальному закону с математическим ожиданием и дисперсией $M_V$, $D_V$ и $M_\theta$, $D_\theta$ соответственно. Прежде всего, на основе непрерывного

класса Flight создадим гибридный класс Throw с параметрами $V_0$ и $\theta_0$, который описывает однократное бросание тела с запоминанием дальности падения в переменной L. Карта поведений этого класса показана на рис. 2.23.

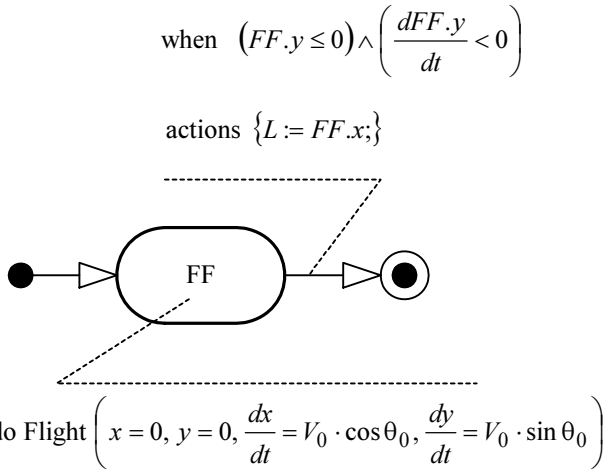



**Рис. 2.23.** Карта поведений для однократного бросания тела

Теперь на основе этого класса создадим модель статистических испытаний, карта поведений которой показана на рис. 2.24. Нетриггерные переходы из точки ветвления приводят либо в конечное состояние, когда число выполненных бросаний $n$ достигло $N$, либо в состояние $T$ в противном случае. При каждом входе в состояние $T$ число выполненных бросаний $n$ увеличивается на единицу. Безусловный переход из состояния $T$ в точку ветвления соответствует второму случаю использования безусловного перехода и сработает только при достижении конечного состояния в карте поведений объекта класса Throw, т. е. в очередной точке падения (рис. 2.23). Значение очередной дальности падения запоминается в векторе $L$ размера $N$, который полностью заполняется к моменту завершения вычислительного эксперимента.

Одновременно в гибридном времени в данном состоянии может срабатывать только один внешний переход, в противном случае результаты моделирования становятся неоднозначными или требуется произвольный выбор одного из переходов и модель становится недетерминированной. На рис. 2.25 приведен пример двух взаимодействующих гибридных объектов: объект U2 с периодичностью Tau2 посылает сигнал SignalX, а в объекте U1 по этому сигналу срабатывает переход T2, в действиях которого значение переменной $x$ уменьшается на единицу. В то же время, в самом объекте U1 с периодичностью Tau1 срабатывает переход T1, в действиях которого значение переменной $x$ увеличивается на единицу.

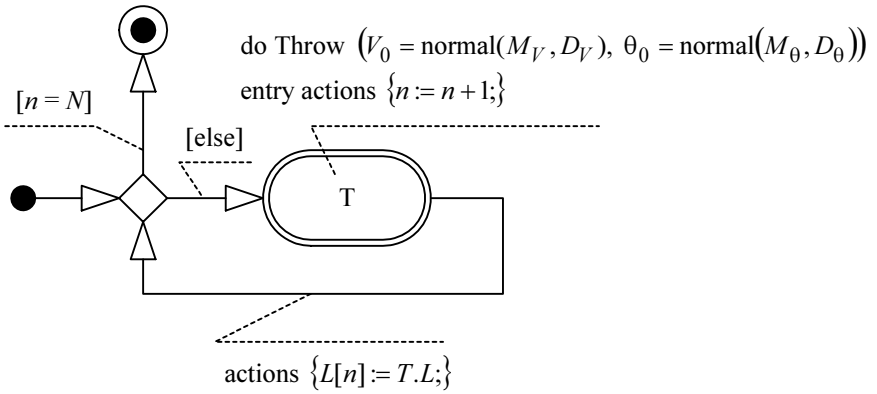



**Рис. 2.24.** Карта поведений модели статистических испытаний

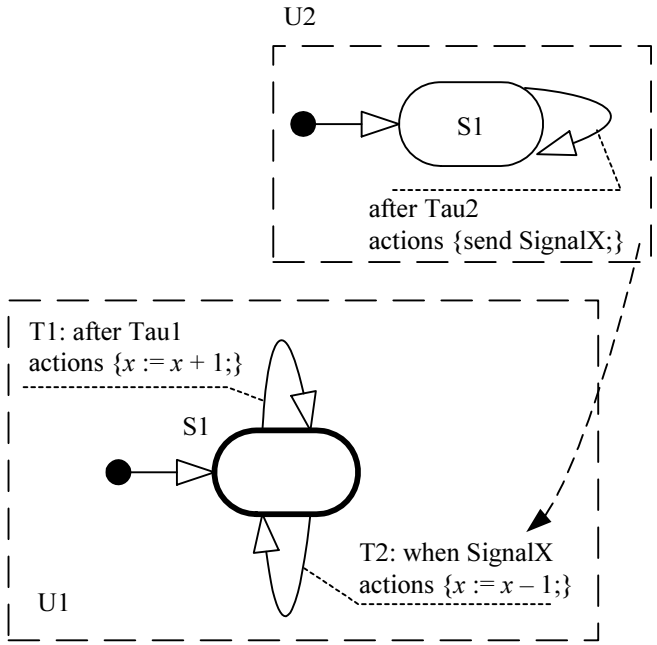



**Рис. 2.25.** Некорректные одновременные переходы

Эта модель будет прекрасно работать, пока Tau1 и Tau2 не кратны. Если же, например, они принимают значения 1 и 2, то в момент $t = 2$ в объекте U1 окажутся готовыми к срабатыванию два перехода, исходящие из одного и того же состояния. В то же время, это вполне соответствует семантике модели: в момент $t = 2$ значение переменной $x$ просто не должно измениться.

Для того чтобы отразить эту семантику корректно, необходимо модифицировать карту поведений объекта U1, как это показано на рис. 2.26. Можно оба перехода сделать внутренними (рис. 2.26, *а*). В этом случае все готовые к срабатыванию внутренние переходы будут срабатывать в одном такте дискретного времени в произвольном порядке. В этом варианте остается некоторая опасность неоднозначности результатов, которая должна устраняться на прикладном уровне. Корректным является также и вариант, показанный на рис. 2.26, *б*: переход T1 сделан внутренним, а переход T2 оставлен внешним. В этом случае сначала срабатывают все готовые к срабатыванию внутренние переходы (в данном случае один), а затем срабатывает единственный внешний переход.

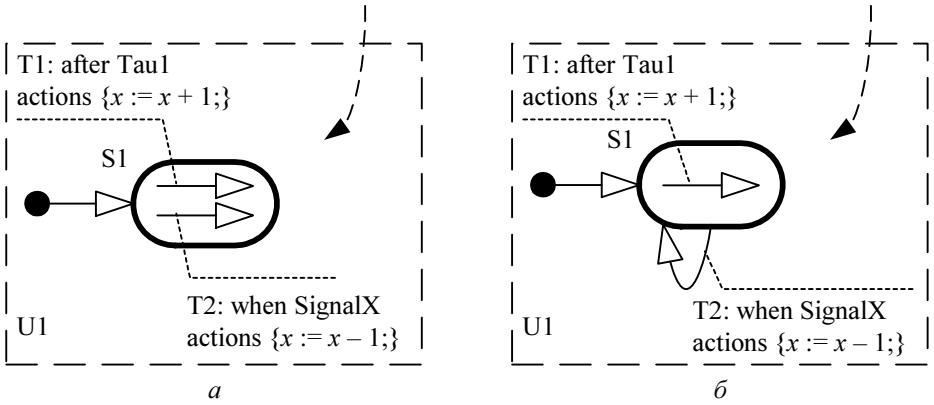



**Рис. 2.26.** Корректные одновременные переходы: *а* — оба перехода внутренние; *б* — один внутренний и один внешний переходы

Чтобы проиллюстрировать использование карты поведений для создания гибридных моделей второго типа (состав переменных сохраняется, но меняется набор уравнений), а также использование локальных классов, рассмотрим модель шарика, падающего на пружину (рис. 2.27).

В принципе, поведение такой модели можно задать и без всякой карты поведений с помощью условных уравнений:

$$\begin{cases} \dfrac{d^2y}{dt^2} = \text{if} \quad y > HS \quad \text{then} - g \quad \text{else} \quad -g + K \cdot (HS - y), \\ ys = \text{if} \quad y > HS \quad \text{then} \quad HS \quad \text{else} \quad y, \end{cases}$$

где $y$, $ys$, $HS$, $K$ — соответственно положение шарика, положение свободного конца пружины, размер пружины в распрямленном состоянии, коэффициент жесткости пружины. Однако эти уравнения можно использовать

только в предположении, что коэффициент жесткости пружины таков, что шарик никогда не ударится о горизонтальную плоскость. Если это не так, то поведение модели нужно задавать картой поведений, показанной на рис. 2.28.

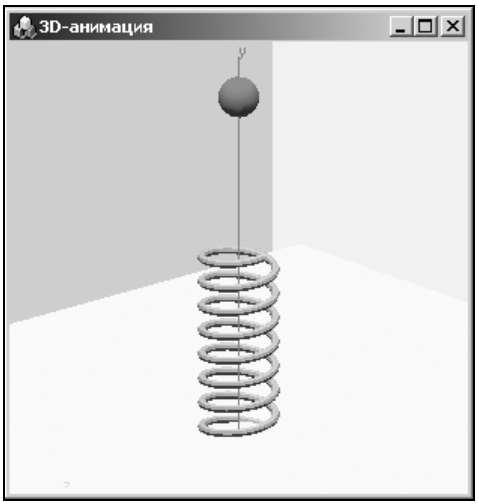



**Рис. 2.27.** Шарик, падающий на пружину

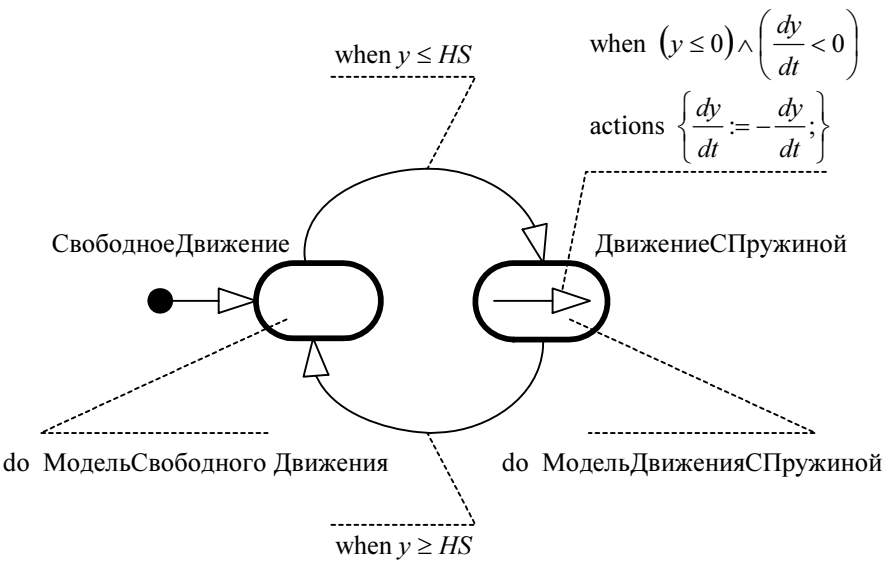



**Рис. 2.28.** Карта поведений шарика, падающего на пружину

В этой карте поведений используются локальные непрерывные классы *МодельСвободногоДвижения* с системой уравнений

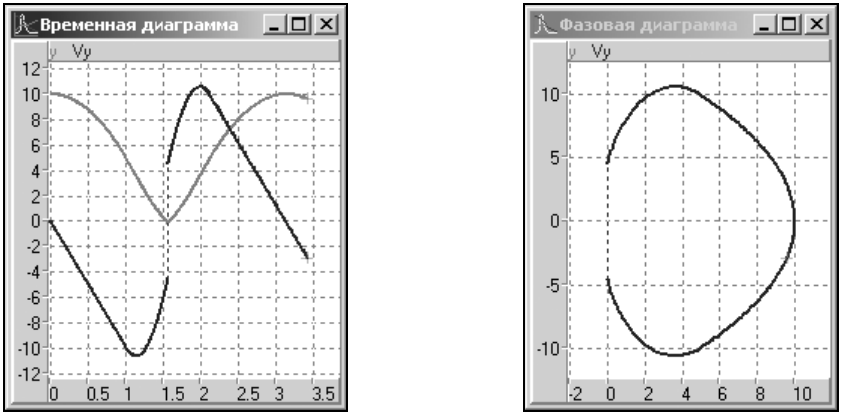
$$\begin{cases} \dfrac{d^2y}{dt^2} = -g \end{cases}$$

и *МодельДвиженияСПружиной* с системой уравнений

$$\begin{cases} \dfrac{d^2y}{dt^2} = -g + K \cdot (HS - y), \\ ys = y. \end{cases}$$

Поскольку переменные $y$, $ys$ определены в самой модели, они и их производные видимы в локальных классах и в карте поведений модели. Обратите внимание, что в данном случае нет необходимости явно указывать начальные условия для объектов-деятельностей: при переходе в другое состояние эти переменные просто сохраняют свои последние значения.

На рис. 2.29 показаны временная и фазовая диаграммы для этой модели.




*а* *б*

**Рис. 2.29.** Диаграммы для модели шарика, падающего на пружину: *а* — временная; *б* — фазовая

# Структурная схема

Структурная схема содержит описания экземпляров локальных активных динамических объектов и их связей, т. е. задает совокупность параллельно функционирующих компонентов и их взаимодействия. Взаимодействие меж-

ду локальными объектами также может осуществляться и косвенным образом — через собственное поведение (систему уравнений или карту поведений) объекта-контейнера.

## Объекты

Локальный объект может являться экземпляром класса, определенного в данном проекте или импортированного из других пакетов. Локальный объект не может являться экземпляром локального класса, т. к. в этом случае не гарантируется синхронное объединение гибридных автоматов. Визуальным образом локального объекта на структурной схеме является прямоугольник, на границах которого условно изображаются внешние переменные (рис. 2.30). На изображении объекта или рядом показывается имя объекта. На изображении объекта также отображается значок класса, если он есть. Локальные объекты изображаются внутри большого прямоугольника, соответствующему гипотетическому экземпляру класса-контейнера.

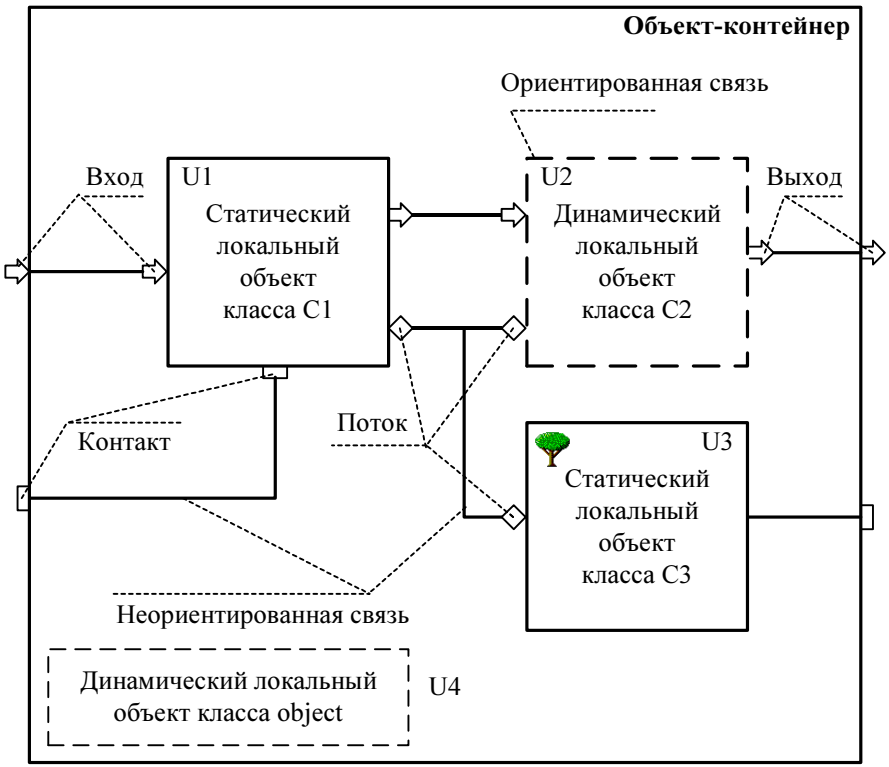



**Рис. 2.30.** Структурная схема

Локальные объекты могут быть *статическими* и *динамическими*. Статические объекты создаются вместе с экземпляром объекта-контейнера и уничтожаются вместе с ним. Динамические объекты создаются и уничтожаются явно с помощью операторов `new` и `destroy` в действиях карты поведений объекта контейнера. При создании экземпляра объекта-контейнера указатель на динамический объект остается пустым, при уничтожении объекта-контейнера динамические объекты также уничтожаются. Динамический объект обозначается пунктирным прямоугольником.

При описании динамического объекта может быть указан его базовый класс или интерфейс. В этом случае динамическому объекту может быть присвоен экземпляр любого класса — потомка указанного базового или любого класса, реализующего указанный интерфейс *(см. разд. "Полиморфизм" далее в этой главе)*. В случае указания базового класса или интерфейса динамического объекта в описании объекта контейнера могут использоваться его внешние переменные. Если же базовый класс или интерфейс не указан, то динамический объект может быть использован лишь как аргумент функций `time` и `finalized`. Например, в структурной схеме на рис. 2.30 динамический объект U2 является типизированным (указан базовый класс C2) и потому может участвовать в связях, в то время как динамический объект U4 является нетипизированным и потому не может участвовать в связях.

Следует отметить, что с точки зрения языка UML локальные объекты являются атрибутами объекта-контейнера с семантикой указателя: статические объекты соответствуют атрибутам с начальным значением, а динамические объекты — атрибутам с начальным значением `null`. Структурная схема является лишь удобным визуальным образом, наглядно отражающим состав моделируемой системы и взаимодействия между объектами.

## Связи

*Связь* или *соединение* является указанием стандартного взаимодействия, имеющего свой графический образ на структурной схеме. Связь изображается линией, соединяющей две внешних переменных (см. рис. 2.30). Соответственно типу соединяемых переменных связь является направленной или ненаправленной. Совокупности связей соответствует набор уравнений и формул, неявно добавляемый к совокупной системе уравнений модели. Ненаправленные связи могут соединять только переменные типов `double`, `vector` или `matrix`, а также `record` с полями перечисленных типов. В последнем случае уравнения связей формируются по каждому полю записи. Направленные связи могут соединять переменные любых типов. Связи, в которых участвуют внешние переменные несуществующего динамического объекта, не являются активными. Например, связи, в которых участвует динамический объект U2

(см. рис. 2.30), не являются активными до тех пор, пока в действиях объекта-контейнера не будет выполнен оператор

```
U2:=new C2;
```

Эти связи вновь перестанут быть активными после выполнения оператора

```
destroy U2;
```

В процессе функционирования объекта-контейнера могут создаваться новые связи и уничтожаться существующие с помощью специальных операторов `connect` и `disconnect`.

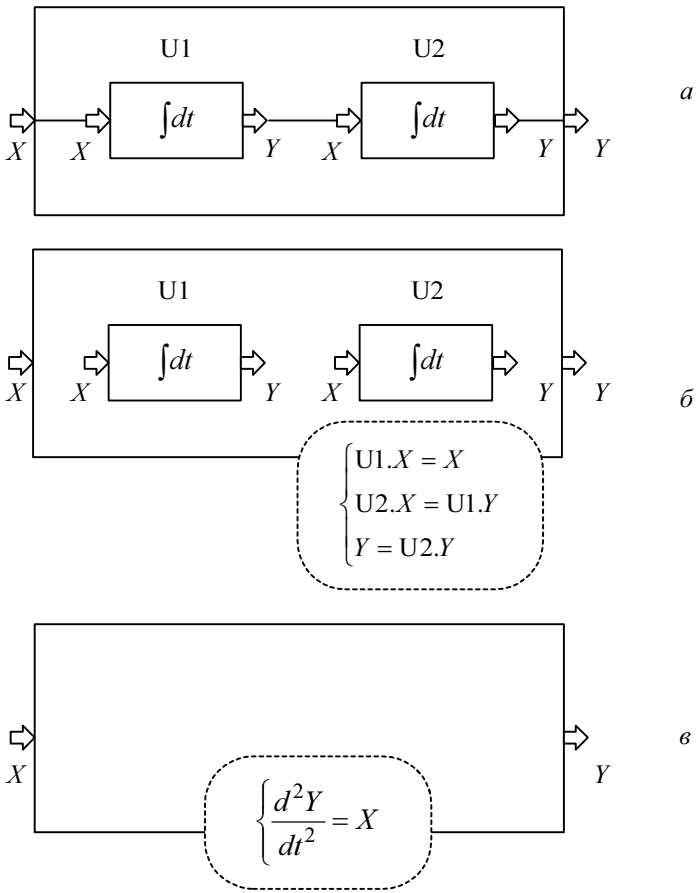



**Рис. 2.31.** Структурная схема и собственное поведение объекта-контейнера:
*а* — поведение задается совокупностью локальных объектов и связей;
*б* — поведение частично задается локальными объектами;
*в* — поведение задается уравнениями объекта-контейнера

Следует отметить, что связи являются лишь удобным визуальным образом, отражающим уравнения взаимодействия объектов. Те же уравнения могут быть явно указаны в описании собственного поведения объекта-контейнера. На рис. 2.31 показаны три варианта задания одного и того же поведения объекта контейнера: в случае рис. 2.31, *а* все поведение задается совокупностью локальных объектов и связей, в случае рис. 2.31, *б* поведение частично задается локальными объектами, а взаимодействие этих локальных объектов задается собственными уравнениями объекта-контейнера, в случае рис. 2.31, *в* все поведение задается уравнениями объекта-контейнера.

Карта поведений объекта-контейнера может использоваться в качестве алгоритма вычислительного эксперимента. На рис. 2.32 показана структурная схема модели простейшей системы автоматического регулирования с пропорциональным регулятором.

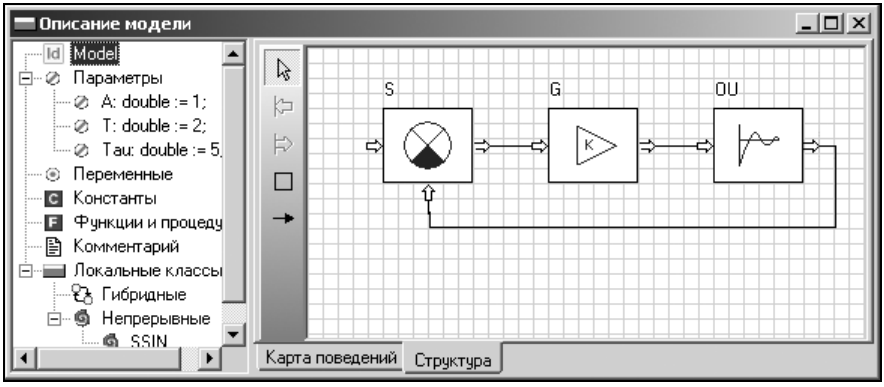



**Рис. 2.32.** Модель простейшей системы регулирования

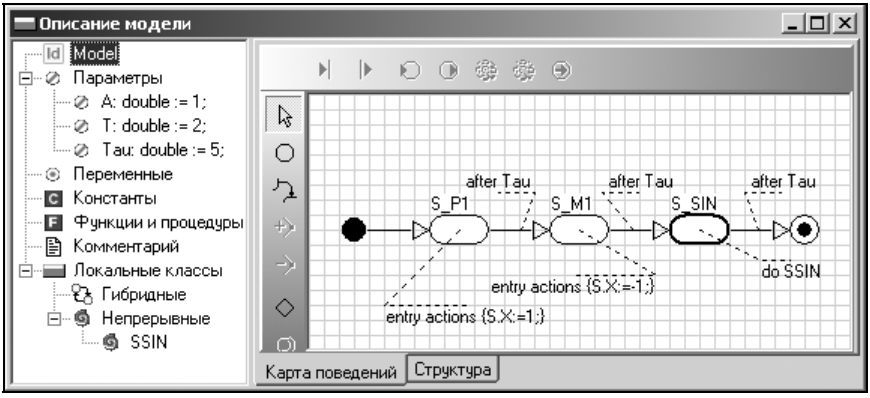



**Рис. 2.33.** Карта поведений — план вычислительного эксперимента

На рис. 2.33 представлена карта поведений этой модели, задающая алгоритм вычислительного эксперимента: сначала на вход требуемого значения подается положительная ступенька, потом отрицательная и затем синусоидальный сигнал. После подачи тестовых сигналов в течение некоторого интервала изучается реакция модели.

Результаты вычислительного эксперимента показаны на рис. 2.34.

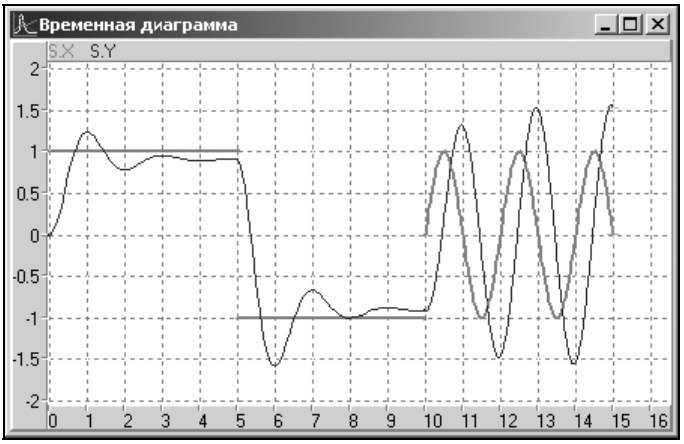



**Рис. 2.34.** Результаты вычислительного эксперимента

## Регулярные структуры

Иногда возникает необходимость в использовании регулярных структур локальных объектов — *мультиобъектов*. Это могут быть мультиобъекты постоянной размерности — массивы объектов или мультиобъекты переменной, заранее неизвестной размерности — списки объектов. Мультиобъекты аналогичны обычным массивам и спискам, однако в качестве типа элемента у них указывается класс или интерфейс. Мультиобъект изображается на структурной схеме как прямоугольник с тенью (массив) или прямоугольник с закругленными углами с тенью — список. В описании поведения класса-контейнера доступен как весь массив, так и его элементы. Следует отметить, что внешние переменные мультиобъекта будут иметь регулярный тип, первое измерение которого соответствует размерности мультиобъекта. Функция `time` для мультиобъекта будет иметь тип `array of double`, а функция `finalized` — тип `array of boolean` соответствующей размерности.

Для иллюстрации использования мультиобъектов рассмотрим две модификации модели статистических испытаний, рассмотренной выше. В карте поведений на рис. 2.24 предполагается строго последовательное функционирова-

ние объектов: объект — материальная точка создается, бросается со случайными начальными значениями скорости и угла наклона траектории, отслеживается точка падения, объект уничтожается и так $N$ раз. Однако те же результаты могут быть получены и еще двумя способами:

- все $N$ тел бросаются одновременно, эксперимент заканчивается, когда зафиксировано $N$-е падение;
- $N$ тел бросаются последовательно с интервалом $\tau$, эксперимент заканчивается, когда зафиксировано $N$-е падение.

В первом случае удобно использовать массив объектов класса `Throw`, создав локальные объекты с помощью итеративного литерала.

$$\text{TA} : \text{array}\,[1..N]\ \text{of Throw} :=$$
$$:= [\text{for } i \text{ in } 1..N \mid \text{new Throw}\,(V_0 = \text{normal}(M_V, D_V), \theta_0 = \text{normal}(M_\theta, D_\theta))].$$

Далее достаточно подождать, пока все тела не упадут (после падения объекты класса `Throw` переходят в конечное состояние), заполнить вектор дальностей падения и завершить эксперимент.

Во втором случае необходимо использовать список объектов класса `Throw`

```
TL: list of Throw;
```

заполняя его в действиях карты поведений объекта-контейнера (рис. 2.35).

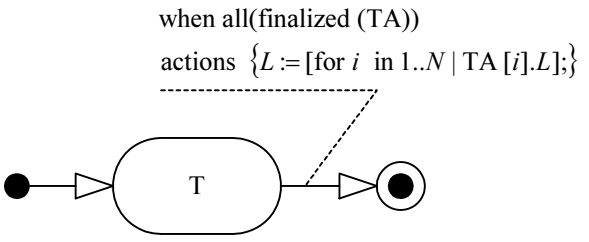



**Рис. 2.35.** Карта поведений модели для вычислительного эксперимента с массивом тел

В действиях внутреннего перехода, срабатывающего с интервалом $\tau$, пока число брошенных тел $m$ не достигнет числа измерений $N$, создается очередной экземпляр класса `Throw` со случайными значениями начальной скорости и начального угла. При падении любого тела из списка `TL` срабатывает другой внутренний переход (рис. 2.36), в действиях которого число упавших тел $n$ увеличивается на единицу, определяется индекс первого упавшего тела, запоминается его дальность падения, объект разрушается и удаляется из списка. Эксперимент завершается, когда все тела упадут.

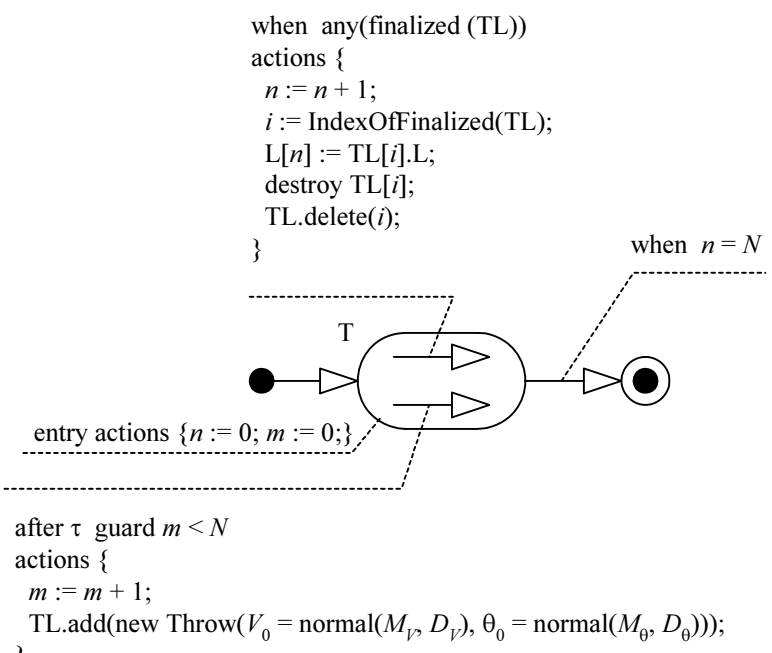



**Рис. 2.36.** Карта поведений модели для вычислительного эксперимента со списком тел

В действиях перехода используется алгоритмическая функция:

```
function IndexOfFinalized(L: list of object) return integer is
  X: object;
begin
  for i in low(L) to high(L) loop
    X:=L[i];
    if Finalized(X) then
      return i;
    end if;
  end loop;
  return -1;
end IndexOfFinalized;
```

Отметим, что если случайно в некоторый момент непрерывного времени упадет несколько тел одновременно, то в этой временной щели внутренний переход сработает нужное число раз.

# Наследование классов

С помощью наследования можно обеспечивать повторное использование ранее разработанных и отлаженных моделей. Если класс C2 объявляется прямым потомком класса C1 (рис. 2.37), то класс C2 наследует все элементы класса C1: переменные, процедуры и функции, локальные классы, карту поведений, систему уравнений и структурную схему. Применительно к отношению C1↔C2 класс C1 будем далее называть базовым классом или суперклассом, а C2 — производным классом или подклассом. Отношение наследования транзитивно: если класс C3 является производным от класса, C2, то класс C3 является производным от класса C1 и наследует все его элементы.

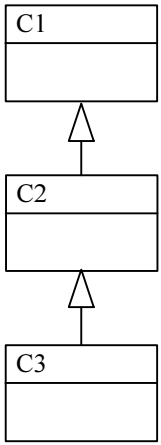



**Рис. 2.37.** Наследование классов

Все изменения, вносимые в класс C1 и в любой из его предков, будут автоматически отражаться на классе C2. В языке моделирования СДС допустимо только одиночное наследование: любой производный класс может иметь только один базовый.

Целью наследования является расширение и/или модификация описания базового класса. Это можно осуществить с помощью добавления новых элементов описания и переопределения унаследованных элементов описания. Никакие унаследованные элементы не могут быть удалены. Все активные динамические объекты являются потомками предопределенного класса `ActiveDynamicObject` (для краткости `object`).

## Добавление новых элементов описания

В языке моделирования СДС новые элементы описания не могут иметь имена, совпадающие с элементами описания базового класса. Это противоречит традиционному для объектно-ориентированных языков программирования правилу, согласно которому новая переменная с тем же именем, что и унаследованная, скрывает унаследованную переменную в описании производного класса. Однако в описании активного динамического объекта, в отличие от пассивного объекта, оба элемента (например, переменные) могут использоваться одновременно — в системе уравнений, к которой добавлены новые уравнения или в карте поведений, к которой добавлены новые состояния и переходы. Такая ситуация совершенно недопустима, т. к. может привести к неверному пониманию пользователя и провоцировать ошибки.

В производном классе могут быть определены новые переменные, процедуры и функции, а также новые локальные классы. В систему уравнений могут быть добавлены новые уравнения, в карту поведений — состояния и переходы, а в структурную схему — новые локальные объекты и новые связи.

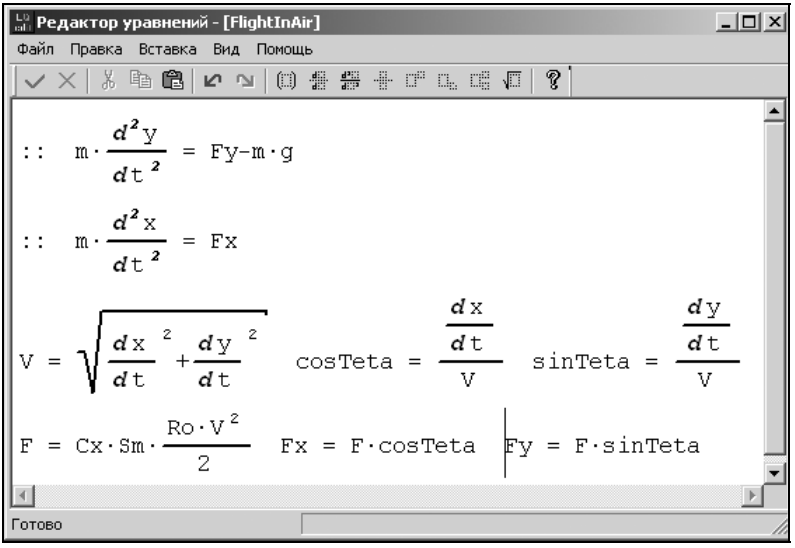



**Рис. 2.38.** Расширение унаследованной системы уравнений

В качестве примера рассмотрим класс `FlightInAir` (рис. 2.38), являющийся потомком класса `Flight` и описывающий полет в воздухе. В этом классе добавлены новые переменные, а в систему уравнений — новые уравнения, определяющие силы сопротивления воздуха (унаследованные уравнения маркированы двойным двоеточием).

## Переопределение унаследованных элементов

В описании производного класса может быть переопределено определение унаследованного элемента. Переопределение является основным механизмом модификации базового класса. Переопределяемый элемент определяется по совпадению имени или прямым указанием в визуальном редакторе: "вот этот переход".

- **Переопределение переменных.**

  При переопределении переменной можно изменить только ее стереотип. Таким способом можно, например, параметр сделать входной переменной или состоянием. Возможность изменения стереотипа позволяет гибко адаптировать существующие классы к особенностям конкретных моделей.

- **Переопределение процедур и функций.**

  При переопределении процедуры или функции должна быть полностью сохранена унаследованная сигнатура (список параметров и тип результата). Для дополнительных параметров должны быть обязательно указаны значения, принимаемые по умолчанию. Тело процедуры или функции изменяется полностью (хотя редактор в качестве начального приближения предлагает унаследованный текст, всякая связь с телом соответствующей функции в базовом классе утрачивается).

- **Переопределение локального класса.**

  При переопределении локального класса в него могут быть добавлены новые элементы и переопределены унаследованные.

- **Переопределение системы уравнений.**

  Унаследованную систему уравнений можно переопределить только целиком. Переопределение происходит при попытке редактировать унаследованные уравнения.

- **Переопределение элементов карты поведений.**

  При переопределении карты состояний можно:

  - заменить входные/выходные действия в состоянии;
  - заменить деятельность в состоянии;
  - заменить условие срабатывания и/или охраняющее условие перехода;
  - заменить последовательность действий в переходе;
  - изменить графическое изображение состояния или перехода.

  В качестве примера рассмотрим модификацию модели прыгающего мячика для случая неупругого удара. Для этого необходимо в производном классе переопределить действия в переходе на $\frac{dy}{dt} = -k\frac{dy}{dt}$, где $0 < k < 1$.

- **Переопределение элементов структурной схемы.**

  При переопределении структурной схемы можно заменить локальный объект на любой другой, в котором существуют внешние переменные, соответствующие по идентификатору и типу значения внешним переменным заменяемого объекта (т. е. использовать полиморфизм "по интерфейсу"). Например, можно модифицировать схему системы автоматического регулирования путем замены P-регулятора на PD-регулятор, имеющий те же самые внешние переменные.

## Полиморфизм

В языке моделирования СДС используется как "традиционный" полиморфизм объектов, так и полиморфизм "по интерфейсу". "Традиционный" полиморфизм заключается в том, что вместо декларированного объекта определенного класса C фактически могут использоваться экземпляры любых производных от C классов. Такой полиморфизм можно назвать полиморфизмом "по наследованию". Полиморфизм "по интерфейсу" означает, что объект можно заменить на любой другой, в котором существуют внешние переменные, соответствующие по идентификатору и типу значения внешним переменным заменяемого объекта. При этом замещающий объект может не находиться с замещаемым в одной цепочке наследования.

Выражение "могут использоваться вместо" конкретно означают, что:

- экземпляр производного класса может передаваться как действительное значение параметра, типизированного для базового класса (в примере функции `IndexOfFinalized` список `list of Throw` передается вместо `list of object`);
- экземпляр производного класса может присваиваться рабочей переменной базового класса (в примере функции `IndexOfFinalized` элемент списка класса `Throw` присваивается рабочей переменной класса `object`);
- экземпляр производного класса может присваиваться динамическому объекту, типизированного для базового класса;
- экземпляр производного класса может являться элементом мультиобъекта (массива или списка), типизированного для базового класса.

Аналогичные соотношения справедливы между объектом и соответствующими конструкциями, типизированными как некоторый интерфейс, если класс объекта реализует этот интерфейс.

## Параметризованные классы

*Параметризованный класс* в языке моделирования СДС является некоторым аналогом класса-шаблона (template) в объектно-ориентированных языках

программирования. В параметризованном классе некоторые из его параметров имеют тип `class` или `interface`. В определении параметризованного класса в качестве значения параметра по умолчанию может быть указан конкретный класс или интерфейс. При создании экземпляра параметризованного класса в качестве действительного параметра может быть указан другой конкретный класс или интерфейс, отличный от используемого по умолчанию.

Рассмотрим в качестве примера параметризацию класса "прыгающий мячик" `BBall` (вариант с внутренним переходом). В карте поведений этого класса (см. рис. 2.22) в качестве локальной деятельности в состоянии `FF` указан экземпляр конкретного класса `Fliht`, который определяет двумерное движение тела в постоянном по величине и направлению плоском гравитационном поле без учета сопротивления воздуха. В то же время алгоритм функционирования "прыгающего мячика" гораздо более общий, он будет успешно работать и при движении в воздухе, и при движении в переменном центральном гравитационном поле. Можно построить цепочку классов, производных от `BBall` — "прыгающий мячик в воздухе", "прыгающий мячик в центральном гравитационном поле" и т. д., — в которых будет соответственно переопределяться деятельность в состоянии `FF`. Однако гораздо более изящным решением будет параметризация класса `BBall`. Для этого введем специальный параметр `FFClass`, задающий класс локальной деятельности в состоянии `FF`, и определим эту деятельность так:

$$\text{do} \quad \text{FFClass}\left( x = 0,\ y = 0,\ \frac{dx}{dt} = V_0 \cdot \cos\theta_0,\ \frac{dy}{dt} = V_0 \cdot \sin\theta_0 \right).$$

Тип параметра можно задать как "любой потомок класса `Flight`"

```
parameter FFClass: class of Flight := Flight;
```

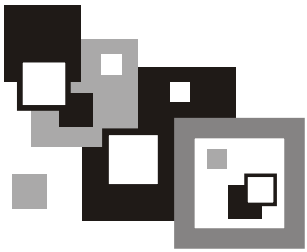
а можно задать как интерфейс "двумерное движение"

```
parameter FFClass: MovingXY := Flight;
```

где

```
interface MovingXY is
  x: double;
  y: double;
end MovingXY;
```

# ГЛАВА 3



# Моделирование гибридных систем и объектно-ориентированный подход в различных пакетах

> "Конечно, в идеале пакет должен быть оснащен таким инструментарием, который на основании общих требований к будущей программе сам автоматически создает, собирает ее из модулей и при этом проводит оптимизацию либо по числу модулей, либо по времени работы программы, либо по каким-либо другим критериям".
>
> *Попов Ю. П., Самарский А. А. [13]*

Объектно-ориентированное моделирование еще только интенсивно развивается, однако уже создано довольно много инструментальных средств, позволяющих моделировать сложные динамические системы, используя объектно-ориентированный подход. Встав на сторону максималиста, можно сказать, что моделирование сложных систем вообще немыслимо без объектно-ориентированного подхода, и использовался он всегда, — пусть даже неосознанно. В этой главе будут рассмотрены различные подходы к объектно-ориентированному моделированию и сделана попытка описать их с единой точки зрения. Возможно, это поможет осознать, какие элементы новой технологии "вечны", а какие — "приходящие".

Общим требованием объектно-ориентированного подхода (ООП) является наличие у среды моделирования языка, поддерживающего классы и позволяющего строить новые классы, используя инкапсуляцию, наследование и полиморфизм. Частным требованием, относящимся к моделированию сложных динамических систем, является поддержка возможности моделировать системы, состоящие из большого числа компонентов, меняющих свое поведение и структуру во времени. С этих позиций и попробуем оценивать различные пакеты моделирования. Особое внимание будем уделять возможности пакетов моделировать гибридные системы.

# Моделирование гибридных систем в инструментальных средствах для "больших" ЭВМ

Рассмотрим несколько подходов, типичных для "ранних" инструментов компьютерного моделирования (до начала 1990-х гг.). Их характерными чертами являются: отсутствие графического языка (только текстовое представление описания модели), способ построения входного языка как проблемно-ориентированного расширения существующего языка программирования (например, Fortran или Simula-67), ориентированность на пакетный режим работы "больших" компьютеров (таких как IBM-370 или БЭСМ-6).

Несмотря на кажущуюся архаичность, в этих инструментах уже присутствуют легко узнаваемые черты современных подходов к компьютерному моделированию сложных динамических систем.

## Язык SLAM II

Язык имитационного моделирования SLAM II является одним из последних и самых мощных языков моделирования, построенных как расширение языка Fortran. Язык вобрал в себя лучшие решения более ранних разработок, таких как GPSS, SIMSCRIPT, GASP IV. О наличии в языке объектно-ориентированного подхода не может быть и речи, зато он включает в себя средства для моделирования "дискретных, непрерывных и непрерывно-дискретных систем" (непрерывных и дискретных динамических и гибридных систем). Термин "непрерывно-дискретные системы", как предшественник термина "гибридные системы", широко использовался в конце прошлого столетия. В "непрерывно-дискретных системах" дискретное время заменяется непрерывным, сеточные функции искусственно превращаются в кусочно-постоянные, но понятие гибридного времени не вводится. Этот подход к моделированию гибридных систем принят и во многих современных системах моделирования.

Для моделирования дискретных систем в языке предусмотрены конструкции, ориентированные на системы массового обслуживания и использующие традиционные для имитационного моделирования понятия транзакции, очереди, ресурса. Язык позволяет описывать и "дискретно-событийные" системы, представляющие собой дискретные параллельные процессы, развивающиеся в непрерывном времени (вместо сеточных используются кусочно-постоянные функции) и взаимодействующие между собой через сигналы, соответствующие произошедшим в каждом отдельном процессе событиям. В языке различаются "временные события" и "события-состояния", которые могут планироваться пользователем. "Временное событие" ("Time-event" в терминах

UML) наступает по истечении указанного промежутка модельного времени. "Событие-состояние" соответствует достижению выбранной переменной заданного порогового значения, причем можно указать направление пересечения порога ("отрицательное" — от значений, превышающих порог, к значениям, меньших его, и противоположное ему "положительное") и точность достижения порога по значению переменной. "Событие-состояние" можно найти и в современном пакете Simulink. Пользователь может связать с событием любого типа "обработчик" — последовательность действий, которая выполняется в модельном времени мгновенно (иными словами, принять и обработать сигнал о наступившем событии). С помощью мгновенных действий можно изменять значения переменных модели, а также планировать новые события.

Поведение динамических систем с непрерывным временем можно задавать в виде систем обыкновенных дифференциальных уравнений первого порядка в форме Коши, а также формул. Транслятор не сортирует формулы, поэтому пользователь должен записать их сам в правильном, вычислимом порядке. Численное интегрирование осуществляется программной реализацией метода Рунге—Кутты в форме Фельберга с автоматическим контролем точности. С непрерывными переменными можно связывать "события-состояния", и таким образом непрерывная составляющая модели может порождать дискретные события. Обработчик событий может менять значения непрерывных переменных скачками, а исполняющая система SLAM II начинает процесс численного интегрирования (автоматического выбора шага интегрирования для новых начальных условий и новых правых частей) заново после каждого дискретного события. В обработчике события могут также изменяться значения специальных переменных-переключателей, в зависимости от которых значения правых частей дифференциальных уравнений и формул могут вычисляться по различным алгоритмам. Таким образом, поведение систем становится событийно-управляемым. Как уже отмечалось в *томе I* [4], этот прием описания гибридных систем требует введения дополнительных переменных и чрезвычайно усложняет восприятие отдельных систем, скрытых за этим общим, включающим все виды правых частей, описанием длительного поведения. "Непрерывная" и "дискретная" составляющая модели могут влиять друг на друга через дискретные события, обработчики которых изменяют значения общих переменных.

Если не принимать во внимание архаический способ описания модели с использованием расширения языка Fortran, то можно сказать, что язык SLAM II предоставляет достаточно мощные средства для моделирования сложных динамических систем. Он позволяет скачками менять значения непрерывных переменных, менять форму уравнений относительно вектора переменных состояния заданной структуры, и делается это корректно с точки зрения чис-

ленного решения. Изменение структуры вектора переменных состояния (в частности изменение размерности системы уравнений) задать невозможно.

Использование в языке SLAM II переключателей для изменения алгоритмов вычисления правых частей уравнений, а также невозможность описать взаимосвязи между обработчиками дискретных событий таят в себе источники различных ошибок. Опыт показывает, что при разработке программ в аналогичном стиле (например, при программировании оконных сообщений MS Windows) разработчики делают очень много ошибок.

## Язык НЕДИС

> "Под моделированием в дальнейшем мы будем понимать такое представление программными средствами реальных систем, процессов, явлений в ЭВМ, при котором вычислительная машина воспроизводит, имитирует процесс функционирования реального объекта".
>
> *Глушков В. М., Гусев В. В., Марьянович Т. П., Сахнюк М. А. [12]*

"Моделируемая система представляется как совокупность объектов, переменного состава, а ее функционирование — как параллельное течение "деятельностей", входящих в ее состав объектов. Под "деятельностью" или процессом понимается конечная или счетная цепочка событий, разделенных конечными промежутками времени".

Под непрерывно-дискретной системой авторы языка моделирования НЕДИС понимали "конечную совокупность (переменного состава) взаимодействующих объектов. Взаимодействие объектов может быть дискретным и непрерывным. Дискретное взаимодействие — это такое мгновенное изменение в системе, как:

- появление нового объекта или уничтожение старого;
- изменение состояния.

Непрерывное взаимодействие понимается как реализация установленных функциональных зависимостей между состояниями объектов, определяемых их параметрами. В общем случае функциональные зависимости сохраняются лишь на конечном интервале времени".

Язык моделирования непрерывно-дискретных систем НЕДИС является расширением объектно-ориентированного языка Simula-67. В НЕДИС язык исполняющей системы совпадает с входным языком (языком пользователя). В исполняющей системе переопределяется стандартный класс `SIMULATION` языка Simula-67 для того, чтобы иметь возможность моделировать непрерывную составляющую поведения, в свою очередь пользователь может переопределить класс `INTGRL` для того, чтобы использовать собственный численный метод интегрирования.

Дискретное поведение описывается в виде последовательности дискретных событий. Дискретное событие связывается с фактом мгновенного исполнения заданной последовательности операторов. Таким образом, текстуальное описание последовательного дискретного процесса представляет собой последовательности операторов, разделенные специальными операторами синхронизации. Операторы синхронизации позволяют связать очередное дискретное событие с заданным моментом непрерывного модельного времени или с выполнением некоторого условия. Кроме того, одни процессы могут активизировать или "пассивизировать" (термин авторов языка) другие. Описание процесса задается в виде класса. В результате обращения к конструктору класса с набором действительных параметров возникает конкретный процесс — экземпляр класса.

В промежутках между событиями моделируемая система ведет себя как динамическая система с непрерывным временем и описывается системой обыкновенных дифференциальных уравнений первого порядка, разрешенных относительно производных ("глобальной" системой уравнений). Глобальная система может быть и пустой. В глобальную систему можно добавить новое уравнение с помощью вызова конструктора предопределенного класса `INTGRL` и убрать из нее при вызове деструктора. В действительных параметрах конструктора класса `INTGRL` указывается начальное значение интегрируемой переменной, а также ссылка на функцию, определяющую правую часть уравнения. Переменные состояния, определяемые дифференциальными уравнениями, называются *непрерывными*. В общем случае, любое событие в процессе функционирования непрерывно-дискретной системы формирует некоторую непрерывную систему, определяющую закон изменения непрерывных переменных на ненулевом интервале непрерывного времени до наступления очередного события. Изменение непрерывных переменных может вызвать очередное дискретное событие, при выполнении заданных соотношений (условий). Используя возможности синхронизации последовательных процессов и операторы ветвления алгоритмов, можно задать практически любую логику изменения непрерывного поведения.

Интересной особенностью языка является то, что с помощью алгоритма главного процесса модели можно не только получать отдельные фазовые траектории, но и строить параметрические зависимости и т. п. Таким образом, алгоритм главного процесса по существу является планом вычислительного эксперимента.

Теоретически с помощью языка НЕДИС можно создавать гибридные модели, в том числе и с переменной размерностью фазового вектора. Однако это достигается за счет серьезных ограничений на форму представления дифференциальных уравнений, описывающих поведение непрерывного объекта, и при этом приходится использовать языковые конструкции очень низкого уровня.

Пользователю практически предлагается вручную обращаться к входам исполняющей системы пакета. Понять текстовое описание даже очень простой модели чисто непрерывной системы автоматического регулирования практически невозможно.

# Гибридные модели в современных инструментах моделирования

Современные инструменты моделирования сложных динамических систем можно разделить на три группы:

- инструменты "блочного моделирования";
- инструменты "физического моделирования";
- инструменты, использующие формализм гибридного автомата.

Все современные среды визуального моделирования позволяют моделировать динамические системы с дискретным и непрерывным временем и гибридные системы.

## Моделирование гибридных систем в пакете Simulink ("блочное моделирование")

Термин *"блочное моделирование"* имеет, как минимум, два значения.

В первом случае его употребляют, когда речь заходит о блоках с направленными связями. В пакете Simulink с каждым блоком связаны три вида переменных — входы $u$ (сохраняем обозначения справочного руководства), выходы $y$ и переменные состояния $x$. Каждая из указанных переменных может быть как вектором, так и скалярной переменной. Если переменные состояния в блоке отсутствуют, он превращается автоматически из блока "вход — выход — состояние" в блок "вход — выход". Каждому блоку соответствуют уравнения

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_d \end{bmatrix};$$

$$\mathbf{y} = \mathbf{f}_{out}(t, \mathbf{x}, \mathbf{u});$$

$$\mathbf{x}_d = \mathbf{f}_{update}(t, \mathbf{x}, \mathbf{u});$$

$$\frac{d\mathbf{x}_c}{dt} = \mathbf{f}_{derivate}(t, \mathbf{x}, \mathbf{u});$$

где:

- $\mathbf{x} = \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_d \end{bmatrix}$ — вектор переменных состояния, включающих в себя непрерывные и дискретные переменные состояния;
- $\mathbf{y} = \mathbf{f}_{out}(t, \mathbf{x}, \mathbf{u})$ — вектор выходных переменных, вычисляемый на основании входов и переменных состояния;
- $\mathbf{x}_d = \mathbf{f}_{update}(t, \mathbf{x}, \mathbf{u})$ — дискретные переменные состояния, вычисляемые в заданных точках дискретного времени;
- $\frac{d\mathbf{x}_c}{dt} = \mathbf{f}_{derivate}(t, \mathbf{x}, \mathbf{u})$ — дифференциальные уравнения для непрерывных переменных состояния.

Как и в любой среде, предназначенной для проектирования многокомпонентных моделей, графическое описание каждого блока превращается исполняющей системой в математическое, в данном случае представленными выше уравнениями. Затем отдельные уравнения блоков объединяются в совокупную систему в соответствии с заданной функциональной схемой.

Во втором смысле термин "блочное моделирование" употребляют, когда хотят подчеркнуть, что блок-схема компонента простой структуры является низкоуровневым графическим представлением решаемых уравнений, восходящим еще к эпохе аналоговых машин. Она чрезвычайно трудна для восприятия человеком и этим резко отличается от мощных графических языков пакетов Mathcad или Mathematica, близких к привычному математическому языку. В пакетах Simulink, VisSim, MBTУ блоки являются направленными и предназначены для описания уравнений. Критика "блочного моделирования" связана в основном с этим низкоуровневым языком.

Язык пакета позволяет строить иерархические модели, в основе которых лежат динамические системы с непрерывным и дискретным временем, и составленные из непрерывных и дискретных блоков соответственно, как это следует из математической модели блоков. Пакет позволяет моделировать и гибридные системы. Для описания гибридных систем используются непрерывные и дискретные блоки одновременно. Пользователь может создавать и свои блоки, не являющиеся композицией предопределенных блоков, но для этого вынужден писать их с помощью процедурных языков — языка пакета Matlab, C, Fortran, Ada (см. библиотеку "User-Defined Functions").

В пакете используются две модели времени — дискретное и непрерывное время. Каждый элементарный дискретный блок может устанавливать свой постоянный шаг дискретного времени (параметр "Sample Time").

Параметр "Sample Time" может принимать:

- положительные значения, что соответствует выбранному шагу дискретного времени;
- нулевое значение, в этом случае речь идет о непрерывном времени;
- значение –1, когда речь идет о событийном времени и событийно-управляемых блоках.

Для синхронизации дискретных времен отдельных блоков выбирается наибольший общий делитель для заданного множества шагов и принимается за единицу фундаментального дискретного времени. Использование фундаментального дискретного времени с постоянным шагом может приводить к большим вычислительным затратам, поэтому допускаются вычисления с переменным шагом дискретного времени. В первом случае говорят о дискретных "решателях" с постоянным шагом, а во втором — с переменным. Для двух блоков с шагами 0.5 и 0.75 шаг фундаментального дискретного времени будет равен 0.25, если пользователь выбирает решатель с постоянным шагом. Если выбран решатель с переменным шагом, то вычисления будут проводиться в точках {0, 0.5, 075}.

При описании дискретного времени отдельного блока можно задать также и начало отсчета "локального" времени блока, не совпадающее с началом отсчета глобального модельного времени. В общем случае описание дискретного времени два параметра $[T_s, T_0]$ — шаг и начало отсчета, и таким образом $t_n = n \cdot T_s + |T_0|$. Дискретное время с нулевым шагом дискретизации $[0, 0]$ рассматривается как непрерывное время.

Последовательность отсчетов непрерывного времени, в которых производятся вычисления, в основном определяется шагами, необходимыми для численного интегрирования дифференциальных уравнений с заданной точностью.

Если блок не имеет пользовательского диалога, описывающего локальное время блока, то считается, что блок наследует все свойства времени блоков, поставляющих ему информацию на входы или присоединенных к его выходам. На блок-схеме могут быть представлены блоки с разным шагом дискретизации, исполняющая система автоматически синхронизирует их. Таким образом, Simulink различает локальные времена дискретных и непрерывных блоков и глобальное модельное время.

Для описания непрерывных компонентов модели используются стандартные блоки, содержащиеся в библиотеке `Continuous`: `Derivative` — дифференциатор, `Integrator` — интегратор, `State-space` — система "вход — выход — состояние", передаточные функции, задержки различного вида (рис. 3.1).

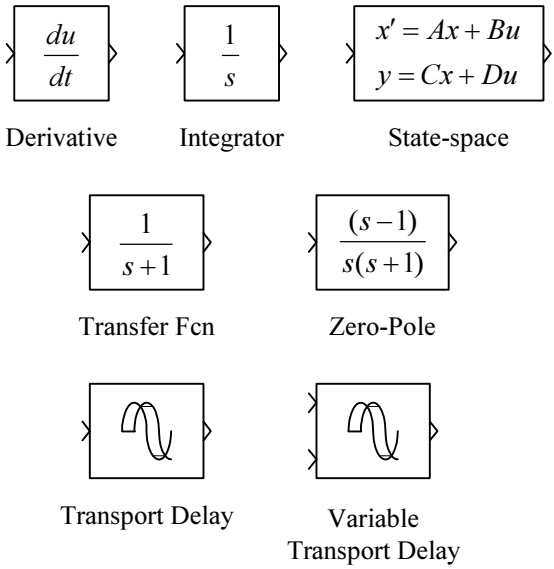


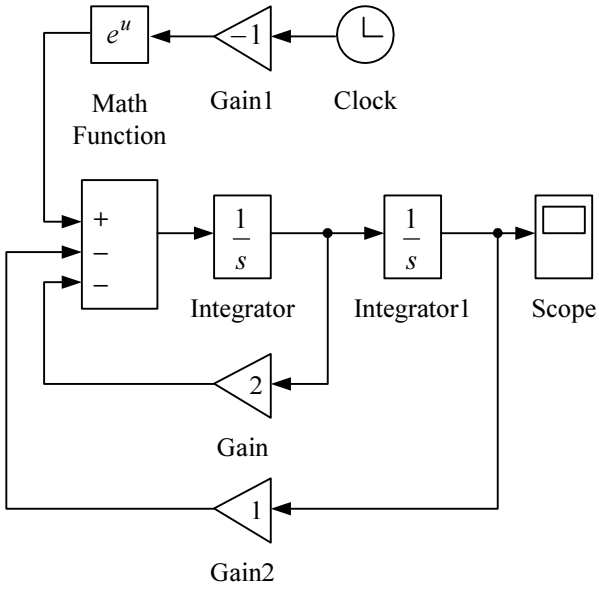

**Рис. 3.1.** Блоки для построения динамических систем с непрерывным временем

С их помощью и с помощью компонентов библиотеки `Math Operations`, содержащей блоки, которые соответствуют стандартным математическим операциям и элементарным функциям, можно собрать блок-схему, соответствующую системе обыкновенных дифференциальных уравнений в форме Коши или формулам. Пример описания дифференциального уравнения $\frac{d^2x}{dt^2} + 2\frac{dx}{dt} + x = e^{-t}$ с нулевыми начальными условиями в виде блок-схемы представлен на рис. 3.2.

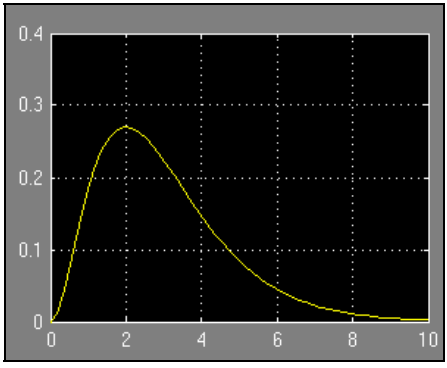
Для задания дискретных компонентов модели используются стандартные блоки, содержащиеся в библиотеке `Discrete`. К их числу относятся: `Unit Delay` — задержка на один такт, `Discrete-Time Integrator` — дискретный интегратор (с его помощью можно решать дифференциальные уравнения явным и неявным методами Эйлера или методом трапеций), `Discrete State-space` — дискретная система "вход — выход — состояние", передаточные функции (рис. 3.3).

Выходы дискретных блоков обычно являются кусочно-постоянными функциями времени, что позволяет соединять их с непрерывными блоками. Исключение составляют блоки, вырабатывающие сигналы (напомним, что мы под сигналом понимаем булеву функцию, принимающую значение "истина" при наступлении события, в то время как в пакете Simulink под сигналом

понимается любая информация, поступающая на информационный вход блока).




*а*




*б*

**Рис. 3.2.** Графическое представление динамической системы с непрерывным временем: *а* — блок-схема; *б* — результат моделирования

Пример блок-схемы, описывающей разностное уравнение второго порядка $z_{n+2} + z_{n+1} + \frac{1}{4} z_n = e^{t_n}, t_{n+1} = t_n + h$ на постоянной сетке с шагом $h = 0.1$ с нулевыми начальными условиями, представлен на рис. 3.4.

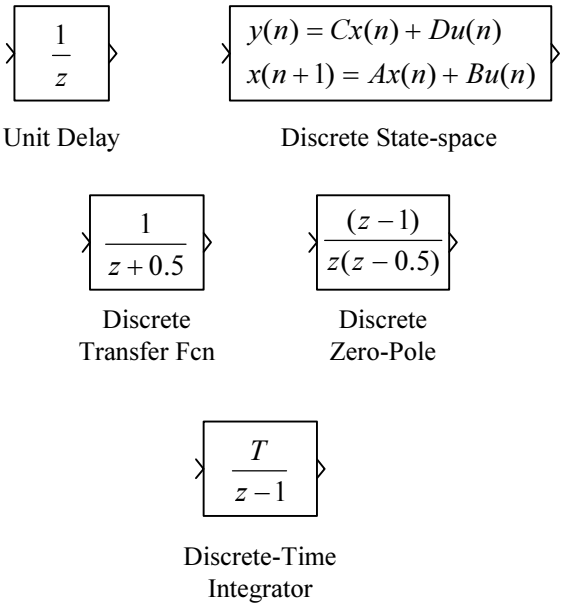



**Рис. 3.3.** Блоки для построения моделей с дискретным временем

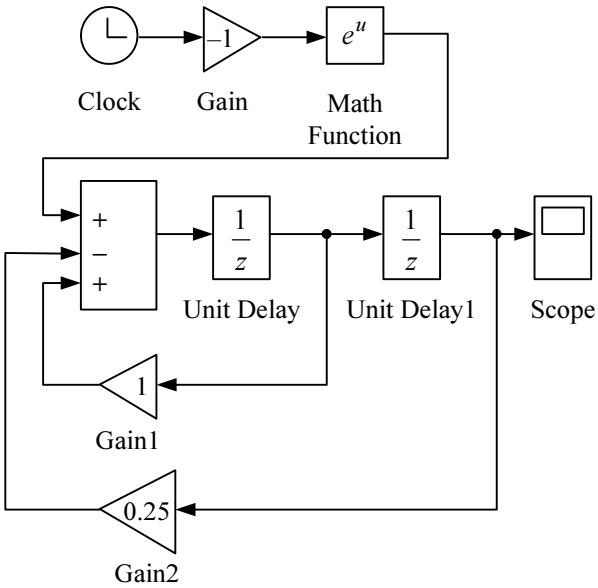
В библиотеках `Discontinuities` (разрывные функции) и `Sources` (источники "сигналов") содержится достаточно богатый набор стандартных "гибридных" блоков. К ним относятся: `Saturation` — звено с насыщением, `Dead Zone` — зона нечувствительности, `Relay` — реле, `Backlash` — петля гистерезиса, `Pulse Generator` — генератор прямоугольных импульсов, `Repeating Sequence` — генератор пилообразного "сигнала" (рис. 3.5).

Наличие таких блоков позволяет говорить о возможности моделировать гибридные системы. Исполняющая система пакета Simulink может автоматически локализовать точки изменения наклона кусочно-линейных функций и точки разрыва первого рода для блоков, представленных на рис 3.5, и некоторых других (у этих блоков есть возможность включать и выключать режим поиска перечисленных "особых" точек с помощью флага **Zero crossing**).

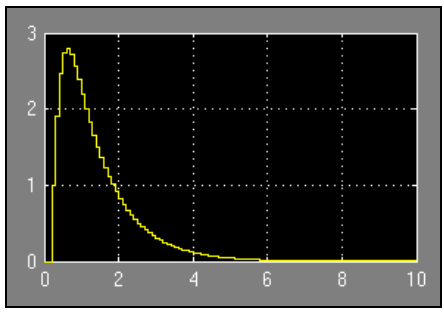
Для моделирования гибридных систем можно использовать различные блоки. Если моделируемая система может быть описана примитивным гибридным автоматом, то стоит ограничиться стандартным блоком `Integrator` (как "непрерывным", так и "дискретным") и использовать дополнительные входы — сброса интегратора (`Trigger`) и повторной инициализации значения интегрируемой переменной ( $x_0$ ). Если размерность системы фиксирована, и при наступлении события меняется только вид правой части, то можно использовать блоки `Switch` и `Switch case` и др., содержащие условные выражения. Блок `Switch`, например, при смене знака управляющего входа меняет выходной

сигнал, полагая его равным значению первого или второго входов. На рис. 3.6 представлена схема, представляющая примитивный гибридный автомат, который соответствует модели вертикально падающего мячика на абсолютно упругую плоскость (упрощенный вариант прыгающего мячика). Эта схема реализована с помощью дополнительных входов и выходов в блоке `Integrator`. Уравнения модели имеют вид

$$\frac{d^2y}{dt^2} = -9.8,\ y(0) = 10,\ y'(0) = 0\,.$$




*а*




*б*

**Рис. 3.4.** Пример блок-схемы для решения разностного уравнения:
*а* — блок-схема; *б* — результат

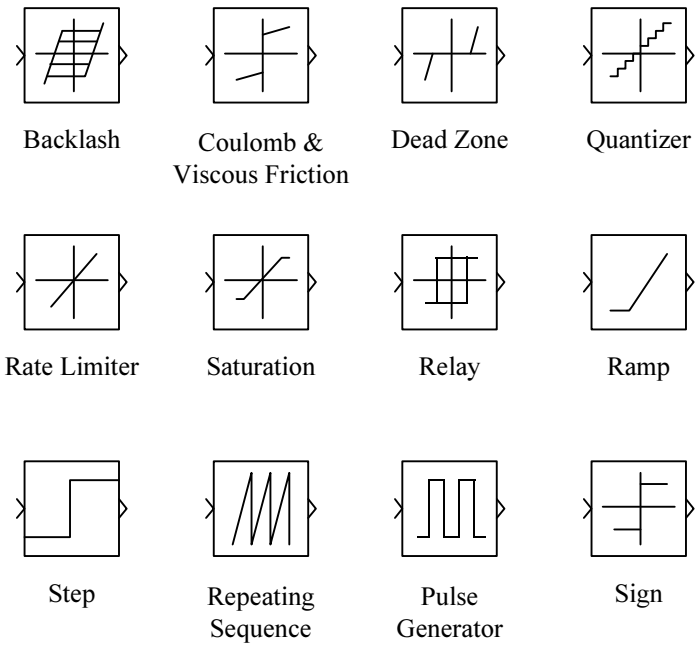
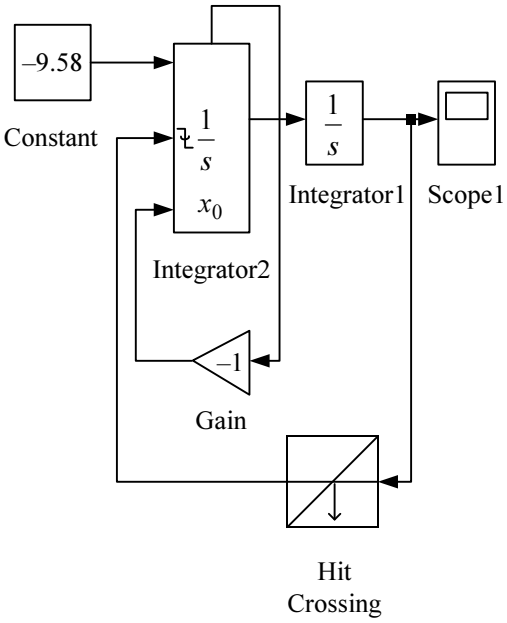


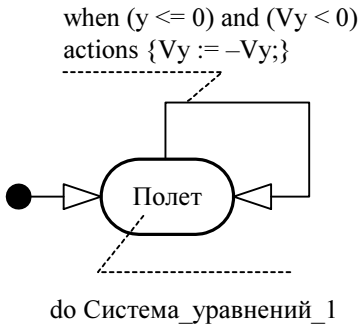

**Рис. 3.5.** Примеры блоков для описания кусочно-линейных функций и функций с разрывами первого рода

На блок-схеме представлены два интегратора (рис. 3.6, *а*). Первый имеет три входа и два выхода. На первый вход подается функция, имеющая постоянное значение. Второй вход, называемый триггерным, ожидает сигнала, указывающего на то, что следует остановить интегрирование и продолжить его с другими начальными условиями. Новые начальные условия необходимо подать на третий вход интегратора. Сигнал (булева функция) вырабатывается блоком `Hit Crossing` в момент, когда шарик коснется земли при пересечении линии падения "сверху вниз" (hit crossing detection — falling). Новые начальные условия попадают в интегратор с его второго дополнительного выхода, после чего меняется их знак, и, таким образом, новое значение скорости становится положительным. Эквивалентный этой схеме гибридный автомат представлен на рис. 3.6, *б*, а на рис. 3.6, *в* — результат интегрирования.

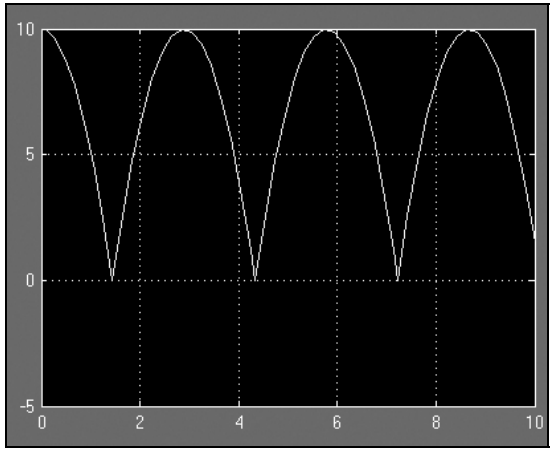
Необходимость дополнительного выхода объясняется тем, что блок `Integrator` с одним входом и одним выходом с точки зрения пакета является блоком "вход — выход", и если вход новых начальных условий подать значение с "обычного" выхода интегратора, то получившаяся схема будет эквивалентна уравнению. Результат неверно сконструированной модели представлен на рис. 3.7.




*а*




*б*



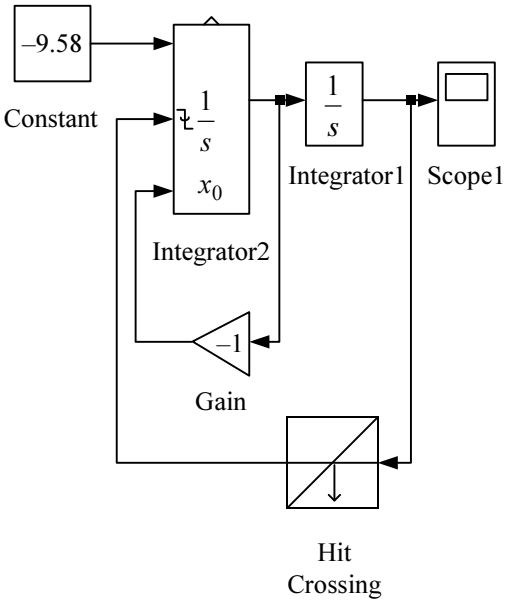

*в*

**Рис. 3.6.** Модель прыгающего мячика: *а* — блок-схема; *б* — гибридный автомат; *в* — результат интегрирования

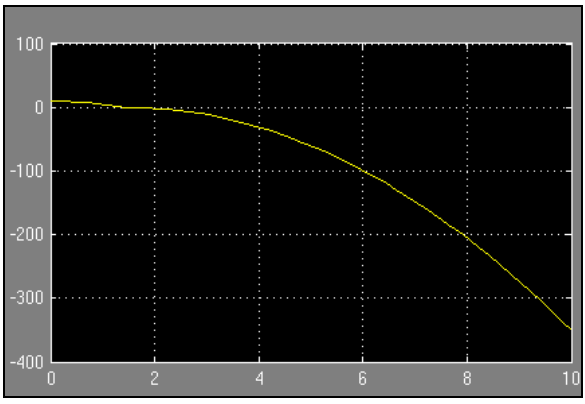
В пакете Simulink легко создавать иерархические блок-схемы (рис. 3.8). Можно сказать, что все подсистемы наследуются от блока `Subsystem`, в котором есть один выход, напрямую соединенный с единственным входом. Вход

и выход могут быть либо единственной переменной, либо векторами. Внутреннее устройство блока можно переопределять, включая в него новые подсистемы.

На рис. 3.9 представлена модель прыгающего мячика в виде иерархической блок-схемы.




*а*




*б*

**Рис. 3.7.** Неверно составленная модель прыгающего мячика:
*а* — блок-схема; *б* — результат интегрирования

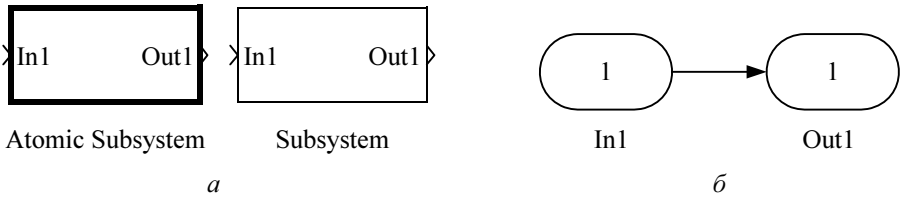



**Рис. 3.8.** Иерархическая блок-схема: *а* — простейшая подсистема; *б* — ее наполнение

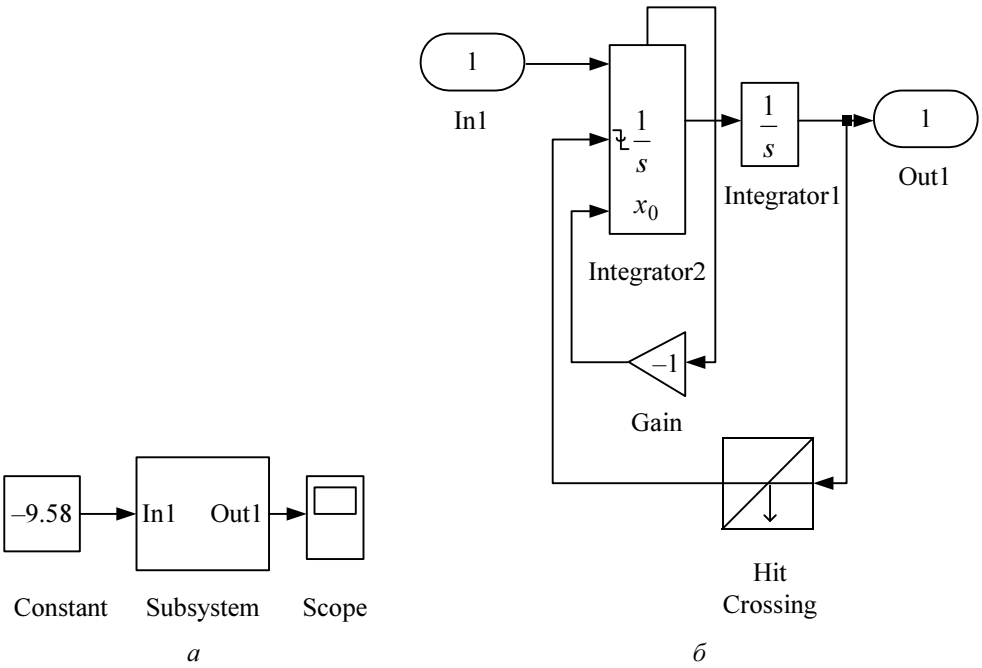



**Рис. 3.9.** Модель прыгающего мячика, представленная:
*а* — в виде иерархической блок-схемы с блоком `Subsystem` в качестве подсистемы;
б — полная блок-схема

Два специальных блока `Trigger` и `Enable`, встраиваемых в любую подсистему (рис. 3.10), позволяют включать и выключать подсистему в зависимости от поступающих на них сигналов и создавать модели с блоками переменной структуры (рис. 3.11).

На рис. 3.11, *б* показан результат работы блока с встроенным выключателем (стереотип `Enable`), у которого предусмотрен специальный управляющий вход. Блок работает так, как определено его блок-схемой (рис. 3.11, *а*), если значение этого входа положительно, и останавливает свою работу в противном случае. В данном случае, выход повторяет вход, как на рис. 3.8, и мы ви-

дим только "положительную полуволну функции $\sin(t)$. Когда блок выключен, его выходное значение доступно соединенным с ним блокам, и в данном случае соответствует значению выходного сигнала в момент выключения.

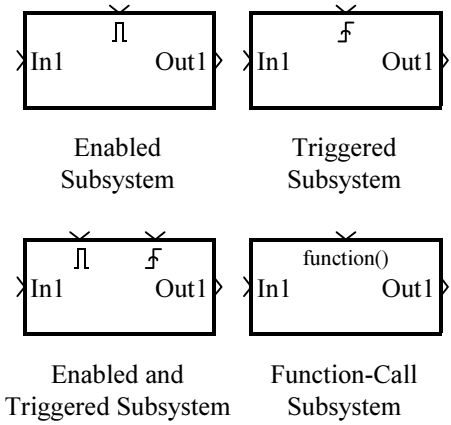
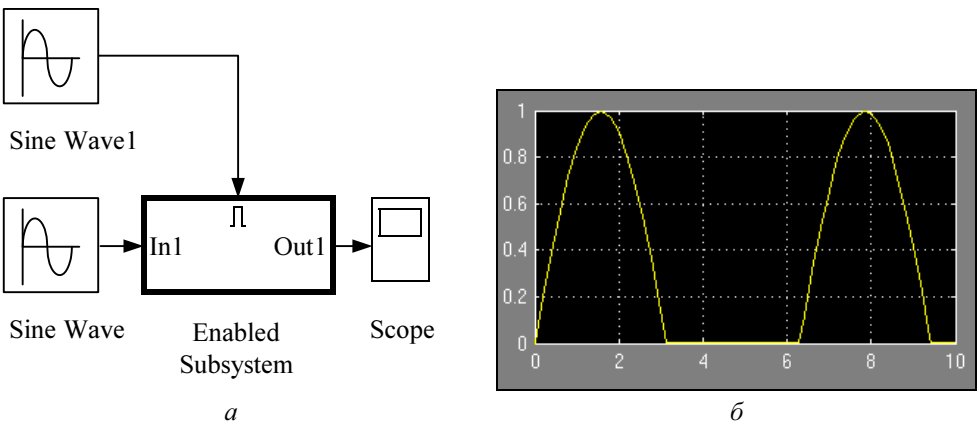



**Рис. 3.10.** Событийно-управляемые блоки

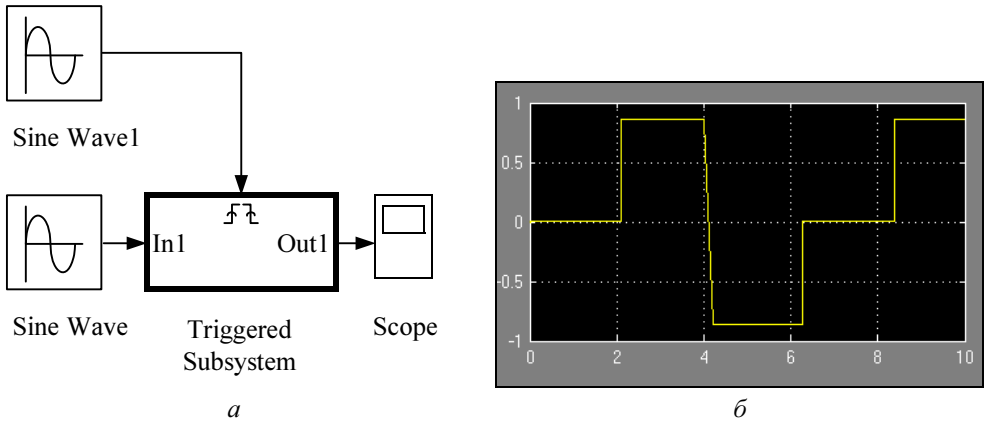



*а* *б*

**Рис. 3.11.** Пример блока со встроенным выключателем `Enable`:
*а* — блок-схема; *б* — результат работы

На рис. 3.12 показан блок с выключателем `Trigger`. Такой блок включается только в момент, когда происходит событие определенного типа, и тут же выключается, а на его выходе может сохраняться результат работы блока. В нашем примере на его информационный вход подается синусоида, имеющая частоту, равную 1, а на управляющий — синусоида с частотой в три раза

больше. Событием является пересечение синусоидой нуля в любом направлении.

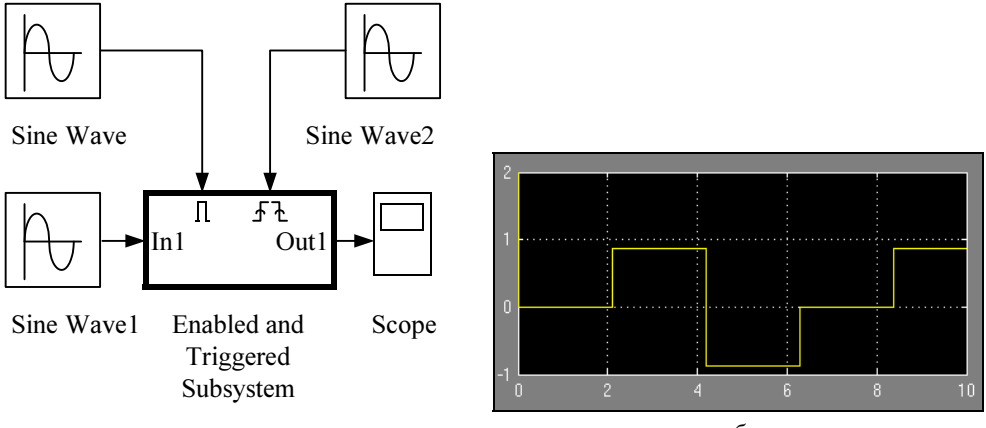



*а* *б*

**Рис. 3.12.** Пример блока со встроенным выключателем `Trigger`:
*а* — блок-схема; *б* — результат работы

В блок `Subsystem` можно поместить оба выключателя — и `Enable`, и `Trigger`. Роль событий играют события выключателя `Trigger`, а роль сторожевого условия — условие выключателя `Enable`.

На рис. 3.13 показан результат работы блока с обоими выключателями, у которого частота синусоиды на информационном входе равна единице, а на обоих управляющих входах частота синусоиды — в три раза больше.




*а* *б*

**Рис. 3.13.** Пример блока со встроенными выключателями `Enable` и `Trigger`:
*а* — блок-схема; *б* — результат работы

Все событийно-управляемые блоки наследуют свои свойства от блока, куда были помещены значки выключателей. К числу событий, включающих и выключающих блоки, относятся:

- ❒ события, связанные с изменением значения логического выражения (change event);
- ❒ события, порождаемые вызовом функции (call event).

Событие, связанное с изменением значения логического выражения, в свою очередь может наступить, когда выбранная переменная меняет знак от минуса к плюсу (rising), от плюса к минусу (falling), в обоих направлениях (either).

Блоки, описывающие функции с разрывами первого рода (примером может служить блок `Sign`), кусочно-линейные функции (например, `Relay`), могут диагностировать события, связанные с достижением особых точек — разрывов, изменения наклона прямой. Они делают это по умолчанию, но данный режим в некоторых случаях может приводить к дорогостоящим вычислениям, и его можно отключать. В других случаях, когда нет встроенного механизма поиска особых точек, пользователь сам может следить за событиями типа rising, falling, either, связанными с прохождением через ноль выбранной им переменной, с помощью блока `Hit Crossing`.

На рис. 3.14 представлена блок-схема, на которой синусоида ограничивается на уровне $\pm 0.5$. Первый блок `Hit Crossing` настроен на любое пересечение нуля (either) и следит за синусоидой, второй — на пересечение уровня + 0.5 и следит за ограниченной синусоидой. Графики показывают ограниченную синусоиду, результат слежения за пересечением синусоидой нуля, достижение уровня + 0.5.

Событийно-управляемые блоки и блоки, с помощью которых можно реализовать условные выражения, могут использоваться для моделирования гибридных систем, у которых размер фазового вектора постоянен, а меняется только вид правой части. Для моделирования таких гибридных систем необходимо создать блок-схемы, соответствующие всем возможным формам правой части. Кроме того, следует также вырабатывать специальный управляющий сигнал, знак которого изменяется в момент появления дискретного события.

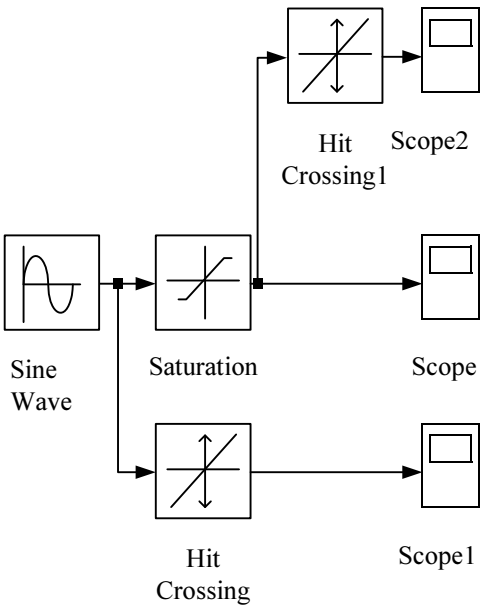
Рассмотрим еще один пример — модель мячика, падающего на вертикальную пружину. Пусть теперь мячик падает на свободный конец невесомой пружины с жесткостью $K$ и длиной $H_s$, закрепленной вертикально на плоскости. Движение мячика задается системой уравнений до касания пружины ($y > H_s$)

$$\begin{cases} \dfrac{dy}{dt} = V_y, \\ \dfrac{dV_y}{dt} = -g. \end{cases}$$

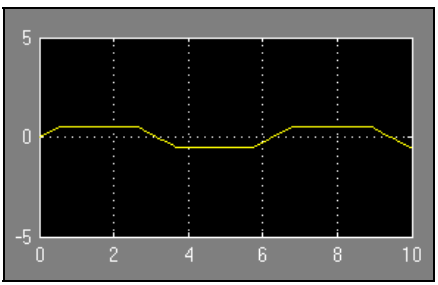
и системой уравнений после ее касания ( $y \le H_s$ )

$$\begin{cases} \dfrac{dy}{dt} = V_y, \\ \dfrac{dV_y}{dt} = K \cdot (H_s - y) - g. \end{cases}$$

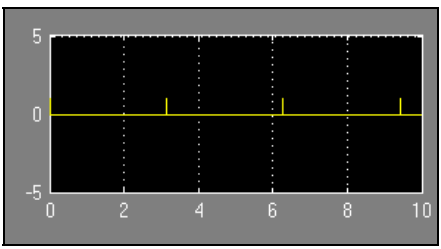
В зависимости от значения коэффициента жесткости пружины (предполагаем, что пружина может сжиматься до нулевой длины) может произойти или



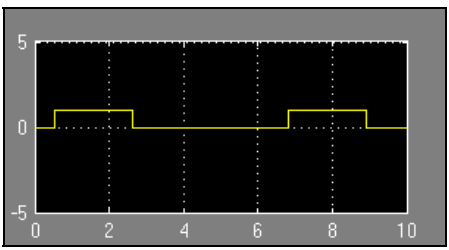

*а*




*б*

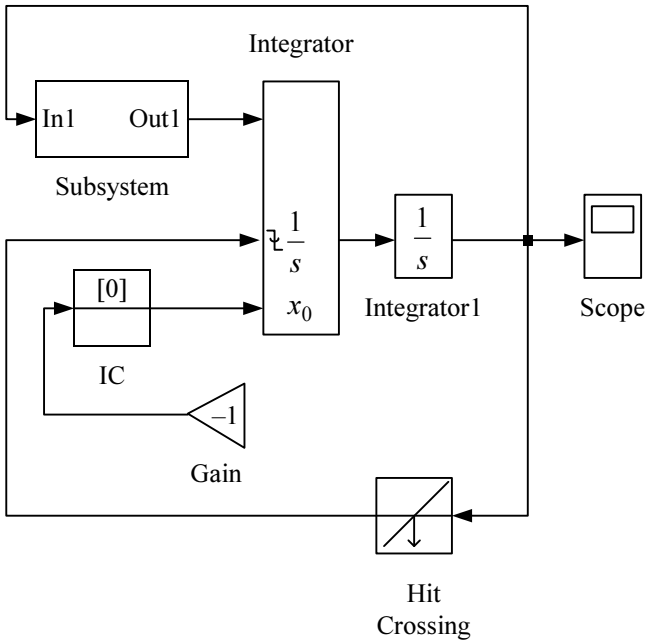



*в*

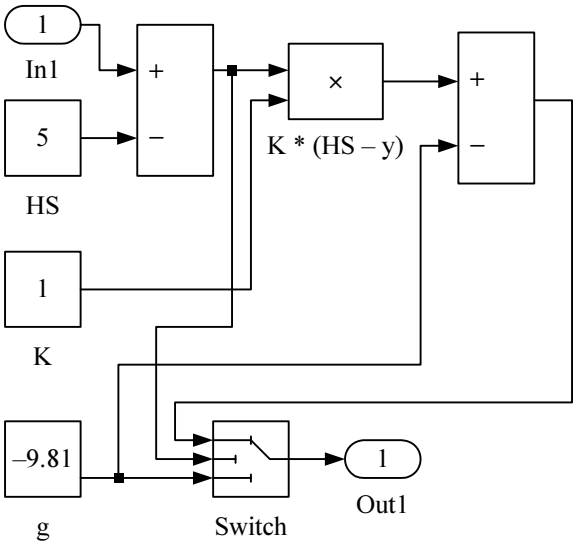



*г*

**Рис. 3.14.** Диагностирование событий: *а* — блок-схема; *б* — ограниченная синусоида; *в* — результат слежения за пересечением синусоидой нуля; *г* — достижение уровня +0.5




*а*




*б*

**Рис. 3.15.** Модель мячика, падающего на пружину: *а* — интегрирование уравнения с изменяющейся правой частью; *б* — подсистема, меняющая вид правой части

не произойти удар мячика о плоскость. При отсутствии удара о плоскость, при касании мячиком пружины происходит только смена уравнений. При наличии удара необходимо также учесть смену знака скорости при отскоке. Блок-схема для этого примера с использованием блока `Switch` для изменения правой части показана на рис. 3.15.

На основной вход первого интегратора подается выражение

```
if (HS-y)>=0 then K*(HS-y)+(-g) else -g
```

набранное из типовых блоков. Переключение ветвей условного выражения осуществляется с помощью блока `Switch`. Таким образом, по существу здесь используется условное уравнение, заданное в свойственной всем пакетам блочного моделирования затейливой форме.

На рис. 3.16 показаны результаты моделирования при `K=10` и `K=1`, соответствующие колебаниям без удара и колебаниям с ударом.

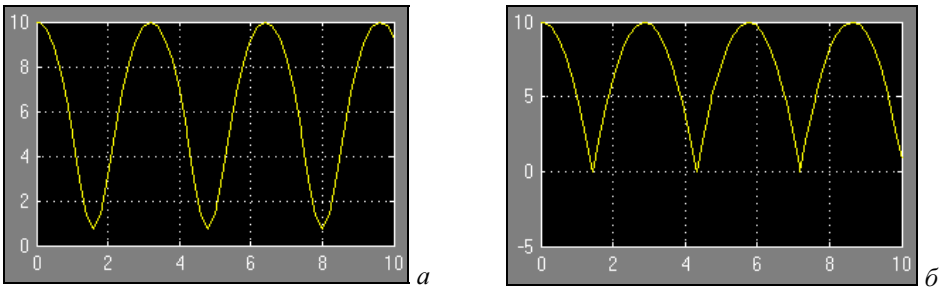



**Рис. 3.16.** Смена уравнений: *а* — колебания без удара; *б* — колебания с ударом

Для большей ясности участки блок-схемы, соответствующие различным вариантам системы уравнений, целесообразно оформлять как подсистемы. На рис. 3.17 представлена модель мячика, падающего на пружину с использованием подсистем для каждого вида правой части уравнения.

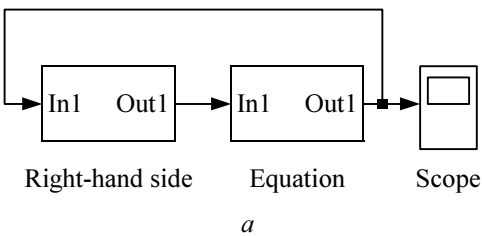
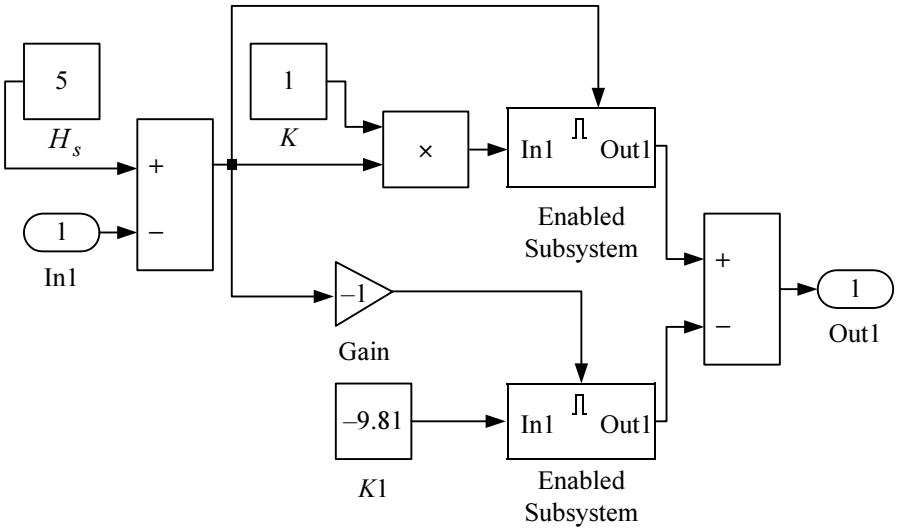
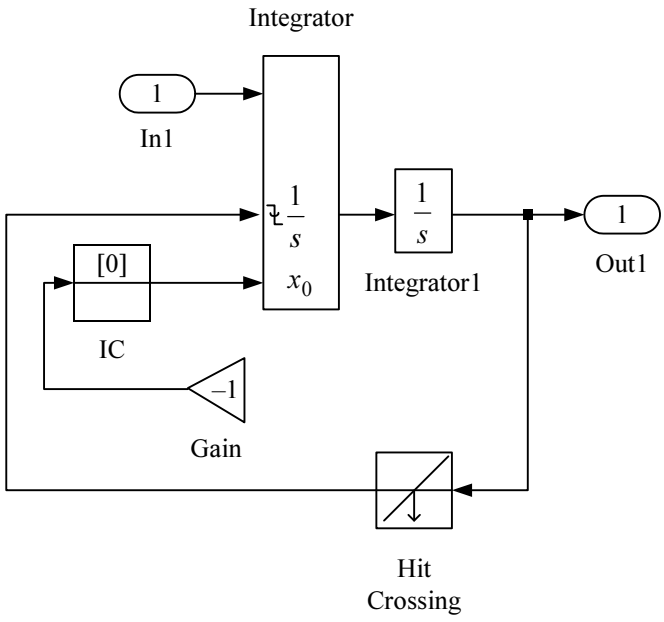



**Рис. 3.17.** *(Часть 1 из 2.)* Модель мячика, падающего на пружину с использованием подсистем для каждого вида уравнений: *а* — блок-схема с двумя подсистемами *б* и *в*




*б*




*в*

**Рис. 3.17.** *(Часть 2 из 2.)* Модель мячика, падающего на пружину с использованием подсистем для каждого вида уравнений: *б* — правые части уравнения; *в* — уравнение

И, наконец, рассмотрим случай, когда возникновение события приводит к новой системе уравнений (качественное изменение поведения модели). В этом случае могут меняться и уравнения, и размер вектора фазовых переменных.

Моделируемая система представляет собой материальную точку (мы будем изображать ее как шарик достаточно малого размера), прикрепленную к нерастяжимому и невесомому стержню длиной $L$, другой конец которого шарнирно закреплен в начале системы координат (рис. 3.18).

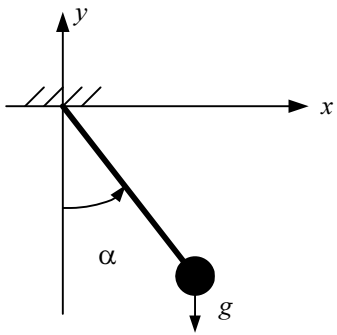



**Рис. 3.18.** Математический маятник в виде стержня и шарика

Состояние маятника полностью определяется значением двух переменных: угла отклонения $\alpha$ и угловой скоростью $\omega$.

Динамика маятника определяется двумя дифференциальными уравнениями:

$$\begin{cases} \dfrac{d\alpha}{dt} = \omega \\ \dfrac{d\omega}{dt} = \dfrac{-g \cdot \sin\alpha}{L} \end{cases}, \ \alpha(0) = \alpha_0, \ \omega(0) = \omega_0 .$$

Пусть в некоторый момент $t^*$ (например, определяемый условием $\alpha \geq \alpha_{\max}$) крепление шарика к стержню разрушается, и далее шарик продолжает свое независимое от стержня движение. Движение шарика после отрыва задается системой уравнений:

$$\begin{cases} \dfrac{dx}{dt} = V_x; & \dfrac{dy}{dt} = V_y; \\ \dfrac{dV_x}{dt} = 0; & \dfrac{dV_y}{dt} = -g. \end{cases}$$

Начальные условия для новой системы уравнений вычисляются в момент $t^*$ по формулам:

$$\begin{cases} x = L \cdot \sin\alpha; \\ y = -L \cdot \cos\alpha; \\ V_x = \omega \cdot \cos\alpha; \\ V_y = \omega \cdot \sin\alpha. \end{cases}$$

Вид траектории в координатах $(x, y)$ для значений параметров $\alpha_0 = -\frac{\pi}{2}$, $\omega_0 = 0$, $\alpha_{\max} = \frac{\pi}{4}$ показан на рис. 3.19.

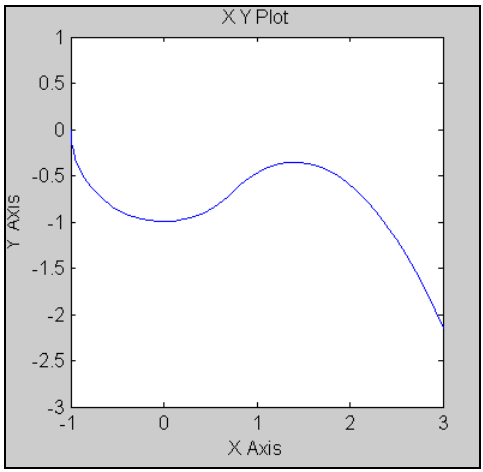



**Рис. 3.19.** Колебания и полет шарика

Попробуем реализовать этот пример в среде Simulink.

На рис. 3.20 показана блок-схема подсистемы Oscillations, соответствующая режиму колебаний маятника. Вместе с величинами $\alpha$ и $\omega$ в подсистеме вычисляются также и значения линейных скоростей и координат. Подсистема содержит блок класса `Enable`, который позволяет отключать всю подсистему по отрицательному значению его входного сигнала. В этом блоке имеется также важный параметр, который указывает, что делать с выходными значениями подсистемы после ее отключения: сбрасывать в начальные или удерживать значения на момент отключения. В данной подсистеме установлен второй режим — удержание последних значений. Это необходимо для ини-

циализации подсистемы Flight. Все интеграторы в подсистеме имеют внешние начальные значения.

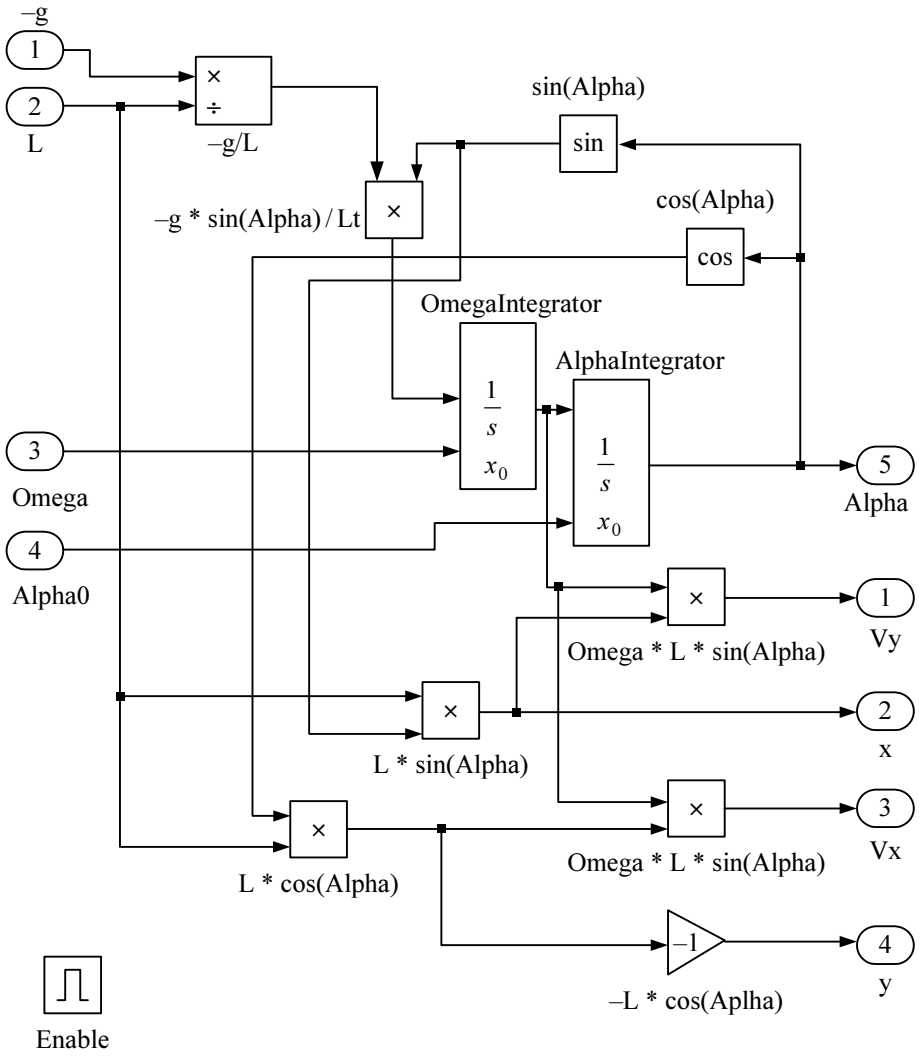



**Рис. 3.20.** Блок-схема подсистемы Oscillations

На рис. 3.21 показана блок-схема подсистемы Flight, соответствующая уравнениям свободного полета. В этой подсистеме также присутствует блок `Enable`, а интеграторы имеют помимо внешних начальных значений интегрируемой величины еще и дополнительные входы сброса, которые управляются

внешним сигналом `Broken`. В параметрах интеграторов указано, что сброс производится по положительному значению сигнала.

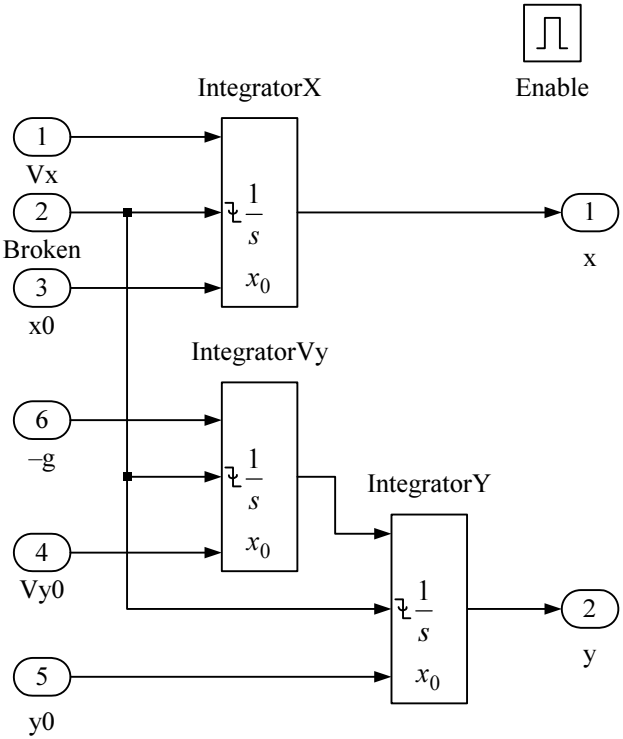



**Рис. 3.21.** Блок-схема подсистемы Flight

Наконец, на рис. 3.22 приведена блок-схема модели в целом. Эта блок-схема помимо подсистем Oscillations и Flight содержит блоки задания констант, блок построения фазовой диаграммы для получения траектории движения в декартовых координатах и схему коммутации подсистем. Для коммутации подсистем вычисляется значение $\alpha_{\max} - \alpha$, которое подается на вход `Enable` подсистемы Oscillations. Таким образом, в момент $\alpha = \alpha_{\max}$ эта подсистема выключается, а ее выходы сохраняют последние вычисленные значения. Одновременно этот сигнал после инверсии (т. е. $\alpha - \alpha_{\max}$) подается на управляющий вход специального переключателя, который при нулевом значении управляющего сигнала (т. е. в момент $\alpha = \alpha_{\max}$) вырабатывает на выходе значение +1. Это значение подается на входы `Enable` и `Broken` подсистемы Flight. Оно же используется для коммутации сигналов, подаваемых на вход блока построения фазовой диаграммы.

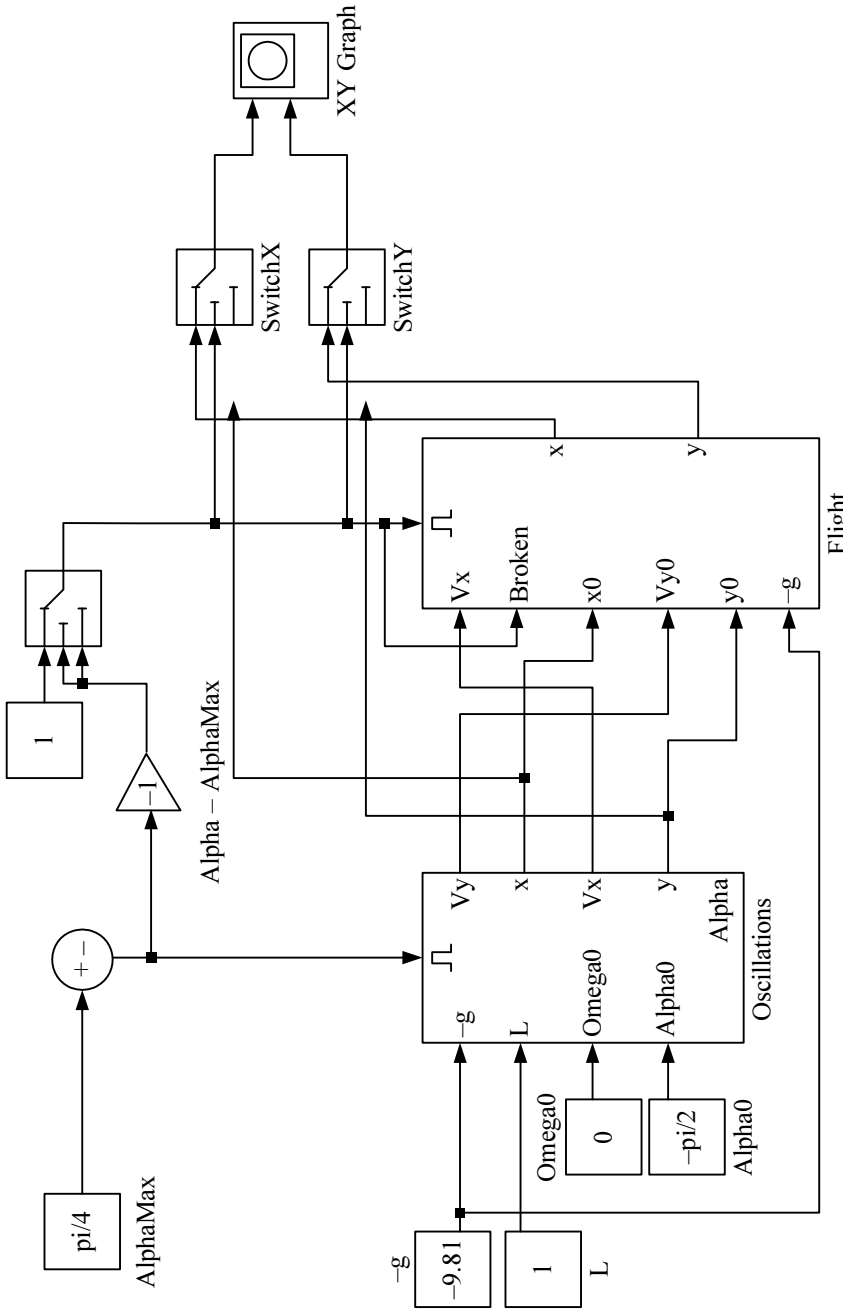



**Рис. 3.22.** Блок-схема модели в целом

Таким образом, стандартными средствами Simulink можно создавать довольно сложные гибридные системы. Трудность создания любых сложных моделей, прежде всего, связана со сложностью набора систем уравнений из примитивных блоков. Для моделирования гибридных систем Simulink предлагает использовать готовые "гибридные" блоки (разрывные функции в правых частях уравнений) или переключать заранее заготовленные альтернативные участки блок-схем (различные виды смены формы уравнений) уравнений. Ясно, что методом переключения ветвей блок-схемы принципиально невозможно моделировать системы с переменным числом объектов. Возникают значительные трудности, связанные с описанием мгновенных действий при обработке дискретного события и тем более при описании цепочки дискретных событий во временной щели (решения типа сброса интегратора следует признать все же искусственными). Кроме того, запутанные переключательные схемы чрезвычайно сложны для понимания.

Для того чтобы помочь пользователю преодолеть эти трудности, в последние версии Simulink введена специальная подсистема Stateflow, позволяющая создавать особые дискретные блоки, функционирование которых задается картой состояний Харела. Блок Stateflow оформляется как подсистема и через свои входы и выходы может взаимодействовать с обычной блок-схемой. Безусловно, подсистема Stateflow существенно облегчает разработку нестандартных дискретных подсистем. Карта состояний позволяет легко задавать любую сложную логику возникновения дискретных событий и их обработки, а также имеет очень наглядное визуальное представление. Однако опыт показывает, что независимое параллельное функционирование дискретной карты состояний и непрерывного поведения, взаимодействующих только через общие переменные, все равно чрезвычайно сложно для восприятия пользователем и часто приводит к трудновыявляемым ошибкам. Подобное решение было использовано в пакете Model Vision 2.1.

Помимо компонента Stateflow, разработчики системы Matlab ввели еще два дополнительных блока для поддержки "физического моделирования": SimPowerSystems и SimMechanics. Введение этих блоков позволяет говорить, что набор продуктов Matlab, Simulink, Stateflow, SimPowerSystems и SimMechanics поддерживает все современные технологии компонентного моделирования.

К числу недостатков "блочного моделирования" следует отнести:

- принципиально невозможно моделирование систем с динамической структурой (переменным числом объектов);
- описание сложного непрерывного поведения приводит к запутанным и неестественным схемам;
- невозможно полноценное объектно-ориентированное моделирование;

- поддержка компонентного моделирования с ненаправленными связями возможна только для специальных наборов блоков.

## Моделирование гибридных систем на языке Modelica ("физическое моделирование")

В основе гибридного моделирования в языке Modelica лежит *принцип синхронного потока данных* (synchronous data flow principle). Это означает, что непрерывная часть модели представляется совокупной системой алгебро-дифференциальных уравнений, а дискретная часть модели трактуется как набор дополнительных уравнений, которые присоединяются к совокупной системе уравнений непрерывной части в момент возникновения дискретного события.

Обработчик дискретного события (блок `when`) представляет собой набор уравнений (их тип ограничен — это могут быть только формулы), добавляемые к текущей совокупной системе уравнений только в момент возникновения ассоциированного с этим блоком дискретного события. В общем случае обработчик ассоциируется с дискретным событием через определенный логический предикат — событие происходит в момент переключения значения этого предиката с `false` на `true`. Предикат для периодических событий, соответствующих разностному уравнению, выделяется как особая предопределенная функция. Декларируется, что исполняющая система должна прекращать численное интегрирование при возникновении дискретного события и затем продолжить его с новыми начальными значениями.

Все переменные делятся на *непрерывные* и *дискретные*. Непрерывные переменные изменяются между дискретными событиями, а дискретные — только в обработчике дискретного события. Значения дискретных переменных сохраняются неизменными до следующего события. Изменение значения непрерывной переменной в обработчике дискретного события возможно только через специальную предопределенную процедуру (вспомним специальный вход интегратора в Simulink).

В результате изменений дискретных переменных в одном обработчике дискретного события может стать истинным предикат другого обработчика и, таким образом, может возникнуть цепочка дискретных событий. Это очень похоже на задание дискретных процессов в языке SLAM II и столь же неаглядно. Очевидное неудобство задания сложной дискретной логики заставляет делать попытки частичной реализации карт состояний в виде макронадстройки над языком Modelica (напрашивается очевидная аналогия с подсистемой Stateflow, добавленной в Simulink). С помощью обработчиков дискретных событий достаточно просто описываются примитивные гибридные автоматы.

Рассмотрим модель прыгающего мячика на языке Modelica. Описание этого примера включает в себя задание константы `g` и переменных `y`, `Vy` с соответствующими начальными значениями, задание двух дифференциальных уравнений и дискретного события с соответствующим блоком `when`, в котором непрерывная переменная `Vy` заново инициализируется своим инвертированным значением.

```
model BauncingBall
  constant Real g = 9.81;
  Real Vy (start=0);
  Real y (start=10);
equations
  when (y<=0) and (Vy<0) then
    reinit(Vy,-Vy);
  end when;
  der(Vy) = -g;
  der(y)  = Vy;
end BauncingBall;
```

Здесь и далее уравнения даются в стандартном текстовом представлении языка Modelica. В то же время в конкретных пакетах, поддерживающих этот язык, уравнения могут представляться и в естественной математической форме.

Modelica позволяет использовать условные выражения в правых частях уравнений. Такие уравнения называют *условными* или *гибридными*. Таким образом, если размер фазового вектора и, соответственно, число уравнений не меняются (гибридное поведение второго типа), то изменение правых частей дифференциальных уравнений задается просто и изящно. Например, запись модели мячика, падающего на вертикальную пружину, выглядит краткой и понятной по сравнению с громоздкой блок-схемой, реализующей эту же модель в Simulink.

В данном примере вполне естественным является использование условного уравнения. Кроме того, вводятся две новые константы.

```
model BauncingBallWithString
  constant Real g = 9.81;
  constant Reak K = 10;
  constant Real HS = 5;
  Real Vy (start=0);
  Real y (start=10);
equations
  when (y<=0) and (Vy<0) then
    reinit(Vy,-Vy);
  end when;
```

```
  der(Vy) = if y>HS then -g else K*(HS-y)-g;
  der(y)  = Vy;
end BauncingBallWithString;
```

Для случая, когда размер фазового вектора меняется, дело обстоит хуже. Язык запрещает делать это во время выполнения модели. Руководство пользователя рекомендует для тех случаев, когда структура фазового вектора изменяется, но размер сохраняется, просто использовать старые переменные, нагружая их другим смыслом. В случае, когда размер меняется, рекомендуется создавать для каждого варианта непрерывного поведения свой локальный объект и по дискретному событию включать следующий объект (у всех объектов языка Modelica имеется предопределенный параметр `enable`), инициализировать его переменные текущими значениями предыдущего объекта и выключать предыдущий объект. Эта методика почти дословно совпадает с методикой Simulink. Рассмотрим ее на примере модели "обрывающегося маятника".

С этим примером язык Modelica справляется уже значительно хуже, чем с предыдущими двумя. Хотя в языке и предусматриваются системы уравнений, структура которых зависит от условия, это условие обязательно должно быть статическим, т. е. окончательная структура системы уравнений должна определяться на этапе компиляции модели, а не на этапе выполнения. В противном случае невозможно построить одноуровневую эквивалентную систему гибридных уравнений. Поэтому в руководстве по языку Modelica предлагается в подобных случаях строить для каждого варианта системы уравнений свой уникальный объект, помещать экземпляры всех этих объектов в объект-контейнер. Затем по соответствующему дискретному событию нужно активизировать соответствующий объект-поведение, задать начальные значения его переменных и использовать далее выходные значения этого объекта как выходные значения всей системы. Для нашего примера этот подход выливается в следующие действия.

Создаются отдельные модели колеблющегося и оторвавшегося маятника.

```
partial model BasePendulum   // абстрактный класс
  parameter Real L;
  constant Real g = 9.81;
  output Real x,y,Vx,Vy;
end BasePendulum;
block Pendulum               // колеблющийся маятник
  extends BasePendulum;
  output Real alpha (start=-pi/2), omega (start=0);
equations
  der(alpha)=omega;
  der(omega)=-g*sin(phi)/L;
```

```
  x=L*sin(alpha);
  y=-L*cos(alpha);
  Vx=omega*L*cos(alpha);
  Vy=omega*L*sin(alpha);
end Pendulum;
block BrokenPendulum          // оторвавшийся маятник
  extends BasePendulum;
equations
  der(x)=Vx;
  der(y)=Vy;
  der(Vy)=-g;
end BrokenPendulum;
model BreaingPendulum         // отрывающийся маятник
  extends BasePendulun (L=1);
  parameter Real phimax = pi/4;
protected
  Boolean Broken (start=false);
  Pendulum pend (L=L, enable=not Broken);
  BrokenPendulum bpend (L=L, enable-Broken);
equations
  when pend.phi>=phimax then
    Broken=true;
    reinit(bpend.x,pend.x);
    reinit(bpend.y,pend.y);
    reinit(bpend.Vx,pend.Vx);
    reinit(bpend.Vy,pend.Vy);
  end when;
  x = if Broken then bpend.x else pend.x;
  y = if Broken then bpend.y else pend.y;
  Vx = if Broken then bpend.Vx else pend.Vx;
  Vy = if Broken then bpend.Vy else pend.Vy;
end BreakingPendulum;
```

Таким образом, в данном примере Modelica предлагает по существу ту же самую схему построения модели гибридной системы, что и Simulink: для каждого поведения строится своя цепочка блоков, входы и выходы которой коммутируются в зависимости от дискретных событий. Отличием Modelica является лишь то, что уравнения можно просто и удобно записывать в исходном математическом виде вместо того, чтобы набирать их из имеющихся блоков низкого уровня (сумматоров, интеграторов и т. п.).

Естественно, при таких ограничениях моделирование систем с динамической структурой (переменным числом объектов) на языке Modelica принципиально невозможно.

Эти ограничения связаны с тем, что поведения всех экземпляров всех классов, входящих в модель, на этапе компиляции превращаются (это декларировано на уровне спецификации языка) в эквивалентную одноуровневую систему уравнений следующего вида:

$$v := \left[ \frac{dx}{dt}; x; y; t; m; \text{pre}(m); p \right],$$
$$c := f_c(\text{relation}(v)),$$
$$m := f_m(v, c),$$
$$0 = f_x(v, c),$$

где:

- $p$ — параметры и константы;
- $t$ — независимое непрерывное время;
- $x(t)$ — дифференциальные переменные;
- $m(t_e)$ — дискретные переменные, которые изменяются только в моменты $t_e$;
- $y(t)$ — алгебраические переменные;
- $c(t_e)$ — условия всех операторов `if` и `when`;
- relation($v$) — отношения между компонентами $v_i$ вида $v_1 > v_2$, $v_3 \geq 0$

и т. п.

Эта система содержит алгебраические уравнения булева типа, и ее решение является весьма непростой задачей, требующей разработки специальных численных методов типа LSODAR. Возникает естественный вопрос: зачем авторам языка Modelica обязательно нужно получать эту систему уравнений, а потом преодолевать трудности ее численного решения?

Скорее всего, имеются две причины. Во-первых, это традиционный подход для моделирования в тех прикладных областях, из которых "выросла" Modelica. Ясно, что разработчики пакета Dymola, поддерживающего язык Modelica, опирались на ранее созданный задел в части компилятора и исполняющей системы. Во-вторых, принятая в языке свобода записи уравнений и ненаправленные связи между компонентами приводят к возникновению непростых проблем при объединении описаний отдельных компонентов в одну эквивалентную систему уравнений. Эти проблемы связаны с анализом корректности системы, поиском набора искомых переменных, поиском согласованных начальных значений и др. Этих проблем нет в языках SLAM II, НЕДИС и Simulink, т. к. там введены жесткие ограничения на форму задания непрерывного поведения и связей между компонентами. Такие ограничения

принципиально неприемлемы для декларированной области применения "моделирование физических систем". Традиционно принято переносить выполнение такого сложного анализа на стадию компиляции и избегать его на стадии выполнения модели. Проблема усугубляется еще и тем, что иногда необходимо выполнять дифференцирование отдельных уравнений. Поскольку выполнять численное дифференцирование крайне нежелательно, необходимо производить аналитическое дифференцирование исходных уравнений. Кроме того, для ряда случаев чрезвычайно полезно аналитическое вычисление матрицы Якоби. Эти вычисления также обычно выполняются на стадии компиляции.

Таким образом, подход "физического моделирования", основанный на языке Modelica, также имеет свои недостатки:

- принципиально невозможно моделирование систем с динамической структурой (переменным числом объектов);
- описание дискретных действий чрезвычайно ненаглядно и не соответствует языку UML.

## Гибридное направление

Подходы, в основе которых лежит модель гибридного автомата, рассмотрим на примере пакета MvStudium из семейства графических сред моделирования Model Vision Studium.

Начнем с примера модели вертикально падающего мячика. Описание этого примера в пакете MvStudium включает в себя:

- определение переменных состояния `y`, `Vy` и параметров `g` и `H` (текстовая форма, фрагмент общего описания модели)

```
parameter g: double := 9.81;
parameter H: double := 10;
y: double := H;
Vy: double := 0;
```

  и его графическое представление (рис. 3.23, окно описания класса);

- систему уравнений $\frac{d^2y}{dt^2} = -9.8$, $y(0) = 10$, $y'(0) = 0$ и ее графическое представление в виде системы двух уравнений первого порядка *(Уравнения_1)* в редакторе уравнений пакета (рис. 3.24). Возможно представление и в виде уравнения второго порядка (на рис. 3.24 это третья, закомментированная строка);
- гибридную карту состояний (рис. 3.25).

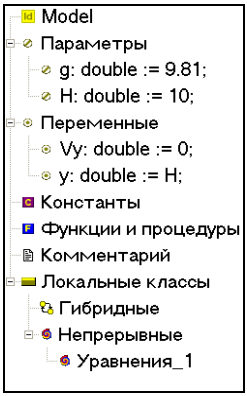



**Рис. 3.23.** Окно описания класса

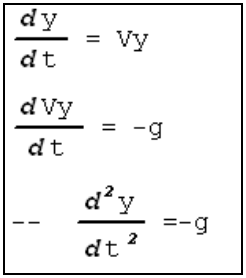



**Рис. 3.24.** Различные формы графического представления дифференциальных уравнений

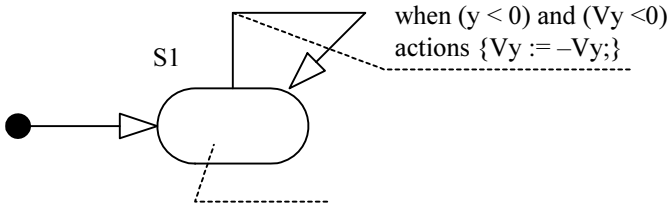



**Рис. 3.25.** Карта поведения модели падающего мячика

Полное описание модели представляет собой текстовый файл, приведенный далее, включающий в себя как собственно описание модели, так и информацию о расположении графических элементов в окнах редактора модели.

```
package BALL is
  import SysLib;
  hybrid class Model is
    parameter g: double := 9.81;
    parameter H: double := 10;
    y: double := H;
    Vy: double := 0;
    local continuous class Уравнения_1 is
      equations {
        d(y)/dt = Vy;
        d(Vy)/dt = -g;
        unknown y, Vy;
      };
    end Уравнения_1;
```

```
    bchart {
      initial state _InitState;
        pragma state(_InitState) rectangle
Top=82,Left=22,Height=16,Width=16;
        pragma state(_InitState) name point X=0,Y=0;
      state S1 do S1: Уравнения_1;
        pragma state(S1) rectangle Top=80,Left=80,Height=20,Width=40;
        pragma state(S1) name point X=-20,Y=-30;
        pragma state(S1) footnote rectangle
Top=160,Left=80,Height=20,Width=110;
      transition _InitTran from _InitState to S1;
        pragma transition(_InitTran) line [(65,90)];
      transition Tr_1 from S1 to S1 when (y<0) and (Vy<0)
        actions {Vy:=-Vy;}
      ;
        pragma transition(Tr_1) line [(110,50),(160,50)];
        pragma transition(Tr_1) footnote rectangle
Top=40,Left=190,Height=60,Width=180;
    };
  end Model;
  static model: Model;
end BALL;
```

На рис. 3.26 представлены различные способы визуализации свободного падения шарика.

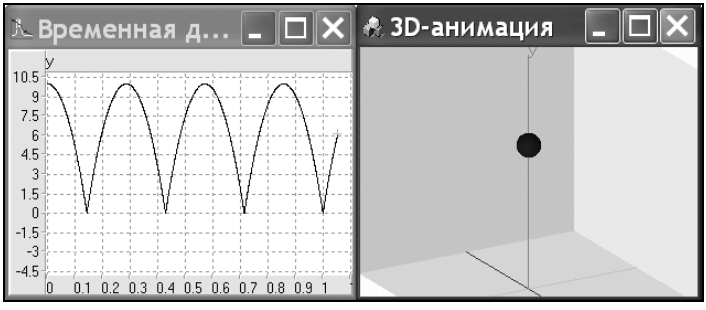



**Рис. 3.26.** Математическая графика (временная диаграмма) и 3D-анимация

Модель мячика, падающего на пружину, в пакете MvStudium может реализовываться способами — с помощью условного уравнения или с использованием вложенной карты поведения. Кроме того, в обоих случаях необходимо ввести дополнительные параметры `K` и `HS` (рис. 3.27).

**Использование условных уравнений.** Пакет Model Vision Studium наряду с гибридными картами состояний позволяет использовать и условные уравне-

ния. В ряде случаев это приводит к значительно более компактному описанию. Заметим, что исполняющей системой пакета переключение ветвей условного уравнения трактуется как неявный переход в карте поведения и обрабатывается соответствующим образом. В данном случае `Система_уравнений_1` должна быть модифицирована, как показано на рис. 3.28.

Карта поведения в этом случае не меняется.

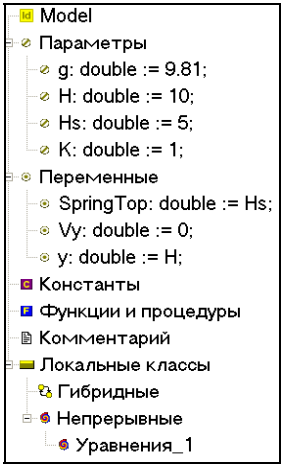
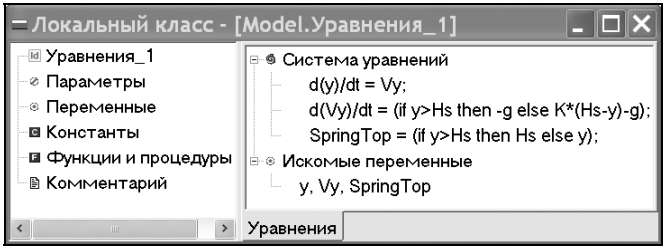
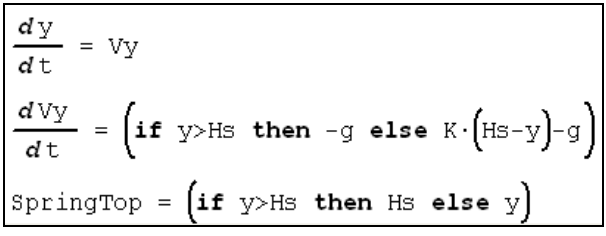



**Рис. 3.27.** Описание класса для модели мячика, падающего на пружину




*а*




*б*

**Рис. 3.28.** Система уравнений: *а* — описание локального класса `Уравнения_1` для модели мячика, падающего на пружину;
*б* — представление условных уравнений в графической форме

**Использование вложенной карты поведения.** В этом случае локальный класс `Уравнения_1` переименуем в `Свободное_падение` и дополнительно создадим систему уравнений `Движение_с_пружиной`. Описание модели приведено на рис. 3.29, а сами уравнения — на рис. 3.30.

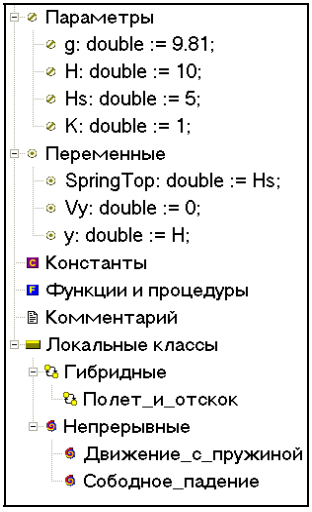



**Рис. 3.29.** Видоизмененное описание модели

$$\mathrm{SpringTop} = \left(\textbf{if } y>Hs \textbf{ then } Hs \textbf{ else } y\right)$$

$$\frac{d\,y}{d\,t} = Vy$$

$$\frac{d\,Vy}{d\,t} = -g+K\cdot\left(Hs-y\right)$$

*а*

$$\frac{d\,y}{d\,t} = Vy$$

$$\frac{d\,Vy}{d\,t} = -g$$

$$\mathrm{SpringTop} = \left(\textbf{if } y>Hs \textbf{ then } Hs \textbf{ else } y\right)$$

*б*

**Рис. 3.30.** Системы уравнения, соответствующие системе условных уравнений: *а* — `Свободное_падение`; *б* — `Движение_с_пружиной`

В главной карте поведения уберем систему уравнений из состояния `Полет` и припишем ему другую карту поведения `Полет_и_отскок` (рис. 3.31).

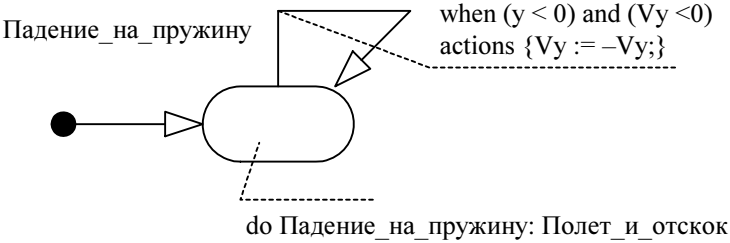



**Рис. 3.31.** Главная карта поведения

В свою очередь карта поведения `Полет_и_отскок` определяет условия переключения между двумя системами уравнений (рис. 3.32).

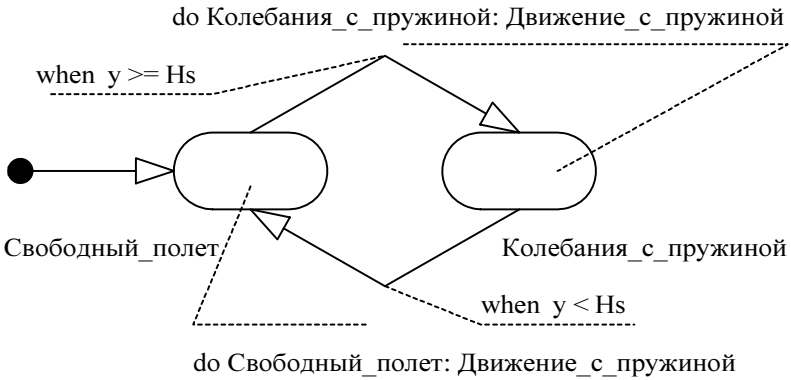



**Рис. 3.32.** Вложенная карта поведения

Таким образом, мы разделили описание сложного движения мячика на две части: главная карта поведения описывает отскок от плоскости, а локальная карта поведения — взаимодействие с пружиной. Этот вариант несколько более громоздкий, чем вариант с условным уравнением, однако он значительно нагляднее при отладке модели, поскольку переключения состояний можно наблюдать визуально и задавать останов модели по срабатыванию перехода. Результат моделирования показан на рис. 3.33.

Далее представлено полное тестовое описание модели, соответствующее модели с главной и вложенной картой поведения.

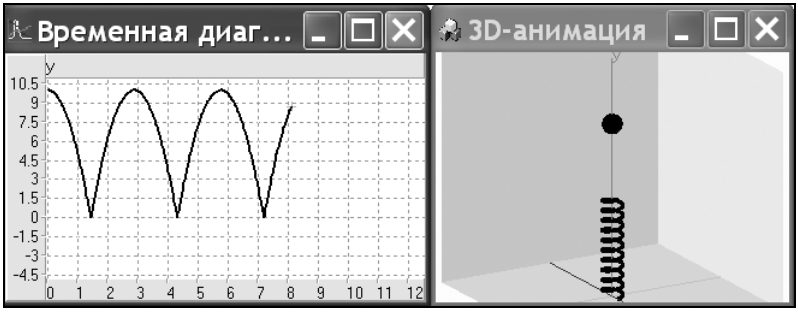



**Рис. 3.33.** Временная диаграмма и 3D-анимация рассмотренного примера

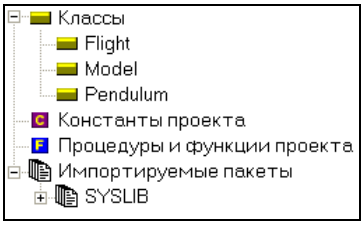
```
package SPRINGBALL_1 is
  import SysLib;
  hybrid class Model is
    parameter g: double := 9.81;
    parameter H: double := 10;
    parameter Hs: double := 5;
    parameter K: double := 1;
    y: double := H;
    Vy: double := 0;
    SpringTop: double := Hs;
    local continuous class Сободное_падение is
      equations {
        d(y)/dt = Vy;
        d(Vy)/dt = -g;
        SpringTop = (if y>Hs then Hs else y);
        unknown y, Vy, SpringTop;
      };
    end Свободное_падение;
    local continuous class Движение_с_пружиной is
      equations {
        SpringTop = (if y>Hs then Hs else y);
        d(y)/dt = Vy;
        d(Vy)/dt = -g+K*(Hs-y);
        unknown y, Vy, SpringTop;
      };
    end Движение_с_пружиной;
    local hybrid class Полет_и_отскок is
      bchart {
        initial state _InitState;
          pragma state(_InitState) rectangle Top=100,Left=30,Height=20,
                                                          Width=20;
```

```
      pragma state(_InitState) name point X=0,Y=0;
    state Свободный_полет do Свободный_полет: Движение_с_пружиной;
      pragma state(Свободный_полет) rectangle
                                  Top=100,Left=80,Height=30,Width=60;
      pragma state(Свободный_полет) name point X=-55,Y=50;
      pragma state(Свободный_полет) footnote rectangle
                                 Top=200,Left=20,Height=30,Width=435;
    state Колебания_с_пружиной do Колебания_с_пружиной:
                                                  Движение_с_пружиной;
      pragma state(Колебания_с_пружиной) rectangle
                                 Top=100,Left=310,Height=30,Width=55;
      pragma state(Колебания_с_пружиной) name point X=-10,Y=50;
      pragma state(Колебания_с_пружиной) footnote rectangle
                                  Top=10,Left=10,Height=30,Width=490;
    transition _InitTran from _InitState to Свободный_полет;
      pragma transition(_InitTran) line [(70,110)];
    transition Tr_1 from Свободный_полет to Колебания_с_пружиной
                                                        when y>=Hs;
      pragma transition(Tr_1) line [(225,45)];
      pragma transition(Tr_1) footnote rectangle
                                  Top=60,Left=30,Height=30,Width=120;
    transition Tr_2 from Колебания_с_пружиной to Свободный_полет
                                                         when y<Hs;
      pragma transition(Tr_2) line [(230,170)];
      pragma transition(Tr_2) footnote rectangle
                                 Top=55,Left=310,Height=30,Width=105;
  };
end Полет_и_отскок;
bchart {
  initial state _InitState;
    pragma state(_InitState) rectangle
                                   Top=82,Left=22,Height=16,Width=16;
    pragma state(_InitState) name point X=0,Y=0;
  state Падение_на_пружину do Падение_на_пружину: Полет_и_отскок;
    pragma state(Падение_на_пружину) rectangle
                                   Top=80,Left=80,Height=20,Width=40;
    pragma state(Падение_на_пружину) name point X=-60,Y=-70;
    pragma state(Падение_на_пружину) footnote rectangle
                                 Top=110,Left=30,Height=30,Width=400;
  transition _InitTran from _InitState to Падение_на_пружину;
    pragma transition(_InitTran) line [(65,90)];
  transition Tr_1 from Падение_на_пружину to Падение_на_пружину
                                                 when (y<0) and (Vy<0)
    actions {Vy:=-Vy;}
  ;
```

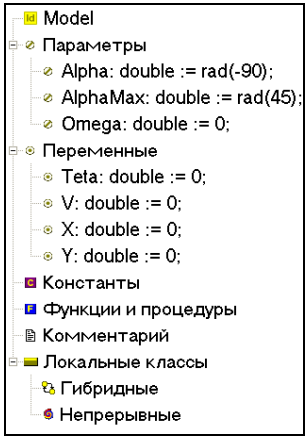
```
        pragma transition(Tr_1) line [(110,50),(160,50)];
        pragma transition(Tr_1) footnote rectangle
                                    Top=40,Left=190,Height=60,Width=180;
    };
  end Model;
  static model: Model;
end SPRINGBALL_1;
```

Перейдем теперь к модели обрывающегося маятника, т. е. к самому трудному для других сред визуального моделирования.

Графическое описание модели показано на рис. 3.34, а ее главная карта поведения — на рис. 3.35.




*а*




*б*

**Рис. 3.34.** Графическое описание модели: *а* — список классов; *б* — класс `Model`

Модель состоит из трех классов: класса `Model`, представляющего моделируемую систему в целом, и ее двух классов, соответствующих двум качественно различным состояниям: `Pendulum` и `Flight` — колебаниям маятника и свободному полету шарика.

Присутствие в модели двух классов `Pendulum` и `Flight` может вызвать недоумение, но в данном случае оно вполне оправдано — сложные модели в MvStudium можно составлять из уже имеющихся классов. Предполагается, что до составления этой "сложной" модели были созданы и отлажены модели "Колебания математического маятника" и "Свободный полет", и соответствующие классы теперь используются для модели "Обрывающийся маятник". Описание классов `Pendulum` и `Flight` и их уравнений представлено на рис. 3.36 и 3.37. Результат моделирования представлен на 3.37.

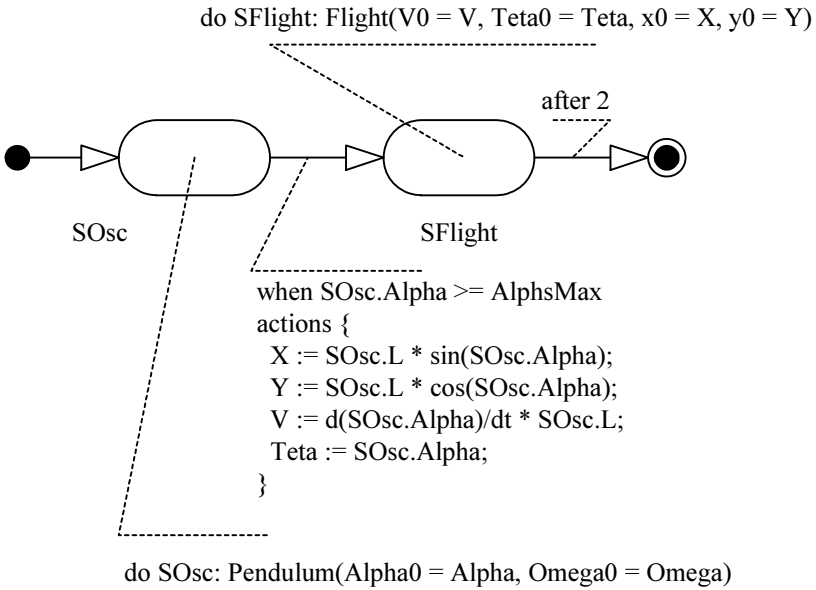
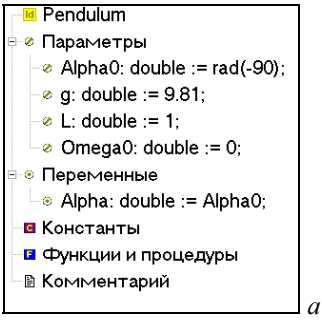
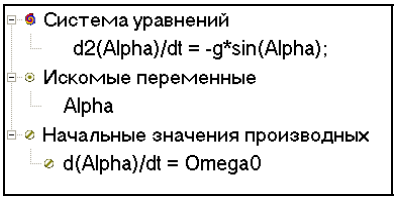



**Рис. 3.35.** Главная карта поведения

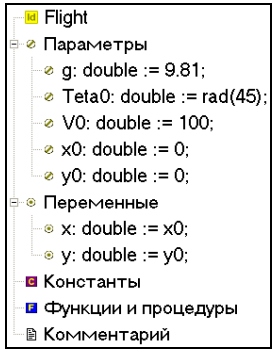



*а*

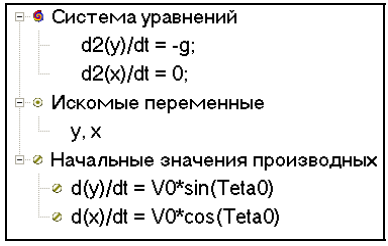



*б*

**Рис. 3.36.** Класс `Pendulum`:
*а* — описание класса;
*б* — описание уравнений класса

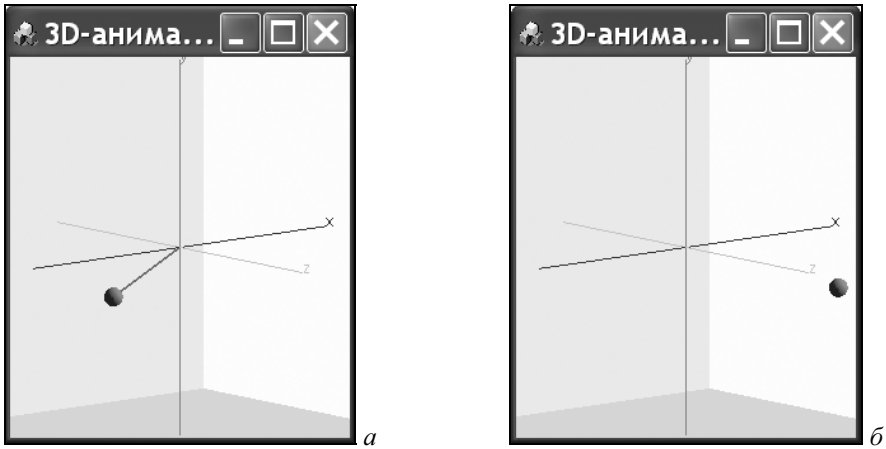



*а*




*б*

**Рис. 3.37.** Класс `Flight`:
*а* — описание класса;
*б* — описание уравнений класса




*а* *б*

**Рис. 3.38.** Визуализация поведения: *а* — колебания маятника; *б* — разрушение

Приведем сразу тестовое описание модели, чтобы показать, что даже в текстовой форме она выглядит очень просто.

```
package BREAKINGPEND is
  import SysLib;
  continuous class Flight is
    parameter g: double := 9.81;
    parameter V0: double := 100;
    parameter Teta0: double := rad(45);
    parameter x0: double := 0;
    parameter y0: double := 0;
    x: double := x0;
    y: double := y0;
    equations {
      d2(y)/dt = -g;
      d2(x)/dt = 0;
      unknown y, x;
    };
    initial {
      d(y)/dt = V0*sin(Teta0);
      d(x)/dt = V0*cos(Teta0);
    };
  end Flight;
  continuous class Pendulum is
    parameter Alpha0: double := rad(-90);
    parameter Omega0: double := 0;
    parameter L: double := 1;
    parameter g: double := 9.81;
    Alpha: double := Alpha0;
```

```
  equations {
    d2(Alpha)/dt = -g*sin(Alpha);
    unknown Alpha;
  };
  initial {
    d(Alpha)/dt = Omega0;
  };
end Pendulum;
hybrid class Model is
  parameter Alpha: double := rad(-90);
  parameter Omega: double := 0;
  parameter AlphaMax: double := rad(45);
  X: double := 0;
  Y: double := 0;
  V: double := 0;
  Teta: double := 0;
  bchart {
    initial state _InitState;
      pragma state(_InitState) rectangle
                                  Top=105,Left=20,Height=20,Width=20;
      pragma state(_InitState) name point X=0,Y=0;
    state SOsc do SOsc: Pendulum(Alpha0=Alpha,Omega0=Omega);
      pragma state(SOsc) rectangle Top=100,Left=70,Height=30,Width=80;
      pragma state(SOsc) name point X=-30,Y=40;
      pragma state(SOsc) footnote rectangle
                                  Top=380,Left=30,Height=30,Width=410;
    state SFlight do SFlight: Flight(V0=V,Teta0=Teta,x0=X,y0=Y);
      pragma state(SFlight) rectangle
                                  Top=100,Left=200,Height=30,Width=80;
      pragma state(SFlight) name point X=0,Y=40;
      pragma state(SFlight) footnote rectangle
                                  Top=20,Left=80,Height=30,Width=350;
    final state _FinalState;
      pragma state(_FinalState) rectangle
                                  Top=105,Left=330,Height=20,Width=20;
      pragma state(_FinalState) name point X=0,Y=-2;
    transition _InitTran from _InitState to SOsc;
      pragma transition(_InitTran) line [(60,115)];
    transition Tr_1 from SOsc to SFlight when SOsc.Alpha>=AlphaMax
      actions {
        X:=SOsc.L*sin(SOsc.Alpha);
        Y:=-SOsc.L*cos(SOsc.Alpha);
        V:=d(SOsc.Alpha)/dt*SOsc.L;
        Teta:=SOsc.Alpha;
      }
    ;
```

```
        pragma transition(Tr_1) line [(180,115)];
        pragma transition(Tr_1) footnote rectangle
                                    Top=200,Left=120,Height=180,Width=280;
      transition Tr_2 from SFlight to _FinalState after 2;
        pragma transition(Tr_2) line [(310,115)];
        pragma transition(Tr_2) footnote rectangle
                                      Top=60,Left=275,Height=30,Width=60;
    };
  end Model;
  static model: Model;
end BREAKINGPEND;
```

Рассмотрение примеров в среде MvStudium завершим хорошо известной моделью, часто обсуждаемой специалистами в области "физического моделирования" (рис. 3.39). Речь идет о моделировании схем, в которых содержатся диоды или любые другие схемные переключатели, приводящие к мгновенной смене конфигурации цепи и, соответственно, к смене уравнений и числа переменных состояния.

Ее функционирование задается уравнениями ("базовая система уравнений"):

$$\begin{cases} \dfrac{dV_2}{dt} = I_2; \\ I_0 = I_1 + I_2; \\ I_2 = \dfrac{V_2}{R_2}; \\ R_1 \cdot I_0 = V_0 - V_1; \\ U = V_1 - V_2; \\ V_0 = A \cdot \sin 2\pi\omega t. \end{cases}$$

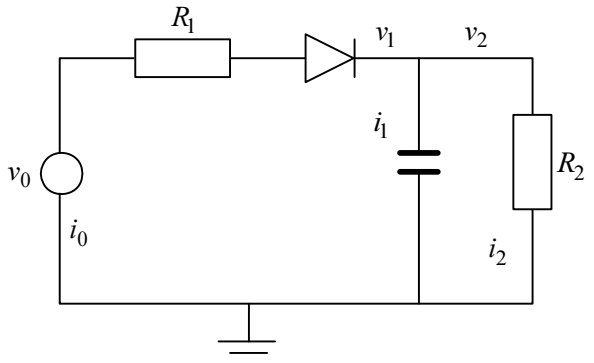
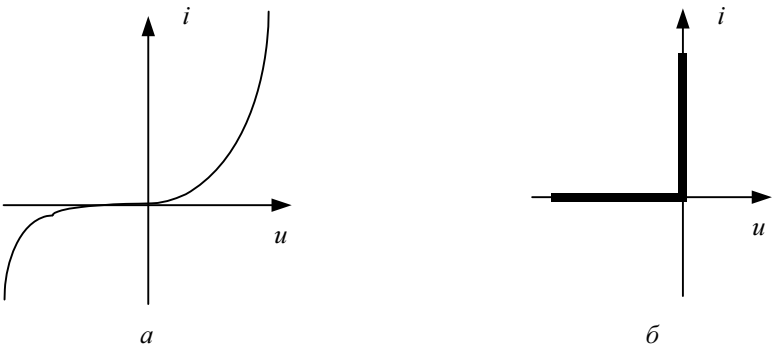



**Рис. 3.39.** Электрическая цепь с выключателем, реализованным с помощью диода

Диод имеет реальную вольт-амперную характеристику, показанную на рис. 1.40, *а*, которая обычно заменяется идеальной 1.40, *б*, чтобы избежать решения нелинейных уравнений.



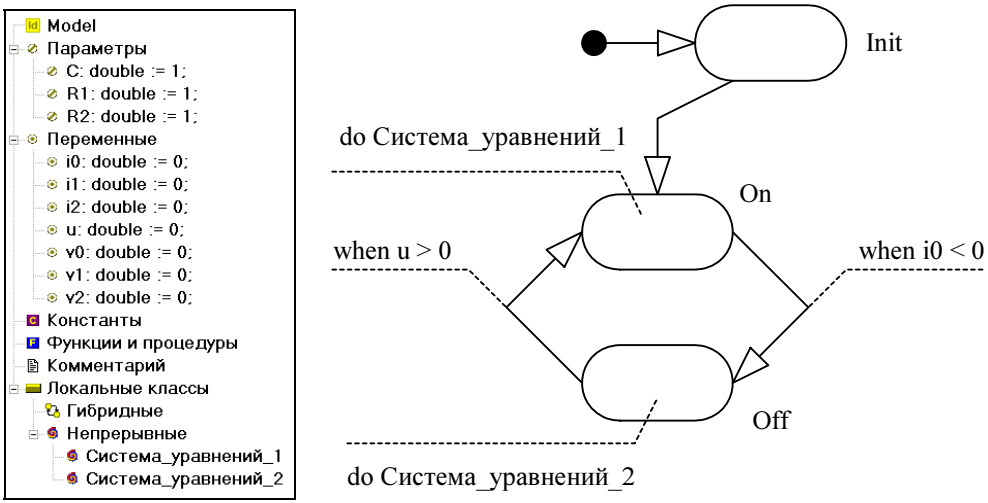

*а* *б*

**Рис. 3.40.** Вольт-амперная характеристика диода: *а* — реальная; *б* — идеальная

Специалистами в области физического моделирования было предложено использовать вспомогательную переменную s, с помощью которой можно описывать идеальную вольт-амперную характеристику однозначно:

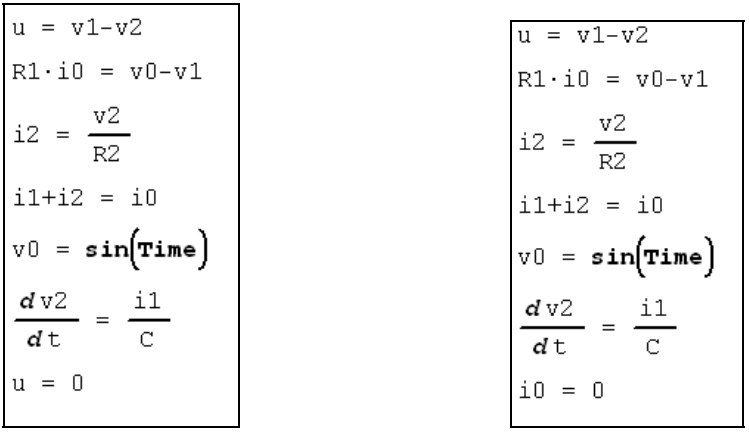
```
model Rectifier
  parameter Real A = 220;
  parameter Real Omega = 50;
  parameter Real R1 = 100;
  parameter Real R2 = 100;
  parameter Real C = 1E-10;
  Real V0,V1,V2,U;
  Real I0,I1,I2;
  Real s;
  Boolean off;
equations
  off = (s<0);
  U = V1-V2;
  U = if off then s else 0;
  I0 = if off then 0 else s;
  R1*I0 = V0-V1;
  V0 = A*sin(2*pi*Omega*time);
  I2 = V2/R2;
  I1 = I0-I2;
  der(V2) = I1/C;
end Rectifier;
```

Это красивый, но искусственный прием. Для решения подобных проблем проще применить формализм гибридных автоматов. Поведение выпрямителя задается гибридной картой состояний (рис. 3.41). Решаемые уравнения представлены на рис. 3.42, результаты моделирования на рис. 3.43.




*а* *б*

**Рис. 3.41.** Гибридная карта состояния: *а* — описание модели; *б* — карта поведения




*а* *б*

**Рис. 3.42.** Решаемые уравнения: *а* — уравнения *Система_1*; *б* — уравнения *Система_2*

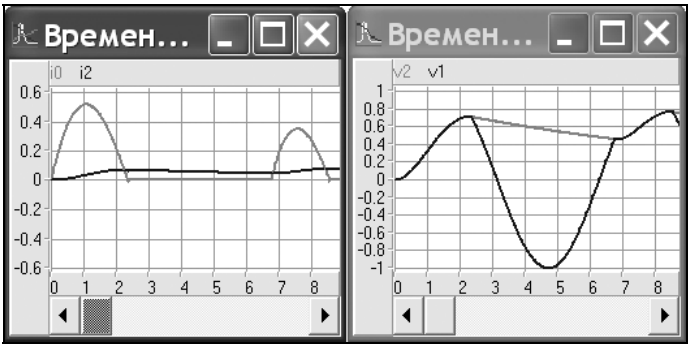



**Рис. 3.43.** Результаты моделирования

# Языки объектно-ориентированного моделирования

Объектно-ориентированный подход позволяет задавать моделируемую систему в виде совокупности объектов. Каждый объект всегда является экземпляром какого-то класса с собственными значениями параметров. Определение класса задает прототип объекта. Иногда предусматривается параметризация самого определения класса. Другими составляющими объектно-ориентированного подхода являются наследование классов и полиморфизм экземпляров объектов. Для объектно-ориентированного моделирования какого-то конкретного вида систем, например гибридных, необходимо определить, что считается объектом, задать способы параметризации объектов, а также определить, что конкретно означают наследование и полиморфизм для объектов данного вида. Далее показано, как это делается в ряде существующих языков объектно-ориентированного моделирования.

## Simula-67 и НЕДИС

В языке Simula-67 под *объектом* понимается совокупность переменных и методов (функций и процедур, т. е. традиционный объект в языках программирования). Кроме того, с объектом может быть связан процесс, выполняемый параллельно с процессами в других объектах. Под процессом понимается просто последовательность операторов, включающая специальные операторы синхронизации. Таким образом, уже в этом языке появляется деление объектов на "активные" (объект, имеющий собственную "нитку управления") и "пассивные" (методы "пассивного" объекта могут быть вызваны из "активных" объектов или из главной программы). Предполагалось, что, построив подходящие наборы стандартных классов, можно будет на базе Simula-67

создавать специализированные языки моделирования. Подкласс наследует все атрибуты и методы суперкласса, виртуальные методы могут быть переопределены. Унаследованное дискретное поведение (процесс), таким образом, может быть переопределено с помощью переопределения используемых в теле процесса процедур и функций.

Язык НЕДИС является расширением Simula-67 и позволяет наряду с дискретным поведением моделировать также и непрерывное поведение. Дифференциальные уравнения задаются в форме явного вызова функции интегрирования с указанием правой части уравнения, заданной в виде функции. Таким образом, подкласс наследует непрерывное поведение суперкласса. С помощью переопределения соответствующих функций можно в подклассе переопределять уравнения, заданные в суперклассе. Возможны как статические, так и динамические экземпляры объектов.

## ObjectMath

Язык ObjectMath является расширением входного языка пакета Mathematica. Основной концепцией этого языка является поддержка "объектно-ориентированного математического моделирования", противопоставляемого объектно-ориентированному программированию. Под *объектом* в ObjectMath понимается совокупность переменных, процедур и функций, а также уравнений (в синтаксисе Mathematica). Некоторые переменные, объявленные как внешние, доступны извне объекта. Взаимодействие объектов осуществляется через уравнения объекта-контейнера, которые играют роль уравнений связей между внешними переменными локальных объектов. Подкласс наследует все элементы суперкласса (включая уравнения). В подклассе могут быть добавлены новые переменные, функции и уравнения, а также переопределены унаследованные функции и уравнения. Все экземпляры объектов статические.

## Omola

Несмотря на то, что язык Omola как бы "влился" в язык Modelica, следует его рассматривать отдельно, поскольку он имеет ряд интересных особенностей, которых нет в Modelica. Прежде всего, в Omola различают два вида объектов: "полноценные" объекты и "объекты с семантикой значения", подобные структурам в языке C#. Основное различие между этими видами объектов состоит в том, что для объектов первого вида имя объекта понимается как указатель на данный экземпляр, а для объектов второго вида оно понимается как значение. Определения классов можно группировать в библиотеки классов. Имеется ряд "базовых" классов, определенных в библиотеке `Base`. "Полноценный" объект в Omola ассоциируется с параллельно функционирующим

компонентом модели. Компонент представляет собой совокупность атрибутов. Атрибутами могут быть:

- ❒ локальные объекты;
- ❒ определения локальных классов;
- ❒ простые переменные вещественного, целого, булева и строкового типов, а также матрицы;
- ❒ уравнения;
- ❒ формулы;
- ❒ связи.

К локальным объектам относятся:

- ❒ локальные компоненты (элементы локальной блок-схемы, которые, возможно, связаны связями);
- ❒ параметры;
- ❒ переменные-объекты;
- ❒ "терминалы", т. е. внешние переменные, через которые осуществляются связи между компонентами;
- ❒ дискретные события.

Определение локального класса находится внутри определений другого класса и в нем видимы все атрибуты этого класса.

Подкласс наследует все атрибуты суперкласса. В подкласс могут быть добавлены любые новые атрибуты. Переопределение атрибута суперкласса осуществляется простым и изящным способом: любой атрибут подкласса с именем, совпадающим с именем атрибута суперкласса, переопределяет последний. Таким способом можно, например, изменить тип значения унаследованной переменной. Скажем, атрибут `value` в базовом классе `Variable` имеет вещественный тип, но в любом потомке класса `Variable` можно объявить его с другим типом значения. Таким образом, нет необходимости в использовании параметризованных классов. Поскольку для переопределения требуется имя атрибута, не могут быть переопределены уравнения и связи. Формулы могут быть переопределены, поскольку они рассматриваются как дополнение ("связывание") к определению переменной, стоящей в левой части формулы. Такой подход к переопределению атрибутов противоречит, правда, общепринятому подходу ООП, в котором атрибут с совпадающим именем лишь замещает атрибут суперкласса в теле подкласса (но не в теле суперкласса).

Еще одной интересной особенностью Omola является возможность указания в атрибуте `quantity` переменной-объекта физической сущности переменной из некоторого списка (угол, расстояние, масса, температура и т. п.), а в атрибуте `unit` — единиц измерения.

# Modelica

Язык Modelica в настоящее время является самым "продвинутым" языком ООМ. В этом языке декларируется, что классом является практически любое определение (даже алгоритмическая функция), вследствие чего структура классов получается довольно запутанной. Выделяются важные семантические разновидности классов ("ограниченные классы" в терминологии Modelica или стереотипы классов в терминологии UML):

- `model` — неориентированный компонент, который может содержать параметры, переменные, коннекторы, локальные компоненты, уравнения (в том числе уравнения связей), алгоритмы;
- `block` — ориентированный компонент, все внешние переменные которого должны быть либо входами, либо выходами;
- `connector` — внешняя переменная, которая может участвовать в связях, но не может содержать уравнений;
- `record` — определение записи;
- `type` — определение типа пользователя, может являться расширением предопределенных типов, записи, перечислимого типа и массива;
- `package` — пакет (библиотека классов), может содержать только определения классов и констант;
- `function` — алгоритмическая функция.

Неявно предполагается, что все классы стереотипа `type` являются классами "объектов с семантикой значения".

Атрибуты компонента могут быть параметрами, переменными, коннекторами, компонентами, функциями или константами. Атрибуты могут быть видимыми извне (по умолчанию) или инкапсулированными в описании компонента. В связях могут участвовать только коннекторы. Таким образом, компоненты могут взаимодействовать между собой по связям через коннекторы, а также через уравнения охватывающего компонента, в которые входят видимые извне атрибуты. Для взаимодействий вида "один со всеми" или "все со всеми", характерных для задач, в которых фигурируют физические поля, предусмотрен специальный механизм inner/outer-атрибутов. Этот механизм заключается в следующем: если в каком-либо компоненте модели объявлен атрибут со статусом `outer`, то он рассматривается как ссылка на некоторый атрибут, имеющий то же самое имя и совместимый тип, но объявленный в каком-либо из охватывающих компонентов и имеющий статус `inner`. Например, в определении класса, задающего свободное движение материальной точки, можно не детализировать зависимость ускорения силы тяжести от высоты, предполагая, что эта функция будет определена в охватывающем компоненте, задающем правила вычислительного эксперимента.

Подкласс наследует все элементы описания суперкласса. Ни удалить, ни переопределить унаследованные элементы описания нельзя. Например, если вы хотите использовать модель сопротивления, в которой учитывается влияние температуры, то не сможете ввести класс "термосопротивление" как потомок класса "сопротивление", поскольку для этого вам пришлось бы заменить уравнение $R \cdot I = V$ другим уравнением $(R_0 + R_T \cdot (T - T_0)) \cdot I = V$. В подклассе можно модифицировать лишь начальные значения переменных, значения параметров по умолчанию и значения констант. Помимо понятий "подкласс" и "суперкласс" вводятся также понятия "подтип" и "супертип". Класс A является подтипом класса B, если содержит все видимые извне атрибуты класса B (совпадение по именам) и типы этих атрибутов являются подтипами соответствующих типов атрибутов класса B. Это очень похоже на отношение `implements` между классом и интерфейсом в языке Java, если под интерфейсом понимать совокупность методов и переменных. Для функций совместимость по интерфейсу означает наличие одноименных параметров с совместимыми типами и совместимость типов возвращаемого значения. Объект класса B может быть заменен на объект класса A, даже если класс A не является производным от B. Это дает возможность параметризации класса, которая позволяет переопределять переменные при наследовании этого класса. Например, вы можете определить класс C1, задающий некоторую схему, в которой используется обычное сопротивление $R_1$. Затем вы можете создать производный от него класс C2, в котором компонент $R_1$ будет заменен или на термосопротивление, или на конденсатор, или на индуктивность. Эти замещения будут корректными, поскольку все замещающие и замещаемый классы по внешнему интерфейсу являются подтипами типа "двухполюсник". Механизм замещения может применяться не только к атрибутам-экземплярам, но и к атрибутам-классам. Таким образом, с помощью замещения создается возможность параметризации классов. Правда, атрибуты, которые могут быть замещены, должны быть специально помечены в определении класса.

Для объединения элементов описания в группы (библиотеки классов) используются пакеты. *Пакет* — это контейнер для группы компонентов, ограничивающий область их видимости. Компоненты, объявленные как `public`, видимы извне под составным именем, включающим в качестве префикса имя пакета. Остальные компоненты видимы только внутри данного пакета. В отличие от языков программирования, где компонентами пакета являются только классы, естественными компонентами пакета в ООМ являются также константы и алгоритмические функции. Для того чтобы пакет был виден в другом пакете или модели, его нужно импортировать, т. е. указать явным образом на его использование. Импортирование пакета означает, что его имя становится видимым в импортирующем пакете или модели. Можно импорти-

ровать не весь пакет, а только конкретные компоненты пакета. В качестве компонента пакета может выступать другой пакет. Относительно видимости компонентов пакета во вложенных пакетах имеются два решения: по умолчанию все компоненты охватывающего пакета видны во вложенном пакете, в пакетах со статусом `encapsulated` в пакете видимы только собственные и явно импортируемые компоненты.

## Инструменты "блочного моделирования"

Инструменты этого направления предоставляют очень ограниченные возможности ООМ. Библиотеки стандартных блоков являются библиотеками классов. Пользователь может поместить экземпляр стандартного класса в модель и изменить значения его параметров. Однако определить новый класс в рамках входного языка можно только как подсистему, поведение которой задается структурной схемой.

# Анализ существующих языков ООМ применительно к моделированию сложных динамических систем

В рассмотренных языках ООМ можно выделить несколько характерных особенностей, существенных для моделирования сложных динамических систем.

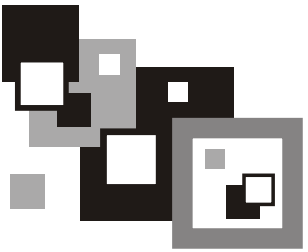
При моделировании непрерывных систем в качестве интерфейса объекта рассматривается совокупность внешних переменных, а не совокупность методов и сообщений, как в языках объектно-ориентированного программирования и UML. Это отражает объективные особенности функционирования непрерывных систем. Следовательно, в языке моделирования сложных динамических систем, ориентирующемся на UML, понятие активного объекта должно быть расширено.

Под *объектом* понимается структурный компонент моделируемой системы, функционирующий параллельно в модельном времени с другими структурными компонентами. Существующие языки ООМ ориентированы в значительной степени на технологию "промышленного" моделирования, предполагающую преимущественное использование в конкретных проектах заранее разработанных прикладных библиотек классов. Предполагается, что определения классов в основном будут писаться высококвалифицированными разработчиками библиотек, а обычные пользователи будут строить модель в виде совокупности стандартных объектов, взаимодействующих явно через связи или неявно через уравнения или дискретные действия объекта-контейнера. В этом смысле подходы Simulink и Dymola очень близки, только Dymola по-

зволяет строить значительно более сложные компоненты, адекватные элементам моделируемых физических систем. Однако далеко не для всех приложений является естественным представление модели в виде структурной схемы. В исследовательских задачах, в учебном процессе и на ранних стадиях разработки систем управления пользователь в основном имеет дело с моделями отдельных изолированных систем. Не случайно для этих задач чаще используются математические пакеты, ориентированные на изолированные системы и функциональный стиль описания, а не пакеты компонентного моделирования. Для моделирования изолированных систем актуальным является формирование сложного поведения из отдельных фрагментов. Эти модели далее становятся основой для разработки компонентов, которые будут объединены в модель всей системы. Возможность пользователя разрабатывать укрупненные высокоуровневые компоненты, отражающие естественную структуру прикладной задачи, является одним из важнейших направлений развития современных инструментов моделирования. Представляется, что в современном инструменте системно-аналитического моделирования структурные схемы должны использоваться только в том случае, когда они отражают естественную структуру системы и облегчают разработку и понимание модели. Формализм гибридного автомата позволяет строить сложные поведения путем последовательного, а не параллельного включения отдельных компонентов. Очевидно, что динамические системы, приписываемые состояниям гибридного автомата, можно рассматривать как компоненты — экземпляры соответствующих классов. Таким образом, открывается своего рода "новое измерение" для ООМ. Используя возможность последовательного включения компонентов, можно многие задачи моделирования решать вообще без составления структурных схем, ограничиваясь моделью изолированной системы.

Практически во всех существующих языках ООМ предполагается жесткая фиксация стереотипа атрибутов, который затем невозможно изменить в производном классе. Однако опыт объектно-ориентированного программирования говорит о том, что использования наследования и полиморфизма объектов действительно чрезвычайно плодотворны, если вся иерархия классов тщательно продумана заранее. Но в ходе исследовательского моделирования сделать это практически невозможно. Прежде всего, это относится к определению стереотипа переменных, т. е. разделению переменных на входы, выходы, контакты, параметры и т. д. На ранних этапах проектирования или научного исследования, на стадии отработки отдельных компонентов чрезвычайно трудно предугадать возможные взаимодействия, которые могут потребоваться при объединении этих компонентов на последующих этапах. Например, если вы исследуете движение тела, брошенного под углом к горизонту, вполне естественно в качестве первого приближения использовать до-

пущение о постоянстве ускорения силы тяжести и плотности воздуха, т. е. задать эти величины как константы. Впоследствии вы можете установить, что рабочий диапазон дальностей требует уточнения модели, и учесть зависимость ускорения силы тяжести и плотности от высоты. В руководстве по языку Modelica именно этот пример демонстрируется как случай плодотворного использования inner/outer-функций для создания набора моделей различной точности. Однако при декларации, например, ускорения силы тяжести как outer-функции вы, во-первых, сразу должны знать, что вам потребуются модели поля тяготения различной точности, а во-вторых, помнить, что ускорение силы тяжести зависит только от высоты. В менее тривиальных случаях это требует специального исследования. Таким образом, язык системно-аналитического моделирования должен позволять изменять стереотип переменной в производном классе.



# ГЛАВА 4

# Многообъектные модели

*Многообъектные* (мультиобъектные, мультиагентные) *модели* являются, если отвлечься от технических деталей, связанных с конкретной технологией моделирования, специальным случаем компонентных моделей. В этом случае исследователь не в состоянии описать поведение исследуемой системы в целом, но может представить ее в виде совокупности параллельно функционирующих взаимодействующих объектов (агентов) и надеется получить информацию о поведении системы в целом, изучая ее с помощью вычислительного эксперимента, как если бы он экспериментировал с реальным объектом. Мультиагентные модели наиболее часто используются в таких областях, как, например, социология и других, еще общепризнанных изучаемых. В одной из первых книг в этой области [14] такие модели называются восходящими (bottom-up). Хотя универсального определения агента не существует, обычно агент определяется как объект, имеющий следующие четыре свойства [11]:

- автономия — агент не должен управляться извне: часто он имеет набор собственных правил, которые определяют его поведение;
- реактивность — агент может осознавать свое окружение и реагировать на него;
- социальность — агент может взаимодействовать с другими агентами в модели;
- проактивность — агент может инициировать собственное, определяемое его целью, поведение.

Однако если внимательно рассмотреть эти требования с позиций изложенного в данной книге подхода к компьютерному моделированию, окажется, что агентом может служить активный динамический объект со стереотипом "открытая система", погруженный в составной объект-контейнер, моделирующий окружающую среду. Поэтому подобно изучению модели социальной

системы возможно "экспериментально" изучать законы поведения газов (предположим, что мы забыли все, что изучали в школе) на основе корпускулярной модели газа как совокупности материальных частиц.

Построим в качестве примера мультиагентную модель распространения социального напряжения в регионе [15]. В этой модели регион представляется совокупностью взаимодействующих между собой элементов-индивидуумов. Предполагается, что каждый элемент может находиться в одном из трех состояний:

- возбужденном (E) со степенью напряжения 1;
- полувозбужденном (SE) со степенью напряжения 0.5;
- невозбужденном (UE) со степенью напряжения 0.

Выделяются три типа элементов:

- устойчивый или невнушаемый (kdStable);
- полуустойчивый или полувнушаемый (kdSemistable);
- неустойчивый или внушаемый (kdUnstable).

Тип конкретного элемента и его начальное состояние задаются соответствующими вероятностями. Предполагается, что элемент (индивидуум) может взаимодействовать с восьмью своими соседями: левым, левым верхним, верхним и т. д., передавая друг другу степень социального напряжения (рис. 4.1). Исключительно для удобства рисования связей степень напряжения элемента выведена на все четыре его стороны (выходы L, U, R, B).

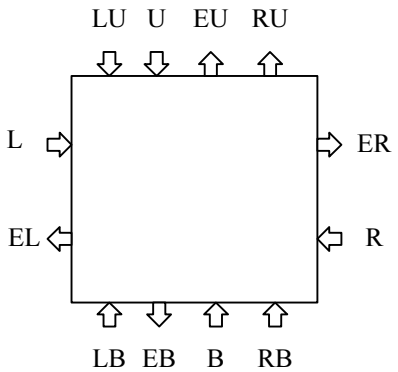



**Рис. 4.1.** Внешний интерфейс элемента

Мы рассмотрим сообщество из шестнадцати элементов — индивидуумов (рис. 4.2). Значением входов, не участвующих в связях, является 0.

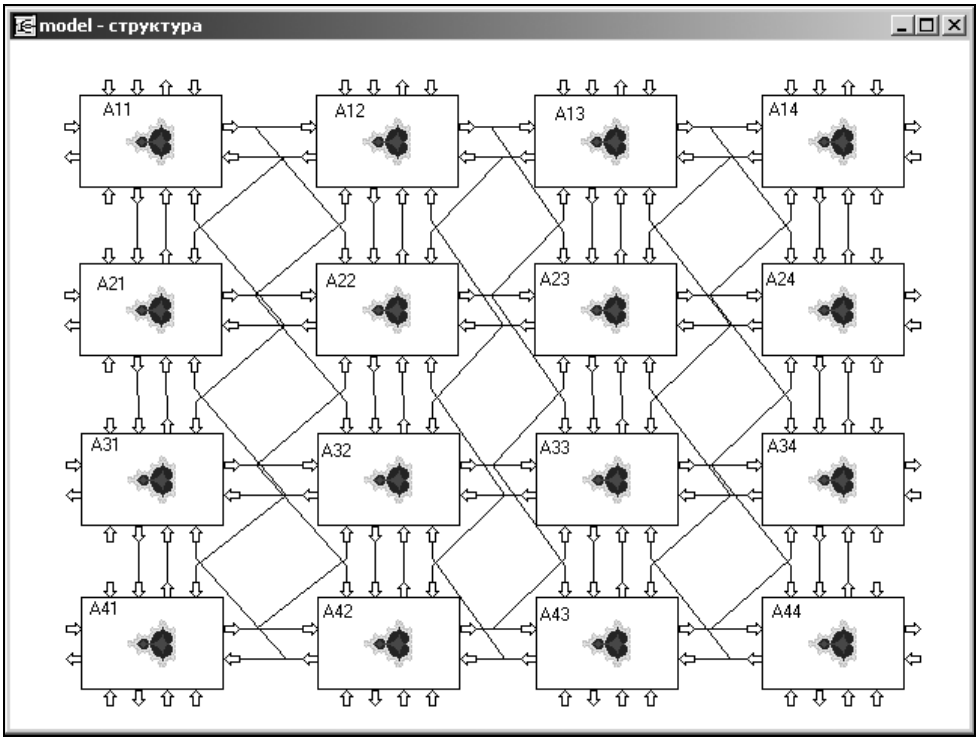



**Рис. 4.2.** Структура модели

На рис. 4.3 показана карта поведений первого уровня для отдельного элемента.

В состоянии `Init` разыгрывается тип данного элемента (рис. 4.4), а затем с помощью вероятностного ветвления выбирается начальное состояние.

Переход из одного состояния в другое происходит по специальному сигналу, вырабатываемому соответствующей данному состоянию картой поведений второго уровня. При входе в какое-либо состояние на выходах элемента устанавливается значение напряжения, соответствующее этому состоянию.

На рис. 4.5—4.7 показаны карты поведений элемента в возбужденном, полувозбужденном и невозбужденном состояниях соответственно. На очередном такте, наступающем через единицу условного модельного времени, происходит вычисление средней степени напряженности соседей на предыдущем такте (переменная `NEX`) и различные детерминированные (ветвление по типу элемента) и случайные (вероятностное ветвление) действия. В результате этих действий элемент либо остается в данном состоянии (переход в состояние второго уровня `Work`), либо происходит переход в конечное состояние и выдается сигнал на переход в другое состояние первого уровня.

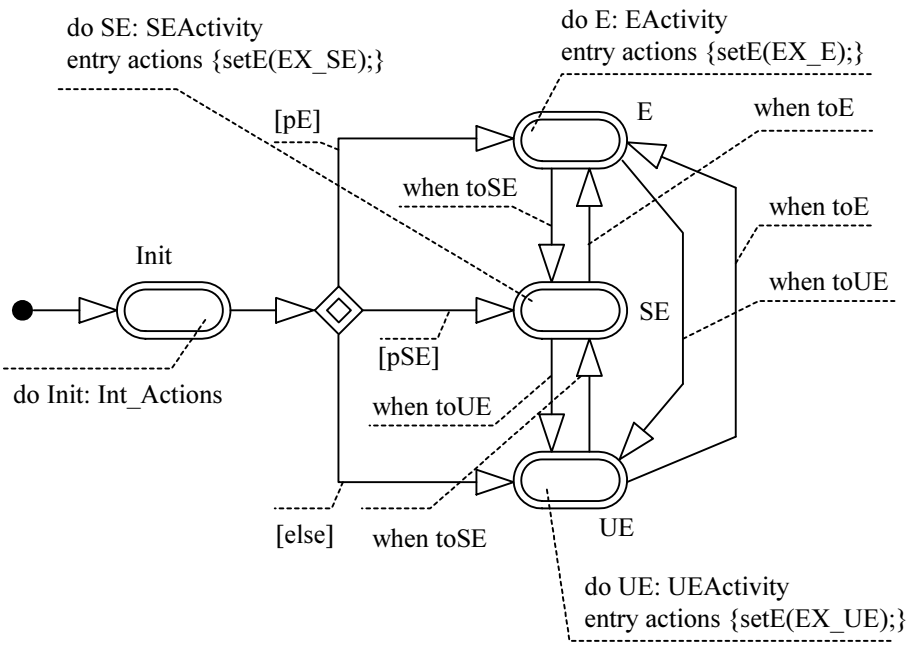



**Рис. 4.3.** Карта поведений элемента

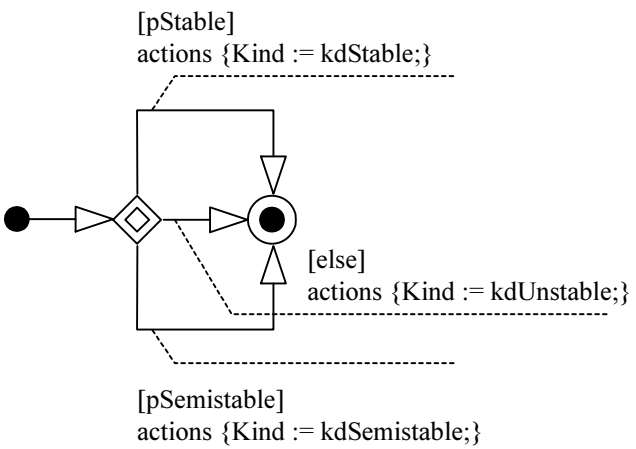



**Рис. 4.4.** Выбор типа элемента

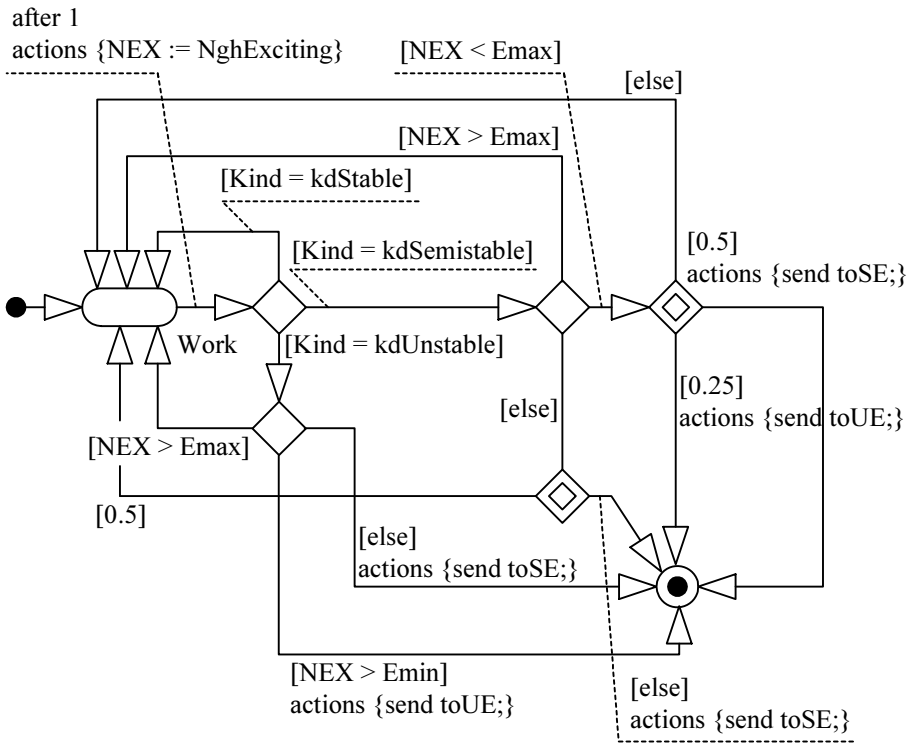



**Рис. 4.5.** Поведение элемента в возбужденном состоянии

Средняя степень напряженности соседей вычисляется по следующему правилу: сумма степеней напряженности всех соседей делится на число реально существующих соседей (для элемента A11 это 3, для элемента A12 — 6, а для элемента A22 — 8). Следует отметить, что согласно принципу синхронной композиции гибридных автоматов [4] значения входов всех элементов автоматически соответствуют завершению предыдущего такта.

На рис. 4.8 и 4.9 приведены результаты вычислительного эксперимента. На рис. 4.8 показано начальное распределение социальной напряженности (с вероятностью 0.2 элемент является устойчивым, с вероятностью 0.2 полуустойчивым и с вероятностью 0.6 неустойчивым, с вероятностью 0.4 начальное состояние является возбужденным, с вероятностью 0.4 полувозбужденным и с вероятностью 0.2 невозбужденным).

На рис. 4.9 показаны распределения социальной напряженности, соответствующие бесконечному циклу, наступающему после 7-го такта — изменяется только значение элемента A22.

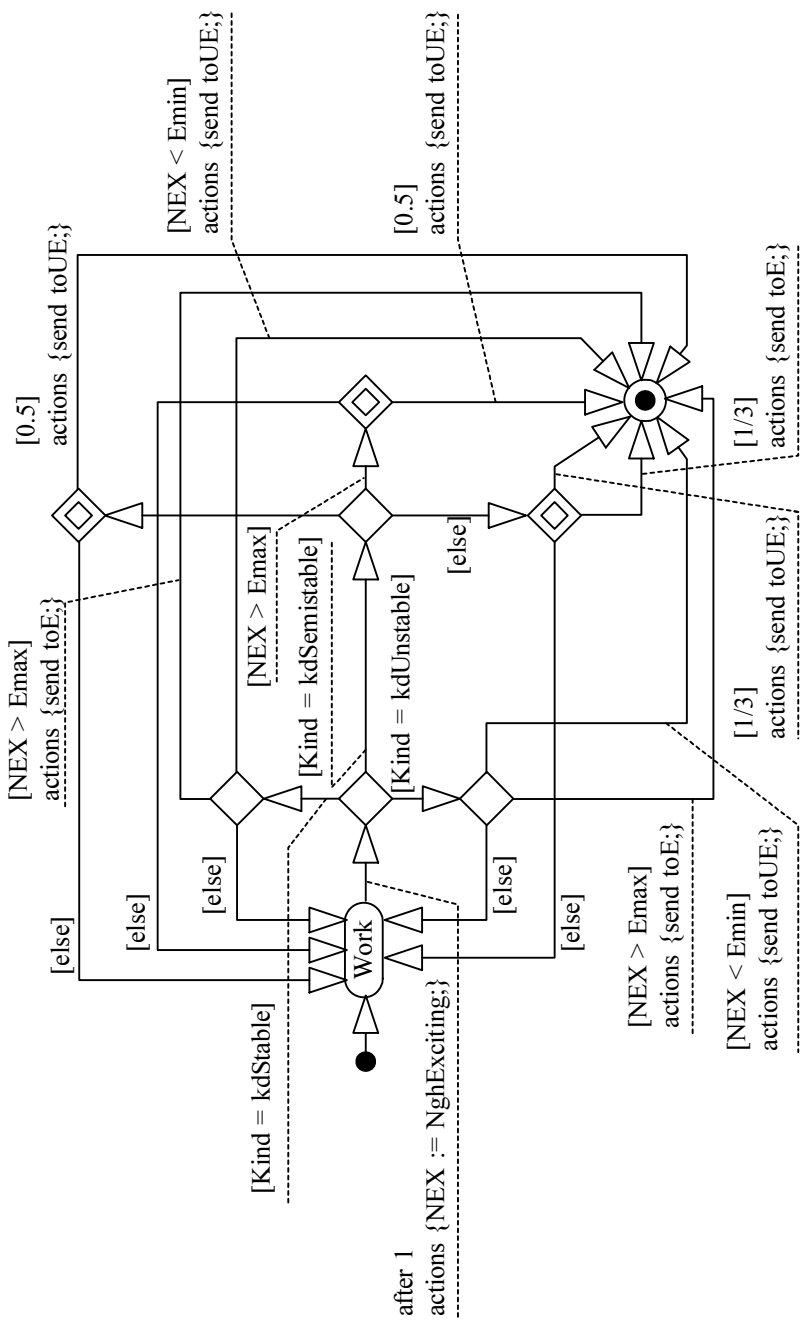



**Рис. 4.6.** Поведение элемента в полувозбужденном состоянии

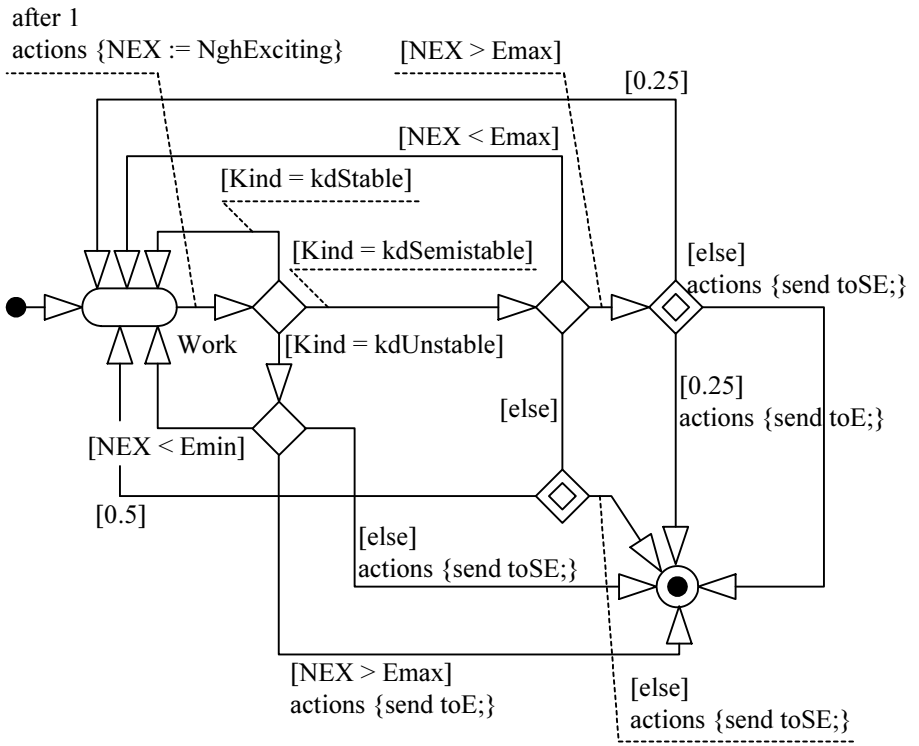



**Рис. 4.7.** Поведение элемента в невозбужденном состоянии

В данной модели, несмотря на ее простоту, есть все признаки мультиагентной модели: существуют автономные объекты с собственным поведением, состояние этих объектов меняется в зависимости от состояния соседних объектов и, в свою очередь, воздействует на состояние этих соседних объектов, объекты функционируют в некоторой среде, которая задает топологию исследуемого региона. Поведение системы в целом определяется суперпозицией поведения совокупности объектов и поведения среды (набора связей). Даже для такой небольшой размерности, как в приведенном примере, результаты эксперимента невозможно предсказать по алгоритмам функционирования отдельного объекта. По существу, это и есть настоящее имитационное моделирование в современной постановке.

Следует отметить, что многоагентное моделирование практически неосуществимо без использования объектно-ориентированного подхода. В приведенном примере совокупность агентов — статическая, и потому используется только один аспект ООМ — отношение "класс — экземпляр". Однако, если вы захотите моделировать реальный регион с тысячами или миллионами элементов-индивидуумов, вам не удастся построить модель среды вручную и придется явно указывать алгоритм соединения элементов, т. е. иметь дело с

переменной структурой. Кроме того, при использовании переменной структуры может оказаться удобным не рассматривать тип элемента как переменную перечислимого типа, а ввести иерархию наследования, показанную на рис. 4.10.


**Рис. 4.8.** Результаты вычислительного эксперимента:
начальное распределение социальной напряженности


*а* *б* *в*

**Рис. 4.9.** Результаты вычислительного эксперимента:
бесконечный цикл после 7-го такта: *а*, *б*, *в* — циклически повторяются состояния

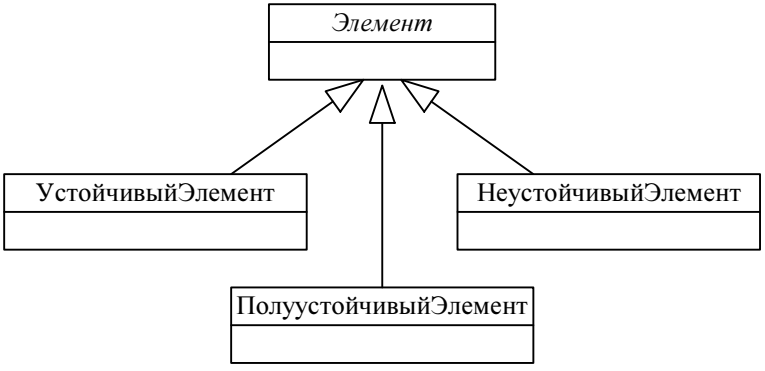



**Рис. 4.10.** Диаграмма классов для модели распространения социального напряжения

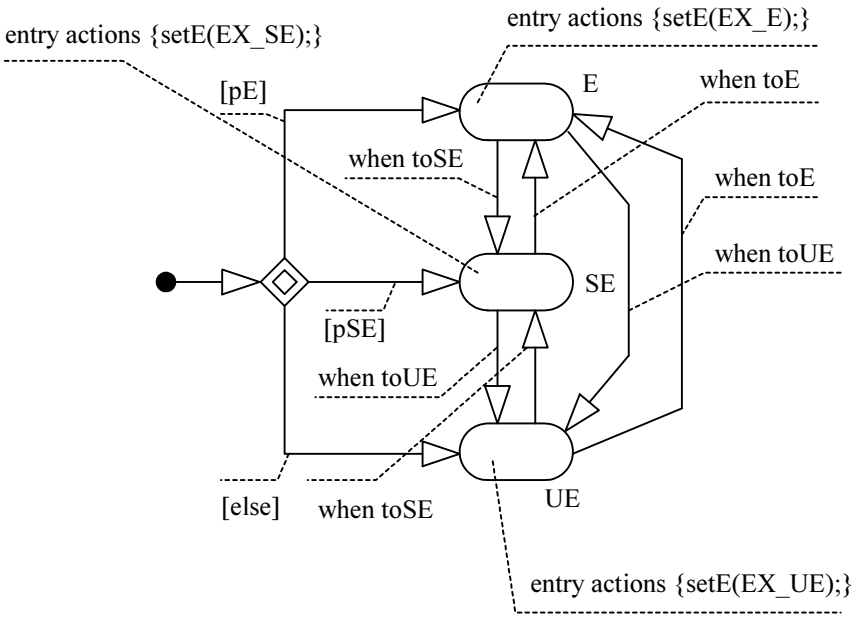



**Рис. 4.11.** Карта поведений абстрактного класса `Элемент`

Абстрактный класс `Элемент` определяет интерфейс элемента (см. рис. 4.1), определения сигналов, функцию вычисления средней напряженности соседей, а также карту поведений первого уровня с пустыми деятельностями в состояниях (рис. 4.11).

Конкретные классы наследуют интерфейс, переменные, функции и процедуры и карту поведений первого уровня от абстрактного класса `Элемент`, дополняют описание поведения определениями локальных классов, задающих карты поведений в определенных состояниях согласно типу элемента (упрощения карт поведений, показанных на рис. 4.5—4.7), и переопределяют деятельности в состояниях первого уровня, заменяя пустые деятельности экземплярами соответствующих локальных классов.

В этом случае множество агентов задается как массив или список объектов класса `Элемент`

```
A: array [1..N,1..M] of Элемент;
```

а при его заполнении элементу массива или списка присваивается с заданной вероятностью экземпляр одного из конкретных классов, показанных на рис. 4.10. Эти действия выполняются в состоянии инициализации карты поведений модели, после чего текущим становится рабочее состояние рис. 4.12.

Карта поведений локального класса `Инициализация` показана на рис. 4.13.

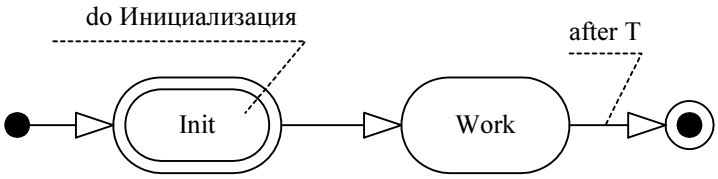



**Рис. 4.12.** Карта поведений модели с $N \times M$ элементами

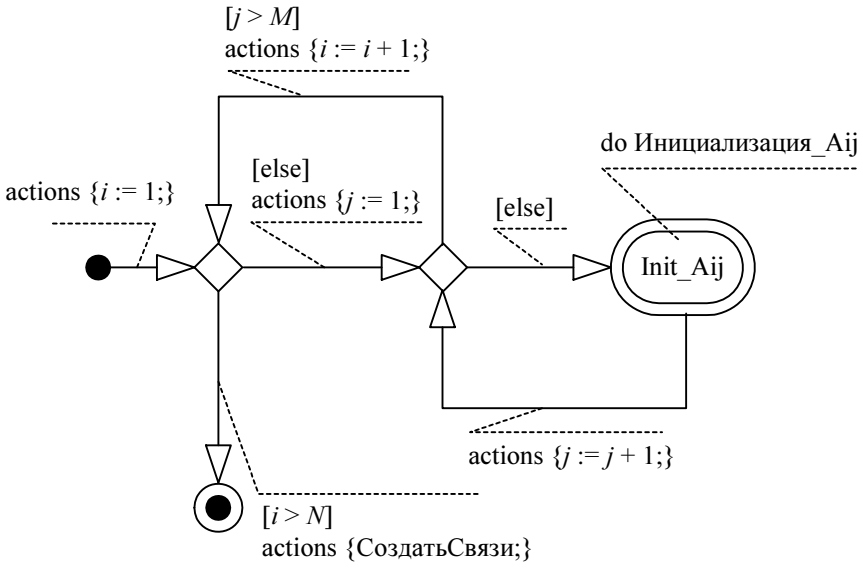



**Рис. 4.13.** Карта поведений локального класса `Инициализация`

Карта поведений локального класса `Инициализация_Aij` показана на рис. 4.14.

В этом случае связи между элементами также должны создаваться динамически:

```
procedure СоздатьСвязи is
begin
  for i in 1..N loop
    for j in 1..M loop
      if (i/=1) and (j/=1) then
        connect(A[i-1,j-1].ER,A[i,j].LU);
      end if;
      if i/=1 then
        connect(A[i-1,j].EB,A[i,j].U);
      end if;
```

```
      if (i/=1) and (j/=M) then
        connect(A[i-1,j+1].EL,A[i,j].RU);
      end if;
      if j/=M then
        connect(A[i,j+1).EL,A[i,j].R);
      end if;
      if (i/=N) and (j/=M) then
        connect(A[i+1,j+1].EL,A[i,j].RB);
      end if;
      if i/=N then
        connect(A[i+1,j].EU,A[i,j].B);
      end if;
      if (i/=N) and (j/=1) then
        connect(A[i+1,j-1].ER,A[i,j].LB);
      end if;
      if j/=1 then
        connect(A[i,j-1].ER,A[i,j].L);
      end if;
    end loop;
  end loop;
end СоздатьСвязи;
```

Таким образом, для сложных мультиагентных моделей используется весь набор возможностей ООМ: наследование, переопределение свойств, полиморфизм.

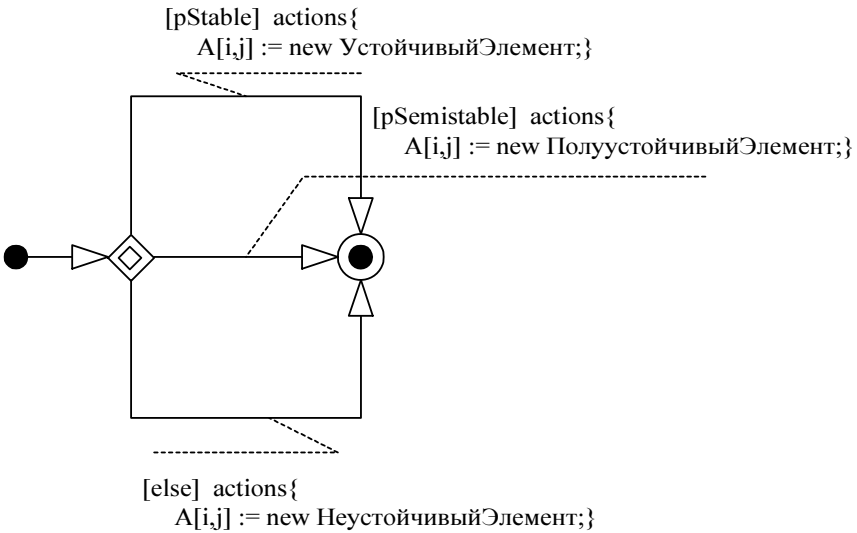
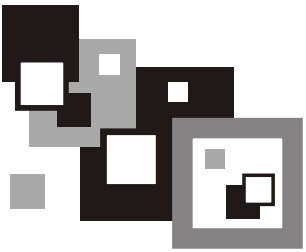



**Рис. 4.14.** Карта поведений локального класса `Инициализация_Aij`

ГЛАВА 5



# Объектно-ориентированное моделирование и объектно-ориентированный анализ

Одной из важных мотиваций использования ООМ является все более широкое применение объектно-ориентированного анализа при разработке сложных технических систем.

## Сложная техническая система

Во второй половине прошлого столетия в ряде областей техники (преимущественно военного направления) появились так называемые *сложные технические системы* или *технические комплексы* [3, 26, 27], к которым, прежде всего, относятся сложные системы управления динамическими объектами. Можно выделить следующие характерные особенности сложных технических систем [3, 20, 22, 27]:

- элементы системы имеют разнородные физические принципы действия (электрические, механические, гидравлические, оптические и другие системы);
- между элементами системы, а также с внешней средой существует множество связей, как информационных, так и физических;
- система имеет иерархическую многоуровневую структуру;
- существует много различных режимов работы, причем эти режимы не совпадают, т. е. один режим работы одной подсистемы может требовать переключений режимов работы других подсистем;
- есть значительная неопределенность в поведении объектов управления и внешней среды;

- ❒ устройства управления помимо задач обычного непрерывного управления решают также задачи логического управления, диагностики и др.;
- ❒ подавляющая часть функций управления реализуется программно на встроенных ЭВМ и микропроцессорах;
- ❒ очень часто программное обеспечение и аппаратура разрабатываются одновременно;
- ❒ часто состав и структура системы изменяется в ходе ее функционирования.

К "традиционным" сложным техническим системам относятся ракетные и космические комплексы, комплексы противовоздушной и противоракетной обороны, некоторые АСУ ТП и др. В последнее десятилетие роль сложных технических систем резко возросла. Благодаря прогрессу микроэлектроники появились дешевые, надежные и быстродействующие встроенные микропроцессоры и ЭВМ. Это привело, во-первых, к усложнению алгоритмов управления и контроля в "традиционных" сложных системах, а во-вторых, к появлению программной реализации функций управления и контроля во все большем числе технических объектов. Некоторые характерные черты сложных систем появились даже в таких "бытовых" технических системах, как автомобиль, стиральная машина, микроволновая печь и т. п. Соответственно расширился и круг инженеров-проектировщиков, занятых разработкой и сопровождением сложных технических систем.

В современных сложных технических системах значительная доля трудоемкости разработки приходится на разработку программного обеспечения (ПО) встроенных ЭВМ и микропроцессоров [19, 24, 25, 28]. Многочисленные ошибки в этом ПО приводят к затягиванию этапов динамической комплексной отладки и испытаний, а также к неожиданным отказам системы во время эксплуатации. Эти ошибки обусловлены, прежде всего, логической сложностью комплекса программ, не случайно число изменений в программных модулях, координирующих работу подсистем, на порядок превышает число изменений в модулях, реализующих отдельные функции [24, 25]. С 1970-х гг. активно разрабатываются методологии структурного проектирования сложных программных комплексов, такие как SADT, IDEF, метод Йордана и др. С конца 1980-х гг. начали также интенсивно развиваться объектно-ориентированные методологии разработки программного обеспечения. В настоящее время объектно-ориентированный подход считается наиболее современным и прогрессивным. В 1997 г. OMG (Object Management Group) приняла язык UML, появившийся в результате слияния ряда известных методологий, в качестве стандарта языка объектно-ориентированного моделирования (в данном случае это скорее "прототипирование") при разработке ПО. В настоящее время уже существуют разработанные рядом компаний CASE-

средства, поддерживающие язык UML, например, такой известный продукт, как Rational Rose. Существуют также и отличные от UML объектно-ориентированные методологии, например, методология ROOM для разработки систем реального времени [17], а также различные комбинации структурного и объектного подходов [28]. В данной книге мы ориентируемся на понятийный аппарат языка UML.

Практика показала, что при разработке сложного программного обеспечения самые принципиальные просчеты делаются на самых ранних этапах разработки и что обнаружение и устранение этих ошибок на ранних этапах в десятки и сотни раз выполняются быстрее и дешевле, чем на завершающих этапах разработки и испытаний [24]. Поэтому в объектно-ориентированной разработке сложных технических систем особенно важен этап объектно-ориентированного анализа. Последний определяет так много важных требований к инструментальным средствам ООМ, что мы рассмотрим его подробней.

## Объектно-ориентированный анализ при разработке сложных технических систем

Объектно-ориентированный анализ в основном должен выполняться на стадиях НИР и ОКР. В идеальном случае анализ должен выполняться один раз (так называемая "водопадная" модель процесса проектирования), однако все же более жизненной является "итеративная" модель, допускающая возврат к анализу системы на последующих этапах разработки с целью уточнения спецификаций системы [16, 19]. Целью объектно-ориентированного анализа является [18, 19]:

- выявление объектов и их связей, т. е. функциональной структуры системы;
- определение желаемого поведения системы в основных режимах работы, так называемых "сценариев", в возможно более формальном виде;
- выделение классов объектов и отношений между классами;
- определение границы между аппаратной и программной составляющими системы.

Совокупность результатов объектно-ориентированного анализа в работе [19] называется *аналитической моделью* проектируемой системы, а процесс ее создания — *аналитическим моделированием*. Собственно, аналитическая модель и есть единственная модель, которую имеет смысл анализировать при разработке сложной СУ, поскольку модели более низких уровней абстракции сравнимы по трудоемкости создания с разработкой самой системы. Поскольку слово "аналитический" часто понимается как синоним слова "символьный", мы будем далее использовать термин "системно-аналитический".

Практически невозможно указать единый принцип выделения компонентов системы и соответствующих им классов, пригодный для любой системы. Обычно при выделении компонентов руководствуются либо физической структурой системы, либо функциональными ролями элементов системы [18, 19]. Типичным можно считать случай, когда в начале разработки сложной технической системы имеется некоторый набор уже готовых аппаратных блоков (типовых или переносимых из предыдущих разработок с небольшими модификациями), некоторый набор аппаратных блоков разрабатывается вместе с системой, а архитектуру управляющего цифрового вычислительного комплекса (УЦВК) еще предстоит разработать (рис. 5.1).

На рис. 5.1 заданные физические подсистемы ("приборы") показаны закрашенными прямоугольниками, толстыми линиями показаны физические связи между ними, тонкими линиями — информационные связи. Заметим, что "физические" подсистемы могут сами содержать свои внутренние встроенные ЭВМ и/или микропроцессоры с "прошитым" собственным ПО. Однако если это ПО не является предметом разработки, то вся подсистема рассматривается как аппаратная.

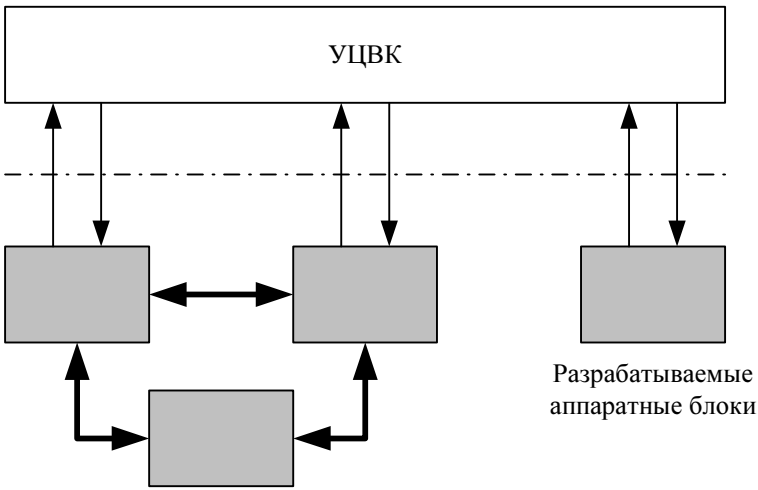



**Рис. 5.1.** Физическая структура проектируемой системы

Таким образом, на самых ранних этапах проектирования сложной системы, как правило, существует значительная неопределенность в физической структуре системы и конкретных аппаратных решениях. Так называемое "сопроектирование" аппаратной и программной составляющих (HW/SW Codesign) является характерной особенностью процесса разработки значительной

части современных сложных систем. Окончательная граница раздела "программы — аппаратура" может сама являться одним из результатов анализа и неоднократно уточняться позднее. Поэтому представляется более предпочтительным при выделении компонентов системы руководствоваться их функциональными ролями.

Методология объектно-ориентированного проектирования рекомендует на стадии объектно-ориентированного анализа выделять только активные компоненты, функционирующие независимо и параллельно [19]. Применительно к сложным техническим системам наиболее естественным представляется рассматривать в качестве основного при выделении активных объектов отношение "объект управления (ОУ) — устройство управления (УУ)". Аппаратные объекты управления ("внешние" объекты в терминологии UML [18, 19]) являются активными по определению. Активные программные объекты являются устройствами управления соответствующего уровня. Совокупность функционирующих независимо и параллельно взаимодействующих активных объектов отражает функциональную структуру проектируемой системы (рис. 5.2).

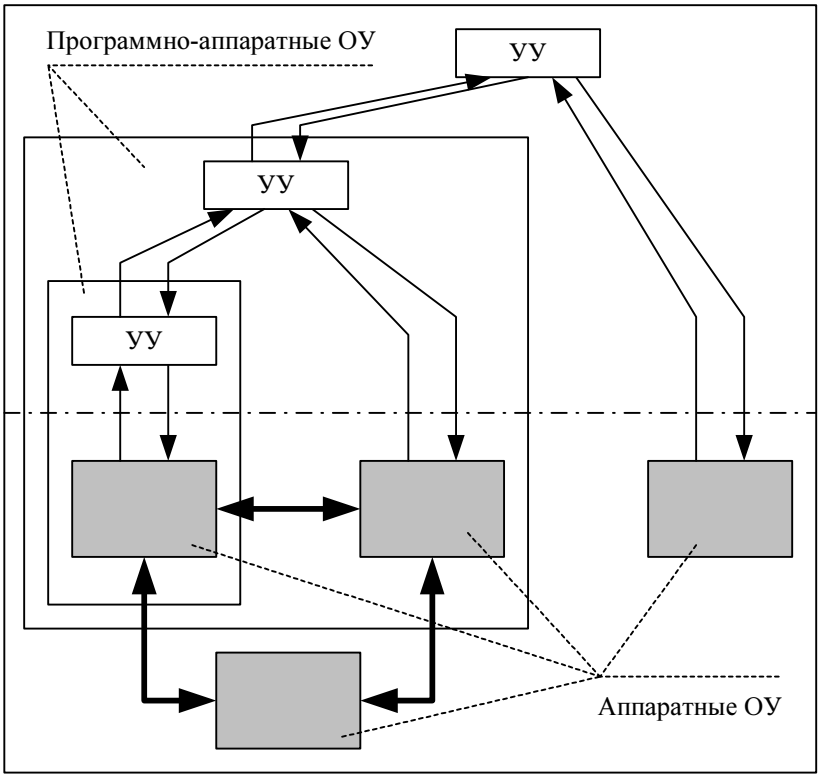



**Рис. 5.2.** Функциональная структура проектируемой системы

При выделении объектов "по управлению" часть составных объектов становится программно-аппаратными. Программно "надстраивая" аппаратуру, мы получаем более высокоуровневую и "интеллектуальную" подсистему. Такие подсистемы соответствуют тому, что иногда называется *виртуальным устройством* (control configured vehicle). Вообще все внешние объекты на этапе объектно-ориентированного анализа можно рассматривать как программно-аппаратные, поскольку точный интерфейс взаимодействия с реальным физическим объектом может быть на этом этапе просто неизвестен. На последующих этапах проектирования можно уточнить этот внешний объект, используя отношение наследования, и выделить его программную составляющую. Иногда физическая подсистема по разным соображениям интегрирует в себе несколько функциональных аспектов. В этом случае в качестве локального объекта управления в функциональной подсистеме целесообразно использовать соответствующий интерфейс физического объекта (рис. 5.3).

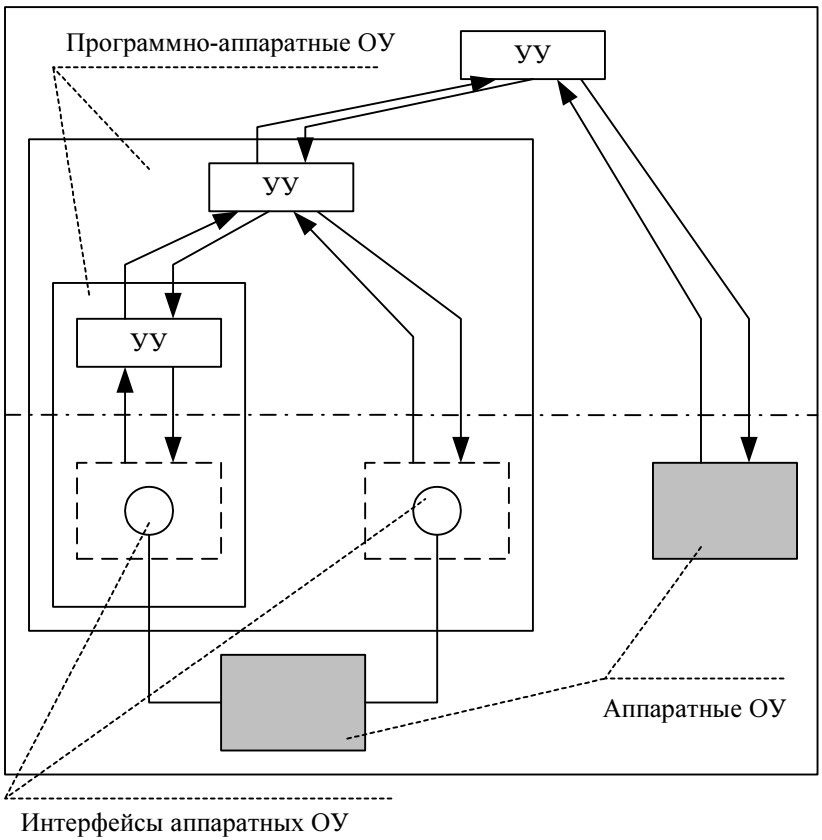



**Рис. 5.3.** Использование интерфейсов объектов

"Традиционные" задачи непрерывного управления (регулирования) обычно характерны для нижних уровней иерархии сложных систем. На верхних уровнях иерархии обычно имеют место процессы так называемого логического управления. Например, УУ верхнего уровня автоматического сверлильного станка осуществляет координацию согласованной работы подсистем управления нижнего уровня: включает вращение шпинделя, включает вертикальную подачу, по контакту сверла с деталью включает подачу охлаждающей жидкости и отключает ее при исчезновении контакта, при достижении заданной глубины отверстия меняет направление вертикальной подачи, контролирует поломку сверла и т. д. В то же время УУ нижних уровней регулируют значения отдельных показателей, например, поддерживают заданную скорость вращения шпинделя.

В руководствах по объектно-ориентированному анализу желаемое поведение рекомендуется определять в виде сценариев для основных режимов работы системы. В первую очередь рекомендуется составлять "первичные" сценарии, отражающие нормальную работу системы. Затем должны быть разработаны "вторичные" сценарии, отражающие реакцию системы на различные исключительные ситуации (отказ аппаратуры, неверные исходные данные, отмена режима оператором и т. п.). Для задания сценариев могут быть использованы две конструкции UML: диаграмма взаимодействий и диаграмма состояний.

Один из видов диаграммы взаимодействий — *диаграмма последовательностей* — прекрасно знакома большинству разработчиков алгоритмов функционирования сложных систем как "циклограмма режима". Диаграмма последовательностей предназначена для фиксации временной последовательности событий в системе. В верхней части диаграммы по оси $x$ указываются объекты, участвующие во взаимодействии, причем инициирующие взаимодействие объекты располагаются левее, а подчиненные правее. Применительно к сложной технической системе это соответствует иерархии уровней управления. По оси $y$ размещаются пунктирные "линии жизни" объектов во времени и показывается стрелками передача сообщений от одного объекта к другому (более поздние показываются ниже). Кроме того, для программных объектов на "линиях жизни" вытянутыми прямоугольниками может быть показан "фокус управления". Применительно к активным объектам "фокус управления" не особенно актуален, поэтому можно обозначить условные переходные процессы, вызванные поступившими сообщениями. Диаграммы последовательностей особенно наглядны для процессов логического управления. Сообщения, поступающие "слева направо" от контроллера к объекту управления ассоциируются с командами, а сообщения, поступающие "справа налево" — с донесениями.

Другим видом диаграмм взаимодействий являются *диаграммы кооперации*, на которых изображаются объекты, участвующие во взаимодействии, и их

связи с указанием передаваемых сообщений. Для СУ это по существу фрагмент структурной схемы. Диаграммы взаимодействий безусловно важны для самой предварительной проработки сценариев работы системы. Вообще говоря, само выделение набора программно реализуемых объектов и их связей практически возможно только после нескольких итераций построения сценариев. Однако эти диаграммы являются неформальными, семантика выполняемых действий задается на уровне комментариев.

Для однозначного понимания возможности объективного тестирования желаемого поведения, а также для последующего использования в качестве имитаторов на стадии проектирования необходима формальная математическая модель, отражающая структуру и поведение разрабатываемой системы на самом высоком уровне абстракции. Кроме того, в процессе анализа и сравнения вариантов могут потребоваться модели отдельных подсистем, причем различных уровней детализации. Анализ предполагает оценку правильности принимаемых решений. Для сложных систем такую оценку можно получить только путем проведения вычислительных экспериментов с компьютерной моделью системы по разработанным сценариям поведения. На этапе объектно-ориентированного анализа должны решаться также задачи синтеза и оптимизации. Для решения этих задач также необходимы математические и компьютерные модели разрабатываемой системы.

Единственная формальная конструкция UML — это *диаграмма состояний*, которая является чуть измененной картой состояний Харела. Карта состояний представляет собой направленный граф, вершины которого связываются с качественными состояниями системы, а дуги — с переходами из одного состояния в другое. Переход может произойти по истечении некоторого интервала времени, по выполнении логического условия и по дискретному событию (сообщению, исключительной ситуации и т. п.). Карта состояний является чрезвычайно удобным и наглядным инструментом для описания взаимодействующих дискретных параллельных процессов, развивающихся в непрерывном времени. Функционирование программной составляющей системы можно описать с помощью карт состояния достаточно адекватно. Иногда с помощью карты состояний можно достаточно адекватно описать и поведение аппаратных объектов управления. Например, в [28] показано успешное использование несколько модифицированной карты состояний для описания системы управления телефонной АТС. Однако карты состояний недостаточно для описания поведения непрерывных объектов управления. Возможна, конечно, "дискретная аппроксимация" непрерывного поведения объектов управления, когда оно заменяется в упрощенных моделях на совокупность чисто дискретных переходных процессов, являющихся реакцией на определенные управляющие воздействия.

В принципе такой подход, конечно, допустим, но имеет ряд серьезных недостатков. Полученную в результате "дискретной аппроксимации" модель очень трудно модифицировать в случае, когда потребуется более детальное рассмотрение динамики процесса (например, для исследования перекрестного влияния подсистем). При дискретной "аппроксимации" достаточно сложно имитировать ошибочные ситуации для отработки "вторичных" сценариев поведения. На взгляд автора правильнее непрерывные объекты всегда представлять непрерывными моделями, пусть даже чрезвычайно упрощенными (например, полагать систему линейной).

Легко видеть, что рассматриваемая в книге [16] сложная динамическая система является обобщением свойств сложных технических систем.

Таким образом, для полноценного выполнения объектно-ориентированного анализа при разработке сложной технической системы необходима объектно-ориентированная компьютерная модель всей проектируемой системы в целом. Создание такой модели вручную практически невозможно из-за огромных трудозатрат и ненадежности результатов экспериментов с такой моделью. Опыт показывает, что программирование компьютерных моделей вручную возможно только для относительно стабильных подсистем, и отработка таких моделей занимает годы. Следовательно, объектно-ориентированный анализ сложных СУ невозможен без средств автоматизации объектно-ориентированного моделирования СДС. Очевидно также, что инструмент системно-аналитического моделирования должен относиться к "исследовательскому" типу.

Имеется еще один аспект применения инструментальных средств системно-аналитического моделирования — техническое образование. Проектирование современных сложных технических систем требует помимо фундаментального и технического образования еще и некоторой минимальной инженерной практики, которая традиционно приобретается в первые годы работы молодого специалиста на предприятии. Предприятие по существу получает "полуфабрикат" инженера-проектировщика и "доводит" его до профессионального уровня. Кроме того, опыт показывает, что далеко не все инженеры склонны и способны к комплексному проектированию сложных систем. Частично такая практика может приобретаться уже в вузе, если студентам предоставляется возможность в ходе учебного проектирования создавать на персональном компьютере визуальные макеты сложной системы. Возможность сразу наблюдать поведение созданной системы замыкает обратную связь в учебном процессе и учит находить инженерные компромиссы. Таким образом, появляется возможность выявлять студентов, склонных к работе проектировщика.

# Объектно-ориентированное моделирование на последующих этапах разработки и сопровождения сложной технической системы

Таким образом, системно-аналитическое моделирование на ранних этапах проектирования позволяет разработать функциональную структуру и алгоритмы функционирования разрабатываемой системы и отладить их в ходе серии вычислительных экспериментов. Далее системно-аналитическая модель становится формальной спецификацией для дальнейшего проектирования технической системы на более низких уровнях детализации. На основании выбранной функциональной структуры принимаются решения по разработке аппаратуры и системного программного обеспечения УЦВК. На основании отлаженных алгоритмов функционирования разрабатывается функциональное программное обеспечение для встроенных ЭВМ или микропроцессоров. Отладка программного обеспечения производится на комплексном моделирующем стенде.

Комплексный моделирующий стенд (КМС) [27] включает в себя компьютерные модели и некоторые реальные устройства разрабатываемой системы управления. Такие модели часто называют также HIL-моделями (Hardware In the Loop) [16] или полунатурными. Поскольку в данной книге любая независимо функционирующая компьютерная модель трактуется как физическое устройство — имитатор [16], то на взгляд авторов комплексный моделирующий стенд следует считать частным случаем распределенной модели, в которой некоторые элементы являются реальными аппаратными блоками системы управления.

В процессе проектирования наиболее часто используются две типовые схемы комплексного моделирующего стенда:

- устройства управления и некоторые объекты управления моделируются, а некоторые объекты управления представлены физическими устройствами (рис. 5.4);
- некоторые устройства управления представлены реальными встроенными ЭВМ с соответствующим программным обеспечением, а остальная часть системы управления моделируется (рис. 5.5).

В обеих этих схемах "разрезаются" только дискретные связи.

Комплексный моделирующий стенд как распределенная модель всегда имеет, как минимум, два компонента: компьютерную модель на инструментальной ЭВМ и адаптер для реального УЦВК, поддерживающий интерфейс распреде-

ленного выполнения. Инфраструктура комплексного моделирующего стенда помимо функций распределенного выполнения должна обеспечивать привязку к реальному или кусочно-реальному (программное обеспечение встроенной ЭВМ может выполняться в режиме отладки) времени [16]. Таким образом, моделирование гибридной системы в составе комплексного моделирующего стенда можно считать специальным случаем распределенного моделирования гибридной системы, рассмотренного выше.

Далее проектируемая система проходит натурные испытания, и начинается этап эксплуатации и сопровождения, длящийся до прекращения эксплуатации системы (часто этот этап охватывает несколько десятилетий).

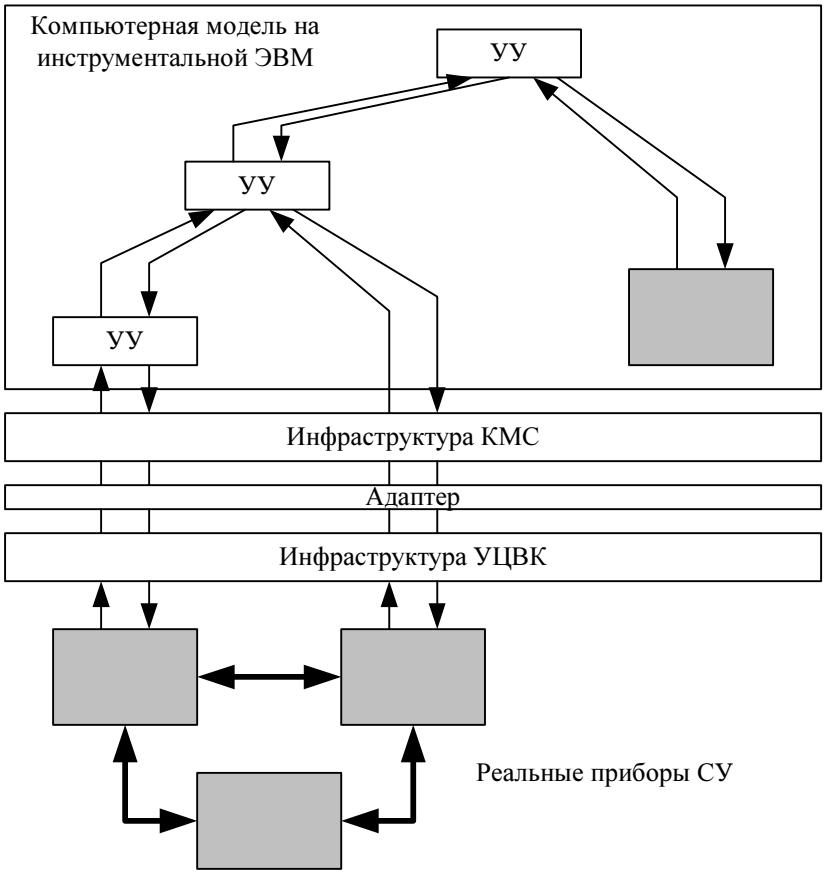



**Рис. 5.4.** Первая типовая схема КМС

На этапе сопровождения в программное и аппаратное обеспечение вносится огромное число изменений, обусловленных выявленными ошибками, а также

изменением предъявляемых требований и сценариев желаемого поведения. Эти изменения затрагивают как функциональную структуру системы, так и алгоритмы ее функционирования. Чрезвычайно важно на всех этапах жизненного цикла технической системы сохранять соответствие спецификаций (в данном случае системно-аналитической модели) и реальных программных и аппаратных решений нижнего уровня. На достижение этой цели направлены различные комплексы организационных и технических решений, часто

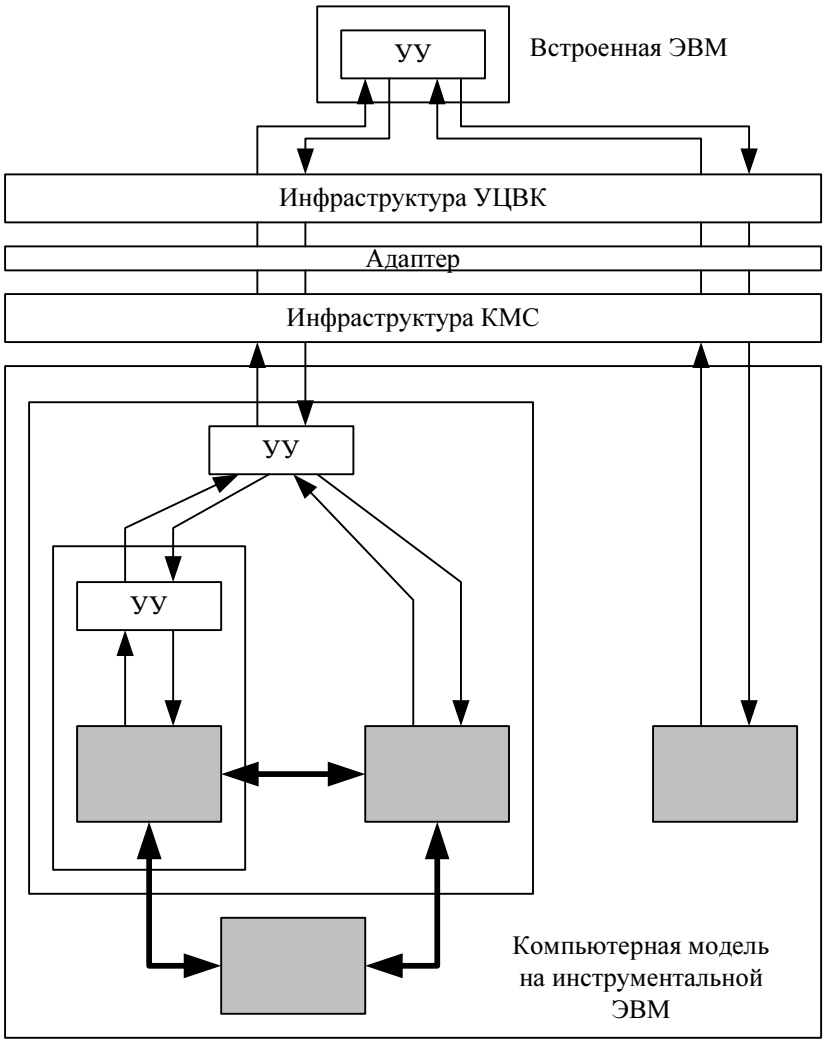



**Рис. 5.5.** Вторая типовая схема КМС

называемых *сквозными технологиями* [24, 25]. Однако опыт внедрения этих технологий говорит о том, что если технология требует от разработчика каких-либо ручных дополнительных действий, то, несмотря на любое административное давление, соответствие спецификаций и решений рано или поздно нарушается. Между тем метод автоматического синтеза выполняемых моделей гибридных систем [3] позволяет говорить о возможности создания новой "сквозной" технологии проектирования сложных систем управления.

## Системно-аналитическая модель как основа "сквозной" технологии проектирования

Основная идея этой технологии заключается в следующем. Объектно-ориентированную математическую модель системы управления, создаваемую на этапе анализа, можно рассматривать как описание (спецификацию) проектируемой системы на некотором формальном языке сверхвысокого уровня. Любая компьютерная модель, однозначно соответствующая этой математической модели, является ее физической реализацией. На этапе анализа эта компьютерная модель строится целиком на базе инструментальной ЭВМ и ее системного программного обеспечения (например, на платформе Intel — Windows).

На последующих этапах разработки могут использоваться различные промежуточные распределенные компьютерные модели в составе КМС. На рис. 5.6 показана распределенная модель сложной технической системы, в которой все устройства управления представлены компьютерными моделями, а объекты управления — реальными штатными объектами.

Предполагается, что распределение устройств управления по отдельным компьютерным моделям соответствует их штатному распределению по встроенным ЭВМ. Кроме того, в этой модели инфраструктура КМС не использует свою локальную сеть, и все обмены данными идут через штатную магистраль УЦВК. Эта система является физической и при достаточной производительности инструментальной ЭВМ может имитировать систему управления (она не может являться штатной только из-за несоответствия инструментальных ЭВМ условиям применения СУ).

На рис. 5.7 показана реальная система, в которой устройства управления реализованы программно на встроенных ЭВМ. Легко видеть, что рассмотренная выше модель системы отличается от реальной системы только использованием инструментальной ЭВМ и инфраструктуры КМС. Предположим, что:

- все функции управления реализуются программно на встроенных ЭВМ;
- эти встроенные ЭВМ достаточно производительны;

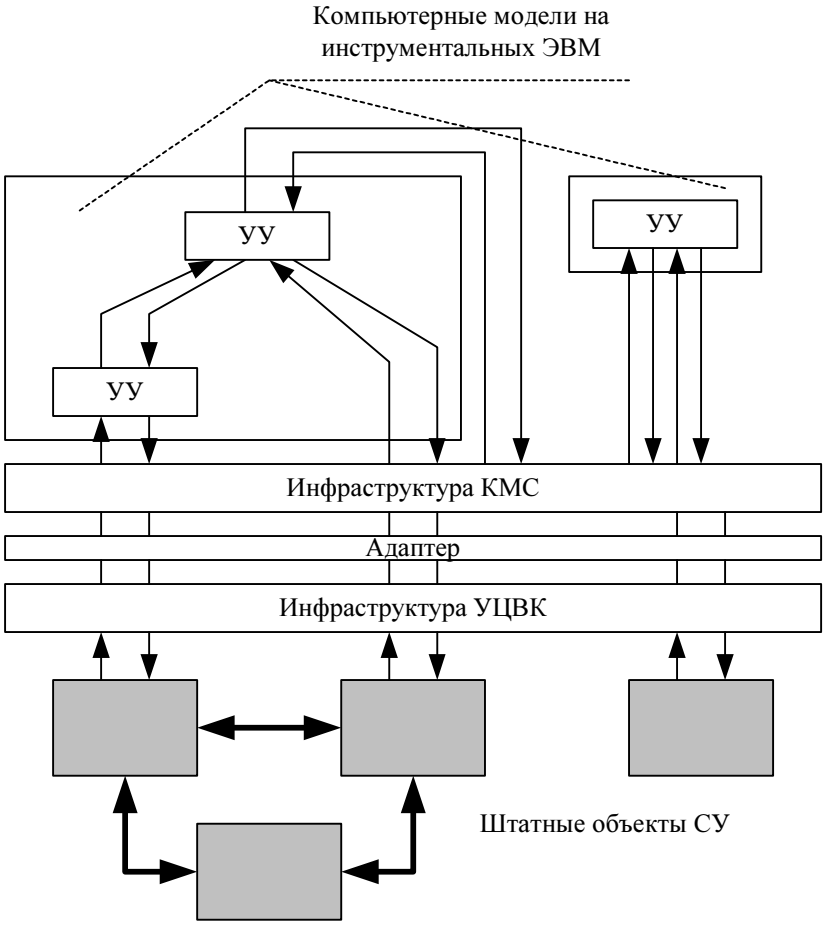



**Рис. 5.6.** Распределенная модель, соответствующая реальной структуре проектируемой системы

- существуют некоторые кросс-компиляторы, поддерживающие для этих ЭВМ какой-либо из объектно-ориентированных языков программирования;
- пакет моделирования включает генератор "кода" выполняемой модели для этого языка;
- создан вариант исполняющей системы пакета моделирования для этого языка и используемой операционной системы реального времени;
- интерфейсы инфраструктур КМС и УЦВК совпадают.

В этом случае ручную разработку прикладного программного обеспечения для встроенных ЭВМ можно просто исключить и использовать вместо него

выполняемые модели устройств управления, генерируемые автоматически по их математическим моделям. Язык системно-аналитического моделирования будет выступать в роли языка программирования сверхвысокого уровня для встроенных ЭВМ. Например, для программирования регулятора достаточно будет просто задать соответствующую передаточную функцию или систему нелинейных уравнений, а численная реализация будет построена автоматически. Программирование устройства логического управления сведется к рисованию соответствующей карты поведений. Собственно программирование для встроенной ЭВМ будет включать в себя разработку операционной системы реального времени (если не используется стандартная), разработку средств поддержки инфраструктуры УЦВК для штатной магистрали обмена и разработку специальной исполняющей системы для пакета моделирования. Попытка создания такой многоуровневой структуры программ для одного частного случая была предпринята в работе [23].

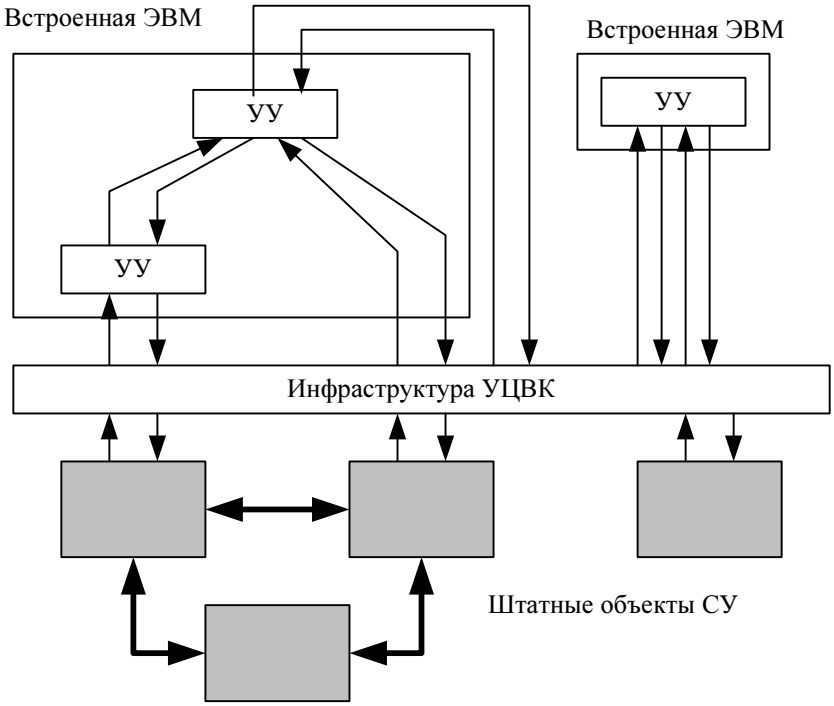



**Рис. 5.7.** Штатная структура проектируемой системы

Естественно, автоматически сгенерированное объектно-ориентированное прикладное программное обеспечение в среднем будет работать существенно

медленнее, чем созданное вручную с использованием процедурных языков программирования. Однако на протяжении последних 15 лет производительность ЭВМ увеличивалась примерно в 100 раз за каждые 5 лет [24] и уже достигла или достигнет в недалеком будущем уровня, когда этой разницей можно будет пренебречь. Лозунгом любой "сквозной" технологии может служить подзаголовок книги [16]: "Build Better Embedded Systems Faster". Использование для разработки прикладного программного обеспечения встроенных ЭВМ языка сверхвысокого уровня позволит принципиально сократить сроки разработки системы управления и повысить ее надежность.

# Литература


1. Бенькович Е. С., Колесов Ю. Б., Сениченков Ю. Б. Практическое моделирование сложных динамических систем. — СПб.: БХВ-Петербург, 2002. — 464 с.
2. Буч Г., Якобсон А., Рамбо Дж. UML: 2-е издание. — СПб.: ПИТЕР, 2006. — 736 с.
3. Колесов Ю. Б. Объектно-ориентированное моделирование сложных динамических систем. — СПб.: Изд-во СПбГПУ, 2004. — 239 с.
4. Колесов Ю. Б., Сениченков Ю. Б. Моделирование систем. Динамические и гибридные системы. — СПб.: БХВ-Петербург, 2006. — 224 с.
5. Майо Д. C#: Искусство программирования. Энциклопедия программиста: Пер. с англ. — СПб.: ДиаСофтЮП, 2002. — 656 с.


## Дополнительная литература к главе 1


6. Дал У.-И., Мюрхауг Б., Июгорд К. СИМУЛА-67. Универсальный язык программирования. — М.: Мир, 1969. — 99 с.
7. Леоненков А. В. Самоучитель UML: 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 432 с.


## Дополнительная литература к главе 2


8. ANSI/IEEE Std 754-1985. IEEE Standard for Binary Floating-Point Arithmetic, 1985.
9. Gollu A., Kourjanski M. Object-oriented design of automated highway simulators using SHIFT programming language.

   **http://www.path.berkeley.edu/shift/publications.html**

10. Ptolemy II. Heterogeneous concurrent modeling and design in java. Edited by: Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun, Liu, Steve Neuendorffer, Yuhong Xiong, Haiyang Zheng. // Department of Electrical Engineering and Computer Sciences University of California at Berkeley.. Document Version 2.0.1 for use with Ptolemy II 2.0.1, August 5, 2002.

**http://ptolemy.eecs.berkeley.edu**

11. Карпов Ю. Г. Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. — СПб.: БХВ-Петербург, 2005. — 400 с.

## Дополнительная литература к главе 3

12. Глушков В. М., Гусев В. В., Марьянович Т. П., Сахнюк М. А. Программные средства моделирования непрерывно-дискретных систем. — Киев: Наукова думка, 1975. — 152 с.

13. Попов Ю. П., Самарский А. А. Вычислительный эксперимент // В сб. Компьютеры, модели, вычислительный эксперимент. Введение в информатику с позиций математического моделирования. — М.: Наука, 1988. — 176 с.

## Дополнительная литература к главе 4

14. Joshua M. Epstein and Robert L. Axtell. Growing Artificial Societies Social Science From the Bottom Up. — Brookings Institution Press MIT Press, 1996. — 228 pp.

15. N. Gilbert and K.G. Troitzsch, Simulation for the Social Scientist. — Buckingham, UK: Open University Press, 1999. — 159 pp.

## Дополнительная литература к главе 5

16. Ledin J. Simulation Engineering. CMP Books, Lawrence, Kansas, 2001.

17. Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. — John Wiley & Sons. Inc., 1994.

18. Буч Г. Объектно-ориентированный анализ и проектирование с примерами на C++, 3-е изд. / Пер. с англ. — М.: Бином, СПб.: Невский диалект, 2001. — 560 с.

19. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений: Пер. с англ. — М.: ДМК Пресс, 2002. — 704 с.

20. Дмитриев А. К., Мальцев П. А. Основы теории построения и контроля сложных систем. — Л.: Энергоатомиздат, 1988. — 192 с.

21. Ефимов К. Социальное напряжение и катастрофические события. // Наука — Культура — Общество. — 2005. — № 3. — С. 31—40.

22. Касти Дж. Большие системы. Связность, сложность и катастрофы. — М.: Мир, 1982. — 216 с.

23. Курочкин Е. П., Колесов Ю. Б. Технология программирования сложных систем управления / ВМНУЦ ВТИ ГКВТИ СССР. — М.: 1990. — 112 с.

24. Липаев В. В. Надежность программных средств. — М.: Синтег, 1998. — 232 с.

25. Липаев В. В. Системное проектирование сложных программных средств для информационных систем. — М.: Синтег, 1999. — 224 с.

26. Меерович Г. А. Эффект больших систем. — М.: Знание, 1985. — 231 с.

27. Подчуфаров Ю. Б. Физико-математическое моделирование систем управления и комплексов. / Под ред. А. Г. Шипунова. — М.: Изд-во физико-математической литературы, 2002. — 168 с.

28. Терехов А. Н., Романовский К. Ю., Кознов Дм. В., Долгов П. С., Иванов А. Н. Объектно-ориентированная методология разработки информационных систем и систем реального времени. // Объектно-ориентированное визуальное моделирование. / Под ред. проф. А. Н. Терехова. — СПб.: Изд-во Санкт-Петербургского университета, 1999. — С. 4—20.

# Предметный указатель

## М



## Н



## О



## П



## С



## Т

## У



## Ф



## Э



## Я