Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Кафедра «Компьютерные технологии»

Н. И. Поликарпова

Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением

Бакалаврская работа

Научный руководитель: к.т.н Шопырин Д. Г.

Санкт-Петербург
2006

Введение

При разработке сложных программных систем очень трудно создать код непосредственно на основе интуитивных представлений. Поэтому собственно программированию должен предшествовать этап *моделирования* системы, цель которого — представить проблемную область в виде множества взаимодействующих между собой абстракций. Для записи этих абстракций и обмена информацией о них используются *спецификации*. «Спецификация — это способ сказать, что есть абстракция, вне зависимости от каких бы то ни было ее реализаций» [1].

Два основных инструмента для создания модели сложной системы — это декомпозиция и абстракция. Определенные способы декомпозиции и абстракции лежат в основе различных парадигм создания программного обеспечения. Например, процедурно-ориентированный подход базируется на функциональной декомпозиции — представлении системы в виде набора *функций*. В основе объектно-ориентированного подхода лежит объектная декомпозиция: архитектура системы в целом формируется из совокупности моделей отдельных сущностей, характерных для предметной области. Модели в этом случае называют *абстрактными типами данных* (АТД) или *классами*.

В процессе создания программного обеспечения часто возникает необходимость реализации сущностей со *сложным поведением*. Таким поведением обладают многие устройства управления, сетевые протоколы, сущности, отвечающие за взаимодействие других сущностей между собой и с пользователем и т.д.

Можно сказать, что объект обладает сложным поведением, если в ответ на возникновение некоторого *события* он может совершить одно из нескольких *действий*. Выбор действия зависит от информации, хранимой внутри и вне данного объекта на текущем шаге вычисления, а также на всех предыдущих шагах.

При традиционной программной реализации таких сущностей возникают избыточные переменные, называемые *флагами*, которым не соответствуют никакие элементы предметной области. Единственное предназначение флагов — участвовать в многочисленных, запутанных конструкциях ветвления, реали-

зующих логику поведения. Это решение трудно для понимания, подвержено ошибкам и практически не расширяемо.

Вместо этого в последнее время предлагается описывать объект со сложным поведением, приписывая ему некоторое множество *управляющих состояний*, в каждом из которых поведение объекта является «простым» и может быть описано непосредственно. Связь управляющих состояний с действиями и механизм переходов между состояниями удобно описывать с помощью *конечных автоматов* с выходами [2]. При этом вся логика поведения объекта оказывается сосредоточенной в автомате. Такой подход к описанию сложного поведения называют *автоматным*.

Те или иные модификации диаграмм переходов конечных автоматов применяются для описания сложного поведения в ряде известных методологий разработки компьютерных систем, таких как UML [3], Statemate [4], SWITCH-технология [5]. Например, в подходе Statemate после выполнения функциональной декомпозиции системы предлагается описать ее поведение в целом при помощи диаграммы Statechart, которая является расширением диаграммы переходов конечных автоматов. Язык моделирования UML, в свою очередь, основан на объектно-ориентированных концепциях, в соответствие с которыми как структура, так и поведение каждой сущности специфицируется изолированно от других. Однако авторы UML предлагают использовать диаграммы Statechart для описания поведения не каждой сущности со сложным поведением в отдельности, а всей системы или произвольной ее части.

С точки зрения автора, «настоящим» объектно-ориентированным подходом к описанию сложного поведения было бы выделение из совокупности АТД, описывающей систему, АТД, моделирующих сущности со сложным поведением, и применение автоматного подхода для описания поведения каждого из этих типов в отдельности. Абстрактный тип данных, который включает в себя конечный автомат, описывающий логику его поведения, будем называть *автоматизированным абстрактным типом данных* (ААТД).

Термин *автоматизированный* заимствован из области систем автоматического управления. Объект управления, интегрированный вместе с управляющей системой (управляющим автоматом) в одно устройство, называют *автоматизированным объектом управления*. ААТД является, в некотором смысле, программным аналогом такого устройства.

При спецификации компьютерной системы с использованием автоматизированных типов данных, сложное поведение инкапсулируется — пользователь может обращаться с ААТД так же, как с обыкновенным типом данных. Это позволяет применять для построения системы, как целого, привычные объектно-ориентированные приемы, например, композицию объектов и построение иерархий типов. Предлагаемый подход способствует большей модульности архитектуры системы и повторному использованию кода (а также спецификаций) благодаря возможности *наследования* и *подтипизации* (*subtyping*) ААТД.

Поскольку объектно-ориентированный и автоматный подходы к разработке программных систем в отдельности обладают рядом преимуществ, решение вопроса об их эффективном сочетании представляется важным, как с теоретической, так и с практической точек зрения. Например, в работах [6, 7] описывается один из вариантов сочетания этих парадигм, названный *объектно-ориентированным программированием с явным выделением состояний*. В статьях [8, 9, 10, 11] приводится множество паттернов проектирования, предназначенных для объектно-ориентированной реализации «машин с конечным числом состояний».

Однако в этих работах ставится следующий вопрос: как с помощью объектов реализовывать конечные автоматы? Таким образом, здесь моделирование системы выполняется в чисто автоматном стиле и лишь на более поздней стадии применяется объектно-ориентированный подход. Именно поэтому появляется огромное количество паттернов, которые трудно классифицировать и сравнивать, а значит и выбрать подходящий для конкретной задачи.

Проблемы, рассматриваемые в настоящей работе, принципиально иные: Как использовать автомат для описания поведения объекта? Как связаны логика поведения и его семантика? Является ли полученный объект со сложным поведением объектом в обычном смысле? Как интегрировать его в систему? В настоящей работе автор делает попытку применить оба подхода: автоматный и объектно-ориентированный уже на стадии моделирования.

Ряд авторов исследует вопрос о наследовании автоматов [12, 13], а также применении к ним других объектно-ориентированных приемов, способствующих повышению модульности и повторному использованию кода. Автор считает целесообразным говорить о наследовании и подтипизации не автоматов, а именно автоматизированных типов. Конечный автомат является лишь одной

из компонент такого типа, поэтому рассмотрение автомата изолированно нарушает инкапсуляцию.

Понятие сущности со сложным поведением как единого целого встречалось в литературе по объектно-ориентированному проектированию и раньше, например, в книге [15]. В разделе, посвященном паттерну проектирования `State`, авторы описывают объект, поведение которого зависит от состояния, однако это описание не развивается дальше постановки задачи.

Цель настоящей работы — построение математической модели сущности, поведение которой зависит от состояния, разработка метода ее формальной спецификации и перенесение на нее свойств и отношений, характерных для обыкновенных АТД.

Глава 1.

Объектно-ориентированное моделирование

Для того, чтобы говорить об *объектно-ориентированном подходе* к моделированию, необходимо зафиксировать некоторую *объектную модель* — определить понятие объекта и его связь с другими программными сущностями. Объектная модель является одной из составляющих более широкого понятия *вычислительной* модели, которое также включает в себя описание структуры процесса вычисления. Исследователями в области теоретических основ объектно-ориентированного программирования построен ряд формальных объектных моделей, свойства которых достаточно хорошо изучены, однако использование их на практике затруднительно, так как для теоретических построений они сильно упрощены. В то же время, распространенным объектно-ориентированным языкам, таким как C++, Simula, Smalltalk или Java соответствуют более сложные и более практически применимые модели, однако они «спрятаны» в самом языке и в явном виде не задаются. Поэтому формальное доказательство многих интересных свойств этих моделей невозможно.

В настоящей работе автор, за исключением нескольких деталей, придерживается модели, описанной Б. Лисков и Дж. М. Уинг в работах [16, 17]. Эта модель близка к потребностям практики, однако не привязана к какому-либо конкретному языку программирования. При ее использовании определения и утверждения предлагается формулировать в терминах спецификаций. Этот подход назван авторами «доказательно-теоретическим», в отличие от «модельно-теоретического», предполагающего формулировку на языке математической модели. Поскольку языки спецификаций лучше приспособлены для выражения свойств объектов, доказательно-теоретический подход является более практичным: он позволяет ввести более сложную объектную модель, сделав ее явное математическое описание менее детализированным.

1.1. Разновидности объектных моделей

В работах [18, 19, 20] подробно обсуждается ряд теоретических объектных моделей, а также их сравнение с практическими моделями, реализованными в объектно-ориентированных языках программирования. Основные признаки, по которым эти модели различаются между собой, можно сформулировать следующим образом:

- Большинство теоретических моделей соответствуют концепциям **функционального** программирования. В них объекту приписывается определенное поведение, но он не может хранить данные (сохранять свое *вычислительное состояние*). В практическом объектно-ориентированном программировании, напротив, более распространены языки с **императивной** семантикой.

- Почти все современные языки программирования типизированы (таким образом, объектная модель языка включает в себя некоторую *систему типов*). Однако понятия о типе объекта в разных языках различаются. Теоретические объектные модели используют более простое понятие — **интерфейсный** тип (в этом случае тип объекта однозначно определяется сигнатурами его методов). В свою очередь, в практических языках более распространены **реализационные** типы, определяемые не только сигнатурами, но и реализацией.

- Для объектно-ориентированного моделирования важно различие между **мономорфными** и **полиморфными** системами типов. В мономорфной системе каждому выражению может быть приписан ровно один тип, а в полиморфной — некоторое множество типов, связанных отношением подтипизации. Подтипизация является важным инструментом объектно-ориентированного моделирования и программирования, поэтому полиморфным системам отдается предпочтение как в теории, так и на практике.

- Большинство практических языков, на самом деле, являются не столько объектно-, сколько **классо-ориентированными**. Основной абстракцией (или единицей моделирования) в них является класс. При этом для реализационной системы типов, понятия *класса* объекта и его *типа* совпадают.

Все объекты в программе порождаются из классов путем инстанцирования. Многие теоретические модели, напротив, являются чисто **объектно-ориентированными** и не вводят понятия класса. В них объекты формируются из других объектов или значений базовых типов путем применения определенных в данной модели операций. В других теоретических моделях классы существуют, однако в непривычном для практического программирования виде: класс представляет собой функцию, конструирующую объект по некоторому набору значений. При этом понятия типа и класса не связаны между собой.

Объектная модель, принятая в настоящей работе, обладает императивной семантикой и полиморфной системой типов, не являющейся в чистом виде ни интерфейсной, ни реализационной. Основной единицей моделирования является абстрактный тип данных, полностью определяемый формальной спецификацией. Спецификация АТД может включать в себя как синтаксические (сигнатуры методов), так и семантические свойства (пред- и постусловия методов, инвариант типа и т.д.). Таким образом, в зависимости от детальности спецификации, понятие АТД может сводиться как к интерфейсному, так и к реализационному типу. В общем случае, АТД допускает некоторое множество *реализаций*. Реализации, в свою очередь, также описывают абстрактные типы данных, специфицированные достаточно подробно для того, чтобы их можно было инстанцировать в объекты. Такие АТД являются *классами* в том смысле, как этот термин понимается в практических объектно-ориентированных языках.

1.2. Математическое описание вычислительной модели

Пусть имеется множество всех возможных *значений* Val, любое подмножество которого называется *сортом*. Кроме того, имеется множество всех потенциально существующих *объектов* Obj. *Вычислительным состоянием* программы $CState$ назовем пару отображений — *окружение* Env, сопоставляющее объекты программным переменным, и *хранение* $Store$, определяющее для каж-

дого объекта его текущее значение.

$$CState = Env \times Store$$
$$Env \colon Var \to Obj$$
$$Store \colon Obj \to Val$$

Если обозначить множество всех типов как $Type$, тогда *типизация* — это отображение $T \colon Obj \to Type$, сопоставляющее каждому объекту его тип, а *статическая типизация* — отображение $ST \colon Var \to Type$, сопоставляющее тип каждой переменной. Тип определяет множество допустимых значений объекта (иначе говоря, возможные значения $Store(o)$ для данного объекта o в любом вычислительном состоянии программы), а также набор методов, предоставляющих единственный способ манипулирования объектом. Утверждение $T(o) = \tau$ часто записывают в виде $o : \tau$.

Определение 1. *Абстрактным типом данных* будем называть пару

$$\tau = \langle V, M \rangle,$$

где $V \subset Val$ — множество *допустимых значений* (или *допустимый сорт*) абстрактного типа, $M = \{m = \langle \alpha, \rho, f \rangle\}$ — конечное множество его *методов*. Каждый метод включает в себя набор типов аргументов $\alpha \in Type^*$, тип результата $\rho \in Type$ и, наконец, *исполняющую функцию* — частичное многозначное отображение $f \colon V \times \alpha.V \multimap V \times \alpha.V \times \rho.V$.

Совокупность соответствующих методу типов α и ρ определяет его сигнатуру, а исполняющая функция — его семантику. Как следует из определения, исполняющая функция не всегда однозначно определяет новое значение объекта, аргументов и результата. Именно это свойство позволяет АТД иметь несколько реализаций. Метод, исполняющая функция которого определена однозначно, называется *однозначно специфицированным*. Если все методы некоторого АТД однозначно специфицированы, он является классом.

Метод, который не изменяет значения вызвавшего его объекта, называется *наблюдателем*, а модифицирующий это значение — *мутатором*. Если исполняющая функция метода имеет вид $f \colon \alpha.V \multimap \alpha.V \times \rho.V$, то такой метод называется *статическим*. Важная разновидность статических методов — конструкторы

типа, которые служат для инстанцирования объектов. *Конструктором* типа τ называется метод вида $m = \langle \alpha, \tau, f\colon \alpha.V \multimap \alpha.V \times \tau.V \rangle$. Отметим, что для того, чтобы действительно можно было инстанцировать объекты, конструктор должен быть однозначно специфицирован.

Вычислением (или исполнением программы) будем называть последовательность чередующихся вычислительных состояний программы и *инструкций*:

$$c_0 S_1 c_1 \ldots S_n c_n$$

Каждая инструкция S_i — это частичная функция на множестве вычислительных состояний программы. Говоря об области определения вычислительного состояния $dom(c)$, будем иметь в виду область определения хранения в этом состоянии — множество объектов, существующих в программе в этом состоянии.

Пусть c — вычислительное состояние программы с окружением $c.e$ и хранением $c.s$, а o — некоторый объект, существующий в c. Через o_c будем обозначать значение o в состоянии c ($o_c = c.s(o)$). В соответствии с нотацией распространенных объектно-ориентированных языков будем обозначать вызов метода m у объекта o с последовательностью аргументов a как $o.m(a)$. Для вызова статического метода типа τ, не требующего конкретного объекта, будем использовать обозначение $\tau :: m(a)$. Говоря о полиморфной системе типов, будем полагать, что для переменной y и любого возможного вычислительного состояния c не обязательно выполняется $T(c.e(y)) = ST(y)$, но гарантированно $T(c.e(y)) \leqslant ST(y)$. Здесь $\leqslant$ — частичный порядок на множестве $Type$, называемый *отношением подтипизации* (разд. 4.2).

1.3. Спецификация АТД

Для спецификации абстрактных типов данных в настоящей работе по ряду причин был выбран язык Larch [1].

- Этот язык напрямую поддерживает объектно-ориентированную парадигму. Спецификация абстрактных типов данных — одно из основных назначений этого языка. Его средства описания АТД согласуются с выбранной в настоящей работе объектной моделью.

- Этот язык является формальным. Свойство формальности необходимо при доказательно-теоретическом подходе к формулировке и доказательству утверждений об абстрактных типах. Кроме того, формальные спецификации являются одновременно краткими и выразительными.

- Larch позволяет описывать модели, предназначенные для дальнейшей реализации на определенном целевом языке программирования. Это облегчает переход от модели к коду, не снижая уровня абстракции.

- Для языка Larch разработан набор инструментальных средств, включая систему автоматического доказательства свойств спецификаций Larch Prover.

Larch состоит из двух уровней, первый из которых называется Larch Shared Language (LSL) и используется, в частности, для описания сортов. Единицей спецификации на LSL является *свойство* (*trait*), содержащее набор аксиом (и, возможно, теорем), описывающих некоторый математический объект. Второй уровень Larch (Larch Interface Language) — это семейство языков, схожих по выразительной мощности, но синтаксически различных. Они предназначены непосредственно для спецификации функций и абстрактных типов данных. Конкретный синтаксис выбирается из соображений подобия с целевым языком программирования. В настоящей работе будем использовать упрощенный вариант Larch/C++ [21].

Спецификация абстрактного типа данных включает следующую информацию:

- имя типа;

- описание множества допустимых значений типа;

- для каждого из методов типа:

  - имя метода;

  - сигнатуру метода;

  - семантику метода в терминах *предусловий* и *постусловий*.

Пред- и постусловия представляют собой удобный способ задания исполняющей функции метода. Предусловие определяет множество «входных данных» (значений объекта и аргументов метода), на котором определена исполняющая функция, а постусловие описывает ее результат при допустимых входных данных.

Допустимый сорт можно задавать непосредственно с помощью свойств, описанных на LSL. Однако опыт показал, что сорта большинства типов представляют собой подмножества некоторых «фундаментальных» математических объектов (множества целых чисел, множества кортежей и т.п.). Поэтому удобнее иметь небольшой стандартный набор свойств на LSL, описывающих такие фундаментальные объекты, и выделять из них в точности допустимые сорта, явно указывая в спецификации АТД дополнительную аксиому, называемую *инвариантом* типа I_τ.

Альтернативным подходом можно считать указание в спецификации АТД всего пространства значений, а не в точности допустимого сорта. В этом случае допустимость конкретного значения может быть доказана при помощи *индукции типов* [17]. Однако для доказательства базы индукции необходимо располагать спецификациями всех конструкторов типа. В то же время, включение конструкторов в спецификацию АТД, не являющегося классом, значительно ограничивает пространство его возможных реализаций и подтипов. Тогда как остальные методы имеют дело с *видимым* состоянием объекта, не затрагивая его конкретного представления, конструкторы непосредственно работают с этим представлением, «запаковывая» его в объект. Поэтому конструкторы концептуально расположены на более низком уровне абстракции, чем остальные методы.

В отсутствие индукции типов теряется возможность доказательства не только утверждений о допустимости отдельных значений объекта, но и *исторических свойств* — утверждений о допустимости определенных последовательностей его значений. При необходимости к спецификации АТД можно добавить явное описание исторических свойств при помощи конструкции, называемой *ограничением* типа C_τ.

Пример 1. В качестве примера приведем спецификацию абстрактного типа данных `Bag`, заимствованную из работы [16].

```
class Bag {
    spec IntMultiset elems;
    spec int bound;

    invariant: |this.elems| <= this.bound;

    constraint: this^.bound == this'.bound;
public:
    void Put(int i) {
        requires: |this^.elems| < this^.bound;
        modifies: this;
        ensures: this'.elems == this^.elems \/ {i} &&
                 this'.bound == this^.bound;
    }

    int Get() {
        requires: this'.elems != {};
        modifies: this;
        ensures: this'.elems == this^.elems - {result} &&
                 result in this^.elems &&
                 this'.bound == this^.bound;
    }

    int Card() {
        ensures: result == |this^.elems|;
    }

    bool Equal(Bag b) {
        ensures: result == (this == b);
    }
}
```

В приведенном примере сорт абстрактного типа задан двумя *переменными спецификации* [21]. Это множество целых чисел с повторениями `elems` и целое число `bound`, ограничивающее размер множества. Ключевое слово `invariant`

объявляет инвариант типа, а после слова `constraint` записано его ограничение. Такая запись является сокращением утверждения, что для любого вычисления и для любого объекта $o : \texttt{Bag}$ выполняется:

$$\forall c, d \in CState\,(c < d \wedge o \in dom(c)) \Rightarrow (o_c.\texttt{bound} = o_d.\texttt{bound}).$$

В описании семантики методов ключевое слово `requires` объявляет предусловие. Постусловие представляет собой конъюнкцию выражения `ensures` и утверждения, что значения никаких объектов, кроме указанных в выражении `modifies`, не изменяются. Через `this^` обозначается значение объекта перед началом исполнения метода, а через `this'` — в момент его завершения.

Глава 2.

Простейшие модели сущностей со сложным поведением

К понятию автоматизированного абстрактного типа данных, положенного в настоящей работе в основу сочетания объектной и автоматной парадигм, можно придти путем обобщения ряда детерминированных *абстрактных вычислителей* (*абстрактных машин*), в том числе, конечного автомата, автомата с магазинной памятью и машины Тьюринга [2]. В теории формальных языков абстрактные машины используются в качестве эталонов вычислительной мощности, однако в то же время они являются простейшими моделями сущностей со сложным поведением. Свойства абстрактных вычислителей хорошо изучены — это будет использовано ниже, например, для классификации автоматизированных абстрактных типов данных.

Рассмотрим подробнее указанные выше абстрактные вычислители (отметим, что все они являются детерминированными).

Определение 2. *Конечным автоматом* называется пятерка

$$A = \langle S, E, s_0, F, \delta \rangle,$$

где S — конечное множество управляющих состояний, E — конечное множество входных символов, $s_0 \in S$ — начальное (стартовое) состояние, $F \subset S$ — множество допускающих состояний, $\delta\colon S \times E \to S$ — функция переходов.

Конечный автомат (рис. 1) называют также *машиной с конечным числом состояний* (*finite state machine*, *FSM*).

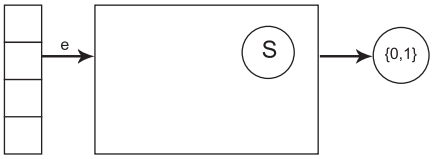

Рис. 1. Конечный автомат

Определение 3. *Автоматом с магазинной памятью* называется семерка

$$P = \langle S, E, X, s_0, F, x_0, \delta \rangle,$$

где S — конечное множество управляющих состояний, E — конечное множество входных символов, X — конечное множество магазинных символов, $s_0 \in S$ — начальное (стартовое) состояние, $F \subset S$ — множество допускающих состояний, $x_0 \in X$ — начальный магазинный символ («маркер дна»), $\delta\colon S \times E \times X \to S \times X^*$ — функция переходов.

Автомат с магазинной памятью — *стековая машина* (*push-down stack machine*, *PDSM*) схематично изображен на рис. 2. Рассматривают также стековые машины без множества F, которые осуществляют допуск по пустому магазину.

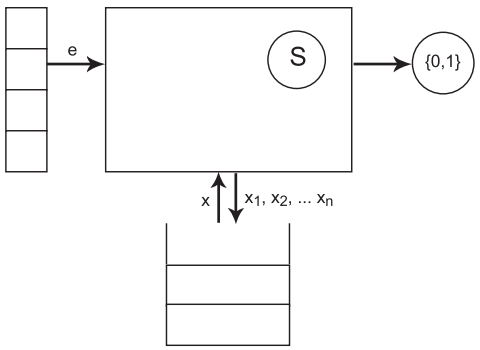

Рис. 2. Автомат с магазинной памятью

Определение 4. *Машиной Тьюринга* называется шестерка

$$M = \langle S, X, s_0, T, x_0, \delta \rangle,$$

где S — конечное множество управляющих состояний, X — конечное множество ленточных символов, $s_0 \in S$ — начальное состояние, $T \subset S$ — множество завершающих (терминальных) состояний, $x_0 \in X$ — пробельный символ, $\delta\colon S \times X \to S \times X \times \{\leftarrow, \rightarrow\}$ — функция переходов.

Машину Тьюринга (рис. 3) можно было бы назвать «автоматом с ленточной памятью». У нее, как и у стековой машины, может быть несколько вариантов допуска. В простейшем случае множество ее терминальных состояний делится на два подмножества $T = Y \cup N$, первое из которых содержит

допускающие состояния, а второе — *отвергающие*. Без ограничения общности можно считать, что каждое из этих подмножеств содержит по одному элементу, поскольку, достигая любого завершающего состояния, машина Тьюринга немедленно останавливается.

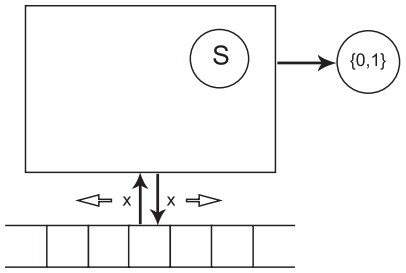

Рис. 3. Машина Тьюринга

Устройство перечисленных абстрактных вычислителей во многом похоже. Подобным же образом организованы счетчиковые машины [2] (их можно условно назвать «автоматами с целочисленной памятью»), а также различные модификации уже рассмотренных машин, такие как автомат с несколькими магазинами, многоленточная машина Тьюринга и т.п. Для них характерны следующие общие черты:

- все абстрактные вычислители имеют хранилище информации определенного вида (лента, магазин или счетчик);
- у каждого из них есть «конечное управление» (определяющее множество управляющих состояний и условия переходов между ними);
- процесс работы абстрактных вычислителей состоит из последовательных шагов, на каждом из которых с помощью функции переходов вычисляется новая «конфигурация» хранилища информации и управления.

Под *конечным* управлением, подразумевается не только конечность множества управляющих состояний всех вычислителей, но и специфика функции переходов, которая всегда оперирует только конечными множествами. Это свойство позволяет непосредственно задать значения этой функции для всех возможных аргументов (при помощи таблицы или диаграммы переходов).

В то время как управление всех абстрактных вычислителей устроено одинаково, их хранилища информации различаются по структуре и назначению.

С ленты конечного автомата можно только считывать символы, но не записывать — такой тип хранилища будем называть *входом*. Обобщая термины «автомат с магазинной/ленточной/целочисленной памятью», хранилище информации, доступное как для чтения, так и для записи, назовем *памятью* машины. В этих терминах существенные различия между абстрактными вычислителями определяются наличием входа, а также структурой и функциональностью памяти.

Руководствуясь данными выше неформальными определениями, можно решить, что вход — это всего-навсего частный случай памяти, однако в действительности наличие входа оказывает существенное влияние на процесс работы машины. Дело в том, что детерминированный абстрактный вычислитель гарантированно считывает по одному символу со входа на каждом шаге и при достижении конца последовательности входных символов прекращает работу. Таким образом, он не нуждается в специальном механизме останова. Более того, можно точно определить число шагов, требуемое для обработки заданной входной строки таким вычислителем, не имея представления о его внутреннем устройстве. Такие абстрактные машины будем называть *открытыми*. *Закрытые* вычислители без входа (такие как машина Тьюринга) являются менее предсказуемыми, так как для них не только нельзя предсказать, сколько потребуется шагов для обработки слова, заданного как начальное состояние памяти, но и невозможно определить, завершится ли эта обработка вообще. Закрытым вычислителям требуется специальный механизм останова (в случае машины Тьюринга он реализован как множество терминальных состояний).

Кроме текущего входного символа («конфигурации входа») и управляющего состояния («конфигурации управления») ход вычисления на каждом шаге зависит также от «конфигурации памяти», которая называется *вычислительным состоянием* машины. В том случае, когда множество вычислительных состояний бесконечно, функция переходов не может непосредственно считывать конфигурацию памяти и присваивать ей новое значение. Вместо этого управление оперирует конечными множествами *наблюдаемых вычислительных состояний* (или коротко *наблюдений*) и *действий*. С помощью действий можно изменить вычислительное состояние, а с помощью наблюдений — получить о нем определенную информацию. Например, стековой машине можно приписать $|X| + 1$ действий (по одному для добавления в стек каждого из символов

магазинного алфавита и еще одно для извлечения символа из стека) и $|X|$ наблюдений, соответствующих каждому возможному символу на вершине стека). Конечное управление само по себе не определяет связь действий и наблюдений с вычислительным состоянием. Образно говоря, исполнением операций с памятью занимается специальное «исполняющее устройство». Совокупность памяти и «исполняющего устройства» будем называть *вычислительной компонентой* машины.

На практике нередко встречаются сущности со сложным поведением, для описания логики которых достаточно абстрактной машины без наблюдений (процесс работы ее конечного управления не зависит от вычислительного состояния). Такую машину будем называть *разомкнутой*, а в противном случае (если она использует наблюдения) — *замкнутой*. Эти термины заимствованы из теории автоматического управления. Действительно, если изобразить абстрактный вычислитель так, как принято изображать управляющие системы (рис. 4), то станет очевидной аналогия полноценной машины с замкнутой системой (системой с обратной связью) и упрощенной машины без наблюдений — с разомкнутой системой.

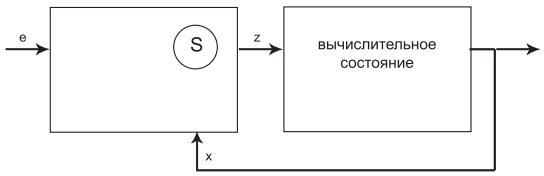

Рис. 4. Абстрактная машина как система управления

Приведенные выше формальные определения содержат информацию лишь о структуре абстрактных машин, однако полезно иметь также представление о процессе их работы. Этот процесс для любой абстрактной машины состоит из *шагов*. Шаг можно определить как последовательность исполняемых машиной инструкций, которая завершается вычислением функции переходов. Отметим, что в этой последовательности функции переходов вычисляется только один раз.

Как должна действовать абстрактная машина после завершения шага: возвратить управление вызывающему контексту или же немедленно перейти к следующему шагу? В первом случае каждый шаг работы машины инициируется пользовательской функцией. По отношению к этой функции абстрактная машина является *пассивной*. В противном случае она запускает сама себя — рабо-

тает в цикле до тех пор, пока не закончится входная последовательность, либо не будет достигнуто терминальное состояние (в зависимости от того, является ли машина открытой или закрытой). Такие абстрактные вычислители будем называть *активными*.

Каждый из введенных видов абстрактных машин имеет собственную область применения в сфере программного обеспечения. Например, в терминологии Д. Харела [4], *трансформирующую* компьютерную систему можно описать активной машиной, а *реактивную* систему — пассивной. Далее, из класса трансформирующих систем можно выделить те, которые посимвольно обрабатывают некоторую входную строку (например, компиляторы) — их моделью служат открытые активные машины, тогда как остальные системы удобнее описывать закрытыми вычислителями. В качестве модели *сущности со сложным поведением* будем использовать открытую пассивную абстрактную машину, на ее основе и строится понятие автоматизированного абстрактного типа данных. Выбор между разомкнутой и замкнутой ее разновидностями осуществляется на основе некоторых соображений о сложности логики поведения моделируемой сущности (разд. 3.1).

Глава 3.

Автоматизированный абстрактный тип данных

3.1. Понятие ААТД

При моделировании сущностей со сложным поведением обыкновенными абстрактными типами данных возникает ситуация, когда одному имени метода (событию) соответствует несколько простых действий, но выбор конкретного действия зависит, в общем случае, и от аргументов метода и от значений, хранящихся в объекте на протяжении всего вычисления. Для сохранения информации об «истории» значений объекта приходится неоправданно расширять допустимый сорт его типа. При этом спецификации постусловий методов этого типа перенасыщены условными конструкциями и становятся сложными для понимания.

При применении автоматного подхода история значений оказывается закодированной в управляющем состоянии. Исполняющая функция каждого метода представляется в виде набора простых функций-действий. Автомат, управляющий объектом, осуществляет диспетчеризацию вызовов функций-действий не только по событиям (как у обыкновенных объектов), но и по управляющим состояниям. С другой стороны, функции-действия можно считать исполняющими функциями методов некоторого нового абстрактного типа данных, который по отношению к первичному типу, описывающему сущность со сложным поведением, будем называть *вложенным*. Таким образом, модель сущности со сложным поведением представляется в виде совокупности *управляющей* и *вычислительной* компонент, которые вместе составляют *автоматизированный абстрактный тип данных*.

На рис. 5 абстрактный тип данных изображен в виде функции, множество входных данных которой разделено на две компоненты: конечное множество событий (имен методов) и потенциально бесконечное множество значений аргументов методов. На рис. 6 изображен *разомкнутый* автоматизированный тип данных — это простейшая разновидность ААТД, управляющий автомат кото-

рого не использует наблюдения. Как следует из рисунка, конечное управление ААТД преобразует только события, но не влияет на аргументы методов.

Рис. 5. Абстрактный тип данных

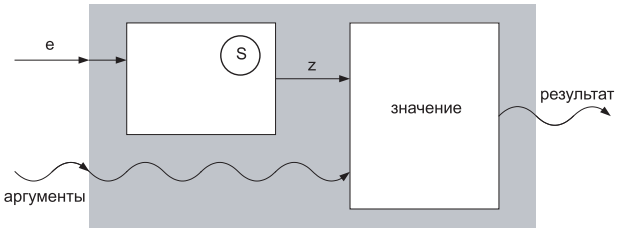

Рис. 6. Разомкнутый ААТД

Определение 5. *Разомкнутым автоматизированным абстрактным типом данных* будем называть пару $\langle A, \tau \rangle$, где A — управляющий автомат, τ — вложенный абстрактный тип данных.

Управляющий автомат представляет собой пятерку

$$A = \langle S, E, Z, s_0, \Delta \rangle,$$

где S — конечное множество управляющих состояний, E — конечное множество входных символов (или событий), Z — конечное множество действий, $s_0 \in S$ — начальное (стартовое) состояние, $\Delta \colon S \times E \to S \times Z$ — управляющая функция.

Управляющую функцию можно разложить на две компоненты ζ и δ, где $\zeta \colon S \times E \to Z$ — функция действий, $\delta \colon S \times E \to S$ — функция переходов.

Пусть $\tau = \langle V, M \rangle$, тогда $Z = M$.

Существует широкий класс сущностей со сложным поведением, для которого модель разомкнутого ААТД оказывается непригодной. Во-первых, вычислительной мощности конечного автомата может быть недостаточно для описания выделенной разработчиком управляющей компоненты автоматизированного типа. Во-вторых, даже в том случае, когда логика поведения может быть описана конечным автоматом, в нем могут присутствовать группы «однотипных» состояний. Такие состояния не несут в себе уникальной информации, их

можно различить, зная вычислительное состояние объекта. Наличие групп однотипных состояний свидетельствует о неправильном распределении обязанностей между управляющей и вычислительной компонентами. Подчеркнем, что поскольку выделение управляющей компоненты — это нетривиальное проектное решение, принимаемое отдельно для каждой моделируемой сущности со сложным поведением, то и критерии выбора между разомкнутой и замкнутой моделями практически невозможно формализовать.

Примером нецелесообразного разделения на вычислительную и управляющую компоненту, по мнению автора, может служить реализация эмулятора «торгового автомата», приведенная в статье [22]. Описанное в ней устройство принимает монеты различного достоинства (5, 10 и 25 центов) и выдает напиток, когда накопленная им сумма достигает или превышает 25 центов. При моделировании «торгового автомата» было принято решение сопоставить отдельное управляющее состояние каждой возможной накопленной сумме. При таком подходе появляется большое количество управляющих состояний, модель трудно расширять при необходимости изменения множества принимаемых монет и стоимости напитка. Интуитивно понятно, что характер поведения торговой машины зависит не от фактического значения накопленной суммы, а только от того, достаточна ли она для покупки напитка. Поэтому, по мнению автора, более целесообразным было бы суммировать достоинства монет средствами вложенного типа и позволить управляющему автомату опрашивать достаточность суммы (использовать *замкнутую* разновидность ААТД). В терминах разд. 2 все множество значений вложенного типа разбивается тем самым на два наблюдаемых вычислительных состояния (наблюдения): «накопленная сумма меньше 25 центов» и «сумма больше либо равна 25 центов».

Каким образом управляющий автомат опрашивает текущее наблюдение? Удобно считать, что для каждого состояния s и события e управляющему автомату известен набор *предикатов* $\{p_i\}$ — методов-наблюдателей вложенного типа, не имеющих побочного эффекта и возвращающих логические значения (отображение, сопоставляющее каждой паре из состояния и события набор предикатов, будем называть *наблюдающей функцией*). После вызова предикатов вектор возвращенных ими значений, который в данном случае и представляет собой наблюдение, используется в качестве аргумента управляющей функции.

Определение 6. *Замкнутым автоматизированным абстрактным типом данных* называется пара $\langle A, \tau \rangle$, где A — управляющий автомат, τ — вложенный абстрактный тип данных.

Управляющий автомат представляет собой семерку

$$A = \langle S, E, Z, P, s_0, \pi, \Delta \rangle,$$

где S — конечное множество управляющих состояний, E — конечное множество входных символов (или событий), Z — конечное множество действий, P — конечное множество предикатов, $s_0 \in S$ — начальное (стартовое) состояние, $\pi\colon S \times E \to 2^P$ — наблюдающая функция, $\Delta\colon S \times E \times \{0,1\}^{|\pi(s,e)|} \to S \times Z$ — управляющая функция.

Пусть $\tau = \langle V, M \rangle$, тогда $M = Z \cup P$, причем $\forall p \in P$

$$p = \langle \alpha, bool, f\colon V \times \alpha.V \to \{0,1\} \rangle.$$

Замкнутый ААТД схематично изображен на рис. 7.

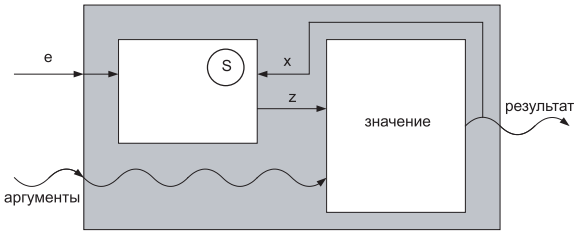

Рис. 7. Замкнутый ААТД

Отметим, что в приведенном определении методы-предикаты предполагаются однозначно специфицированными. Такое решение связано со стремлением сделать процесс работы управляющего автомата как можно более предсказуемым. Замкнутый ААТД (в отличие от разомкнутого) нельзя назвать детерминированным вычислителем. Последовательность состояний, в которых будет находиться его управляющий автомат в процессе вычисления, невозможно предсказать, поскольку новое вычислительное состояние на каждом шаге определено неоднозначно. Однако, в предположении, что текущее значение объекта известно, новое управляющее состояние и действие определены единственным образом.

Несмотря на то, что на практике чаще всего используется описанный выше вариант с предикатами (например, в [5], где они называются *входными переменными*), наблюдения, вообще говоря, необязательно должны иметь вид

битовых векторов. В общем случае можно определить для каждого управляющего состояния s и события e разбиение множества значений на наблюдения $X(s,e) = \{x_i\}_{i=1}^{k(s,e)}$. Тогда управляющая функция автомата будет иметь вид $\Delta\colon S \times E \times X(s,e) \to S \times Z$, а наблюдающая функция будет возвращать методы-наблюдатели (или наборы таких методов), тип результата которых имеет произвольное конечное множество значений.

Чем объясняется выбор именно такой математической модели для сущности со сложным поведением? С одной стороны, она является обобщением простых абстрактных вычислителей (разд. 2), ее можно назвать «автоматом с произвольной памятью». Тезис Черча-Тьюринга гласит, что любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга. Несмотря на это, машина Тьюринга не годится на роль универсальной модели сущности со сложным поведением. Дело в том, что вычислительная компонента этой машины имеет крайне ограниченный набор операций с памятью, иначе говоря, она слишком проста для реального программирования [23]. Естественный способ построения подходящей модели заключается в том, чтобы расширить набор доступных операций с памятью, но при этом сохранить конечное управление, отвечающее за логику поведения объекта.

С другой стороны, построенная модель призвана заполнить «теоретический пробел» между сущностью со сложным поведением и ее удачными программными реализациями, выраженными в терминах паттернов проектирования. Например, паттерн `State` [15] предназначен для реализации объекта со сложным поведением, логика которого описывается *автоматом Мили*, как и в модели ААТД. Кроме того, входные символы (события) автомата также отождествляются с именами методов объекта. Паттерн Р. Мартина `Three Level FSM` [8] (также известен как `Layered Organization FSM Pattern`) еще точнее соответствует построенной модели: в нем действия реализованы в виде методов абстрактного класса `Actions`, который можно считать интерфейсом вложенного типа ААТД.

В методологиях разработки сложных реактивных систем, в частности, в Switch-технологии, каждое действие (называемое *выходным воздействием*) состоит из последовательности *выходных переменных*. Руководствуясь этим примером, в модели ААТД можно было бы сопоставить каждому действию последовательность методов вложенного типа. Однако, по мнению автора, такое ре-

шение наносит ущерб общности и простоте модели. Во-первых, событие ААТД может иметь аргументы, которые напрямую передаются методу-действию: в случае последовательности методов, отвечающих действию, передача им аргументов становится неоднозначной. Подобным же образом дело обстоит и с возвращаемым значением. Во-вторых, структура действия в виде последовательно вызываемых методов накладывает неоправданное ограничение (почему не использовать, например, их композицию). В общем случае, удобно было бы сопоставить действию некоторую программу, вызывающую методы вложенного типа. Однако выразить сигнатуру и семантику такой программы через сигнатуры и семантику вызываемых ею методов в общем случае невозможно. Такую программу всё равно необходимо специфицировать вручную. Поэтому для простоты модели ее целесообразнее заменить еще одним методом вложенного типа.

По мнению автора, именно отсутствие адекватной математической модели сущности со сложным поведением порождает множество разнообразных методологий и паттернов проектирования, которые трудно классифицировать и, тем более, выбрать из них наиболее подходящий для конкретной задачи.

3.2. Спецификация ААТД

В этом разделе предлагается метод формальной спецификации автоматизированных абстрактных типов данных. Отметим, что при этом новой нотации не вводится — для полного описания ААТД достаточно объединить некоторый уже существующий язык спецификации конечных автоматов Мили, с помощью которого будет описываться управляющая компонента, и некоторый язык спецификации абстрактных типов данных, с помощью которого будет описываться вложенный тип.

Для спецификации поведения автоматов Мили в настоящей работе используются *диаграммы переходов* (*state-transition diagrams*, STD) [2]. На диаграмме переходов автомат изображается в виде ориентированного графа, вершины которого отвечают состояниям, а дуги — переходам. Каждая дуга помечается *условием перехода* и *действием на переходе*. Условие перехода содержит одно или несколько событий, а в случае замкнутого ААТД — булеву формулу над событиями и предикатами (по умолчанию тождественно истинную).

Спецификации абстрактных типов данных выполняются на языке Larch (разд. 1.3). Язык спецификации, полученный в результате объединения диаграмм переходов и языка Larch, назовем Larch/STD.

Проиллюстрируем предлагаемый метод спецификации ААТД примером сущности со сложным поведением из работы [24]. В этой статье описана сущность, которая может быть взята на прокат — `LoanableItem`. Это может быть, например, книга в библиотеке или кассета в видеопрокате. На такой предмет можно воздействовать следующим образом: его можно взять (`CheckOut`), вернуть (`Return`) или зарезервировать (`Reserve`). Реакция на эти события зависит от текущего управляющего состояния (доступен ли предмет, зарезервирован ли он или уже взят).

Промоделируем рассматриваемый предмет замкнутым ААТД. Его управляющая компонента может быть описана с помощью диаграммы переходов, изображенной на рис. 8.

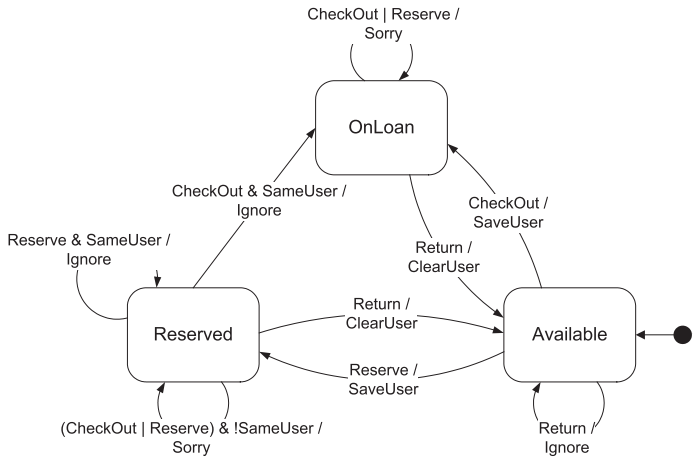

Рис. 8. Диаграмма переходов управляющего автомата `LoanableItem`

Из этой диаграммы следует, что управляющий автомат состоит из компонент:

$$S = \{\texttt{Available}, \texttt{OnLoan}, \texttt{Reserved}\},$$
$$E = \{\texttt{CheckOut}, \texttt{Return}, \texttt{Reserve}\},$$
$$Z = \{\texttt{SaveUser}, \texttt{CleanUser}, \texttt{Sorry}, \texttt{Ignore}\},$$
$$P = \texttt{SameUser},$$
$$s_0 = \texttt{Available},$$

Вложенный тип ААТД (`LoanableItemNested`) можно специфицировать на языке Larch следующим образом:

```
class LoanableItemNested {
    spec const User noUser;
    spec User currentUser;
    spec Date returnDate;
public:
    void SaveUser(User user, Date currentDate) {
        requires: user != noUser;
        modifies: this;
        ensures: currentUser' == user && returnDate > currentDate;
    }

    bool Expired(User user, Date currentDate) {
        requires: true;
        modifies: this;
        ensures: currentUser' == noUser &&
            result == (currentDate > returnDate);
    }

    void Sorry(User user, Date currentDate) {
        requires: true;
    }

    bool Ignore(User user, Date currentDate) {
        requires: true;
        ensures: result == false;
    }

    bool SameUser(User user, Date currentDate) {
        requires: true;
        ensures: result == (user == currentUser);
    }
}
```

Связь между диаграммой переходов управляющего автомата и спецификацией вложенного типа обеспечивается идентичностью действий и предикатов на диаграмме и соответствующих им имен методов вложенного типа. Если управляющий автомат имеет большое количество состояний и переходов, на диаграмме удобнее использовать сокращенные имена событий, действий и предикатов (например, e_i, z_i и x_i соответственно, как в нотации SWITCH-технологии). В этом случае для описания ААТД можно дополнительно сформировать *словарь сокращений*, отображающий связь между полными и сокращенными именами.

Отметим, что при рассматриваемом подходе к спецификации автоматизированных типов данных можно легко заменить классические диаграммы переходов, диаграммами Statechart или графами переходов из SWITCH-технологии. Однако, автор не видит в этом необходимости. Главное достоинство упомянутых нотаций состоит в декомпозиции логики. В SWITCH-технологии она достигается с помощью вложенных и вызываемых автоматов, в нотации Statechart — с помощью вложенных и ортогональных состояний. Поскольку предлагаемый подход является объектно-ориентированным, предлагается использовать вместо этих видов декомпозиции объектную декомпозицию. Таким образом, если поведение выделенного разработчиком автоматизированного типа оказалось слишком сложным, его необходимо разложить на несколько более простых автоматизированных типов.

В терминах SWITCH-технологии спецификация вложенного типа вместе со словарем сокращений заменяет и дополняет часть схемы связей автомата, отвечающую за связи с объектом управления. При этом вместо неформального описания предикатов и действий на схеме связей, приводится их формальная спецификация в терминах пред- и постусловий. В терминах подхода Statemate спецификация вложенного типа в некотором смысле заменяет Activity Chart.

Глава 4.

ААТД с точки зрения объектно-ориентированного моделирования

Как отмечалось в разд. 3.1, с точки зрения теории автоматов, ААТД является «обобщенной» абстрактной машиной.

Для разработки программных систем бóльшее значение имеет вопрос, какое место занимает понятие автоматизированного абстрактного типа среди примитивов объектно-ориентированного моделирования.

Что означает термин *объектно-ориентированная* модель сущности со сложным поведением? Поскольку в объектно-ориентированном подходе любая сущность моделируется абстрактным типом данных, то и модель сущности со сложным поведением должна быть в некотором смысле частным случаем абстрактного типа данных.

В рамках объектно-ориентированного подхода сложное поведение можно рассматривать как особую форму *полиморфизма — динамического* (поведение объекта изменяется в процессе выполнения программы) и *ограниченного* (вариантов поведения конечное число и каждый из них специфицируется вручную). По этим характеристикам «полиморфизм сложного поведения» похож на привычный для объектно-ориентированного программирования полиморфизм подтипов. Принципиальное различие между ними заключается в том, что поведение объектов подтипов должно быть семантически связано с поведением объектов супертипа (разд. 4.2), в то время как поведение объекта в различных управляющих состояниях может варьироваться произвольно. Однако сходство этих двух форм полиморфизма можно использовать для реализации одной из них посредством другой, как, например, в паттерне проектирования `State` [15].

4.1. Эквивалентный АТД

В данном разделе обсуждается связь ААТД с абстрактным типом данных. Утверждается, что по спецификации ААТД $\langle A, \tau \rangle$ на языке Larch/STD мож-

но построить спецификацию *эквивалентного* АТД τ' на языке Larch. Говоря об эквивалентности в терминах спецификаций обычно подразумевают, что любое свойство, которое может быть доказано из спецификации одной сущности, должно быть доказуемо и из спецификации эквивалентной и наоборот. Однако множество всех свойств, имеющих смысл для обеих сущностей, не определено однозначно. С одной стороны, у ААТД есть специфические «автоматные» свойства (например, «количество управляющих состояний равно n»). С другой стороны, при анализе абстрактных типов данных наибольший интерес представляют их поведенческие свойства, выражаемые в терминах пред- и постусловий методов, в то время как поведение ААТД не выражено в этих терминах в явном виде. Поэтому в качестве поведенческой эквивалентности автоматизированного и обыкновенного абстрактных типов можно предложить *неотличимость по возвращаемым значениям*:

Определение 7. Будем говорить, что автоматизированный абстрактный тип данных $\langle A, \tau \rangle$ и абстрактный тип данных τ' *неотличимы по возвращаемым значениям*, если для любых объектов $o : \langle A, \tau \rangle$ и $u : \tau'$, созданных конструкторами с одинаковым именем и аргументами, выполняется: множества возможных возвращаемых значений $R_1, R_2, \ldots, R_n$ в результате последовательности вызовов одних и тех же методов с одинаковыми аргументами у объектов o и u совпадают.

Приведем теперь алгоритмы построения эквивалентного абстрактного типа данных по разомкнутому и замкнутому ААТД (первый алгоритм проще и нагляднее, но, в действительности, он является частным случаем второго). После этого покажем, что построенный при помощи алгоритма эквивалентный АТД и исходный ААТД неотличимы по возвращаемым значениям.

Алгоритм 1. [Построение эквивалентного АТД для разомкнутого ААТД] Пусть имеется разомкнутый автоматизированный абстрактный тип данных $\langle A, \tau \rangle$ с именем `Name`. Пусть диаграмма переходов задает вид его управляющего автомата $A = \langle S, E, Z, s_0, \Delta \rangle$, а спецификация вложенного типа τ задает его имя, множество значений, инвариант, ограничение, а также имена, сигнатуры и семантику его методов в терминах предусловий и постусловий.

Построим спецификацию абстрактного типа данных τ' следующим образом.

- В качестве имени τ' выберем `Name`.

- Множество значений τ' опишем как декартово произведение множества значений τ и множества S. В частности, если множество значений описывается в терминах переменных спецификации, введем перечислимый тип σ с множеством значений S. Тогда набор переменных спецификации типа τ' будет содержать переменные из τ и переменную $state : \sigma$.

- Для описания множества допустимых значений будем использовать инвариант вложенного типа I_τ, а в качестве ограничения типа τ' — ограничение вложенного типа C_τ.

- Для каждого события $e \in E$ добавим к спецификации τ' метод, такой что:

  - имя метода совпадает с e;
  - сигнатура метода определяется по формулам:

$$\forall k \; e.\alpha_k = \min_{s \in S} \zeta(s, e).\alpha_k$$
$$e.\rho = \max_{s \in S} \zeta(s, e).\rho$$

  - семантика метода в терминах пред- и постусловий определяется по формулам:

$$e.pre = \bigvee_{s \in S} \Big((state^\wedge = s) \wedge \zeta(s, e).pre \Big)$$
$$e.post = \bigwedge_{s \in S} \Big((state^\wedge = s) \Rightarrow \zeta(s, e).post \wedge (state' = \delta(s, e)) \Big)$$

- Для каждого конструктора c типа τ добавим к τ' конструктор с тем же именем, сигнатурой и предусловием, изменив постусловие следующим образом:

$$\tau' :: c.post = \tau :: c.post \wedge (result'.state = s_0)$$

В приведенном выше алгоритме наиболее нетривиальный пункт — построение сигнатур методов τ'. Если $\forall e \in E \; \exists \alpha_e : \forall s \in S \; \zeta(s, e).\alpha = \alpha_e$, можно определить набор типов аргументов метода e как α_e. То же справедливо и для типа результата. Однако это слишком сильное требование: в действительности, для правильной работы любого метода автоматизированного типа аргументы e должны только содержать достаточно информации для вызова любого из сопоставляемых ему методов z, и наоборот, возвращаемое значение любого из

методов z должно быть пригодно для возврата из e. Поэтому от метода e и любого из сопоставляемых ему методов z целесообразно потребовать выполнения правил *контравариантности аргументов* и *ковариантности результата*:

$$\forall k \; e.\alpha_k \leqslant z.\alpha_k$$
$$z.\rho \leqslant e.\rho$$

Поскольку эти правила должны выполняться для любого z, которое может быть сопоставлено данному e, приходим к соотношениям для $e.\alpha$ и $e.\rho$, указанным в алгоритме 1. На рис. 9 изображен пример иерархий типов одного из аргументов (слева) и результата (справа). Отношение подтипизации задает лишь частичный порядок, поэтому общего подтипа соответствующих типов аргументов (или общего супертипа типов результата) может не существовать. В этом случае эквивалентный тип τ' не может быть построен и ААТД $\langle A, \tau \rangle$ является некорректным.

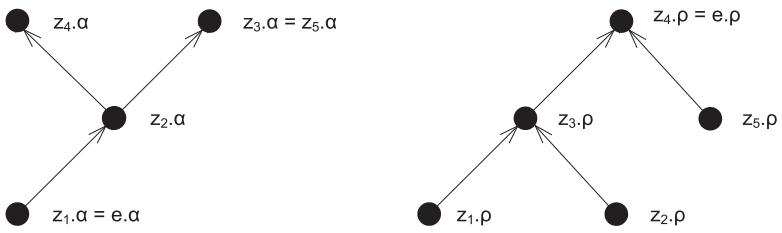

Рис. 9. Пример иерархий типов аргумента (слева) и результата (справа) методов-действий ААТД

Алгоритм 2 (Построение эквивалентного АТД для замкнутого ААТД). Пусть теперь имеется замкнутый ААТД $\langle A, \tau \rangle$ с именем `Name`, диаграмма переходов задает поведение его управляющего автомата

$$A = \langle S, E, Z, P, s_0, \pi, \Delta \rangle.$$

Для краткости введем обозначение $X(s, e) \equiv \{0, 1\}^{|\pi(s,e)|}$ для множества битовых векторов, возвращаемых набором предикатов $\pi(s, e)$ (если этот набор пуст, множество $X(s, e)$ состоит из единственного вектора нулевой длины). В алгоритме необходимо обращаться к конкретным значениям, возвращаемым методами-предикатами вложенного типа. Будем считать, что, если метод однозначно специфицирован и возвращает значение (это справедливо для всех пре-

дикатов), то его предусловие тривиально истинно, а постусловие содержит выражение вида $result = ret\_value$. Ниже используется запись $\pi(s,e).result = x$, которая является сокращением выражения

$$\bigwedge_{i=1}^{|\pi(s,e)|} (\pi(s,e)_i.result = x_i).$$

Если набор предикатов $\pi(s,e)$ пуст, будем считать, что это выражение тождественно истинно.

Построим спецификацию абстрактного типа данных τ' следующим образом.

- В качестве имени τ' выберем `Name`.
- Множество значений, инвариант и ограничение строятся так же, как и в разомкнутом случае.
- Для каждого события $e \in E$ добавим к спецификации τ' метод, такой что:
  - имя метода совпадает с e;
  - сигнатура метода определяется по формулам:

$$\forall k\ e.\alpha_k = \min_{s \in S}\Big\{ \min_{p \in \pi(s,e)} p.\alpha_k,\ \min_{x \in X(s,e)} \zeta(s,e,x).\alpha_k \Big\}$$

$$e.\rho = \max_{s \in S, x \in X(s,e)} \zeta(s,e,x).\rho$$

  - семантика метода определяется по формулам:

$$e.pre = \bigvee_{s \in S} \Big((state^{\wedge} = s) \wedge \bigvee_{x \in X(s,e)} \big((\pi(s,e).result = x) \wedge \zeta(s,e,x).pre \big) \Big)$$

$$e.post = \bigwedge_{s \in S} \Big((state^{\wedge} = s) \Rightarrow \bigwedge_{x \in X(s,e)} \big((\pi(s,e).result = x) \Rightarrow (\zeta(s,e,x).post \wedge (state' = \delta(s,e,x))) \big) \Big)$$

- Конструкторы τ' строятся так же, как и в разомкнутом случае.

Утверждение 1. *Автоматизированный абстрактный тип данных $\langle A, \tau \rangle$ (в общем случае замкнутый) и эквивалентный ему АТД τ' неотличимы по возвращаемым значениям.*

Доказательство. Пусть объект $o : \langle A, \tau \rangle$ создан конструктором c: его управляющий автомат находится в стартовом состоянии s_0, а объект вложенного типа содержит значение из множества V_0, определяемого постусловием конструктора. Пусть объект $u : \tau'$ также создан конструктором c, тогда по правилу преобразования конструкторов, множество его возможных значений $\{s_0\} \times V_0$.

Пусть перед вызовом метода m_i множество возможных вычислительных состояний объекта o — V_{i-1}, управляющих состояний — S_{i-1}, множество возможных значений объекта u — $S_{i-1} \times V_{i-1}$. Вызовем у объектов o и u метод m_i со списком аргументов a_i. Объект o последовательно вызовет методы-предикаты одного из наборов $\pi(s, m_i)$, $s \in S_{i-1}$ и, получив в результате один из возможных битовых векторов x, вызовет затем метод-действие $\zeta(s, m_i, x)$. Если предусловие этого метода не выполняется, то результат вызова $u.m_i$ не определен. По построению, предусловия метода $\tau' :: m_i$ эквивалентного АТД также не выполнится.

Пусть теперь предусловие $\zeta(s, m_i, x)$ выполнено (в этом случае предусловие $\tau' :: m_i$ также выполнено). Тогда множество возможных значений результата R_i и возможных новых значений объекта вложенного типа V_i определяются постусловием $\zeta(s, m_i, x)$, а множество возможных новых управляющих состояний — функцией переходов $S_i = \delta(s, m_i, x)$. По построению постусловие метода $\tau' :: m_i$ определяет то же множество возможных значений результата и множество возможных новых значений объекта вида $S_i \times V_i$. □

После преобразования автоматизированного типа данных в эквивалентный АТД, теряется информация об управляющих состояниях или, другими словами, разделение описания логики сложного поведения и описания его семантики. Однако, если спецификация ААТД приведена к такому виду, для нее могут быть доказаны различные свойства и отношения, характерные для обыкновенных типов данных и определенные в терминах их спецификации. Если при рассмотрении этих свойств учесть потерянную информацию о состояниях, свойства абстрактных типов можно переформулировать в терминах ААТД и в дальнейшем использовать непосредственно.

4.2. Отношение подтипизации для ААТД

Подтипизация является одним из важнейших инструментов объектно-ориентированного моделирования и программирования. По ряду причин, ее иногда путают с другим важным механизмом — *наследованием*, хотя это принципиально различные понятия. Подтипизация — отношение частичного порядка между типами, а наследование представляет собой языковой механизм, позволяющий определять одни классы с использованием других. Оба этих инструмента предназначены для повторного использования кода, однако наследование позволяет повторно использовать определения родительских классов, в то время как подтипизация — пользовательский код [20]. Во многих современных практических объектно-ориентированных языках концепция класса объекта и его типа не различаются, и с точки зрения эффективности реализации и простоты объектной модели отношение подтипизации определяется через наследование классов.

В связи с задачей эффективного сочетания объектной и автоматной парадигм разработки программного обеспечения, актуален вопрос о перенесении механизмов наследования и подтипизации на модели сущностей со сложным поведением [12, 13, 14].

С точки зрения автора, целесообразно определить отношение подтипизации для автоматизированных абстрактных типов данных, а затем на основе этого определения построить механизм наследования для языков, ориентированных одновременно на объектный и автоматный подходы.

Существует множество формальных определений подтипизации (по крайней мере столько же, сколько существует различных объектных моделей). Многие из них опираются на понятие интерфейсного типа и являются чисто «структурными» [18, 19, 20]. В других случаях подтипизация определятся для реализационных типов, это делается либо через механизм наследования, либо через установление между супертипом и подтипом некоторых синтаксических и семантических связей.

В основе всех определений подтипизации лежит интуитивный принцип *заместимости*, который, однако, также может быть сформулирован по-разному. Например, этот принцип для интерфейсных типов формулируется следующим образом: «Объект подтипа может быть использован в программе везде, где

ожидается объект супертипа без ошибок типа». Этот принцип не накладывает семантических ограничений на поведение подтипа. Поэтому для абстрактных типов данных чаще используется его усиленный вариант: «Объект подтипа может быть использован в программе везде, где ожидается объект супертипа, и его поведение „не удивит пользователя“». В этой формулировке не оговаривается важное условие: используется ли объект подтипа в программе только как объект супертипа (у него вызываются только методы, имеющиеся у супертипа) или в другом месте программы его точный тип известен? Во втором случае на поведение подтипа должны накладываться более сильные ограничения.

В настоящей работе за основу принимается принцип в формулировке Б. Лисков и Дж. Уинг, изложенный [16, 17]. Эта формулировка известна как *принцип подстановки Лисков* (*Liskov Substitution Principle* — LSP): «Любое свойство, которое может быть доказано из спецификации супертипа, может быть доказано и из спецификации подтипа». В упомянутых работах приведены два *формальных конструктивных* определения отношения подтипизации в терминах спецификаций АТД на языке Larch, каждое из которых в некотором смысле удовлетворяет LSP. В настоящей работе ограничимся рассмотрением одного из них (использующего ограничения типов):

Определение 8. Тип $\omega = \langle W, N \rangle$ является *подтипом* $\tau = \langle V, M \rangle$ (пишут $\omega \leqslant \tau$), если существует функция абстракции $\mathcal{A}\colon W \to V$ и функция переименования $\mathcal{R}\colon N \to M$, такие что:

1. Выполняется *правило инварианта*: $\forall w \in W\ I_\omega(w) \Rightarrow I_\tau(\mathcal{A}(w))$.

2. Для каждого метода $\omega :: n \in dom(\mathcal{R})$ и соответствующего ему метода $\tau :: m = \mathcal{R}(n)$ выполняется:

   - *Правило сигнатур.*
     - *Контравариантность аргументов.* Количество аргументов m и n одинаково. Пусть α_i набор типов аргументов метода m, а β_i — соответствующий набор метода n. Тогда $\forall i\ \alpha_i \leqslant \beta_i$.
     - *Ковариантность результата.* Пусть результат m имеет тип γ, а результат n — тип δ, тогда $\delta \leqslant \gamma$.

- *Правило семантики.* Для всех объектов $u : \omega$:

$$m.pre[\mathcal{A}(u^\wedge)/u^\wedge] \Rightarrow n.pre$$
$$n.post \Rightarrow m.post[\mathcal{A}(u^\wedge)/u^\wedge, \mathcal{A}(u')/u'].$$

3. Выполняется *правило ограничений*: для любого объекта $u : \omega$ $C_\omega(u) \Rightarrow C_\tau[\mathcal{A}(u_c)/u_c, \mathcal{A}(u_d)/u_d]$

Отметим, что в этом определении функция абстракции может быть частичной на множестве значений (однако, она должны быть определена для всех *допустимых* значений, то есть для значений, удовлетворяющих инварианту). Она может не быть сюръекцией (могут существовать такие значения супертипа, которым не соответствует ни одно значение подтипа) и инъекцией (одному значению супертипа может соответствовать несколько значений подтипа). Функция переименования также может быть частичной (у подтипа могут быть «дополнительные» методы), но должна быть сюръекцией и, из соображений однозначности вызова метода подтипа через интерфейс супертипа, инъекцией (хотя для определения это требование несущественно).

Рассмотрим теперь применение этого определения к типам τ' и ω', эквивалентным автоматизированным типам $\langle A^1, \tau\rangle$ и $\langle A^2, \omega\rangle$ соответственно.

При этом сначала рассмотрим случай, когда оба ААТД разомкнуты. Управляющие автоматы ААТД имеют вид: $A^i = \langle S^i, E^i, Z^i, s_0^i, \Delta^i\rangle, i = 1, 2$, а их вложенные типы — $\tau = \langle V, M\rangle$, $\omega = \langle W, N\rangle$.

Для того, чтобы тип ω' являлся подтипом τ', во-первых, необходима функция абстракции $\mathcal{A}'$, соблюдающая инвариант. Из соображений разделения управляющей и вычислительной компонент ААТД, будем считать, что в данном случае эта функция может быть разложена на две составляющие — функцию абстракции вложенных типов $\mathcal{A}\colon W \to V$ и функцию абстракции состояний $\mathcal{S}\colon S^2 \to S^1$. Поскольку все управляющие состояния являются допустимыми, то функция $\mathcal{S}$ определена для всех $s \in S^2$, а правило инварианта можно сформулировать, используя только функцию $\mathcal{A}$:

$$\forall w \in W\ I_\omega(w) \Rightarrow I_\tau(\mathcal{A}(w))$$

Во-вторых, требуется функция переименования $\mathcal{R}\colon E^2 \to E^1$. Она может быть частичной — автоматизированный подтип может обрабатывать больше событий, чем его автоматизированный супертип.

Рассмотрим взаимосвязь между методом e типа ω' и соответствующим ему методом $g = \mathcal{R}(e)$ супертипа τ'. В соответствии с определением отношения подтипизации действует правило контравариантности аргументов:

$$\forall k \; g.\alpha_k \leqslant e.\alpha_k,$$

или в терминах аргументов методов вложенных типов:

$$\forall k \; \min_{s \in S^1} \zeta^1(s, g).\alpha_k \leqslant \min_{s \in S^2} \zeta^2(s, e).\alpha_k$$

Таким же образом можно записать правило ковариантности результата:

$$e.\rho \leqslant g.\rho$$
$$\max_{s \in S^2} \zeta^2(s, e).\rho \leqslant \max_{s \in S^1} \zeta^1(s, g).\rho$$

Кроме того, для рассматриваемой пары методов e и g действует правило семантики. В соответствии с этим правилом для любого объекта $u : \omega'$:

$$g.pre[\mathcal{A}'(u^\wedge)/u^\wedge] \Rightarrow e.pre$$

Обозначая через $state$ управляющее состояние объекта u, а через $y : \omega$ вычислительную компоненту u, это выражение можно записать следующим образом:

$$\bigvee_{s \in S^1} (\mathcal{S}(state^\wedge) = s) \wedge \zeta^1(s, g).pre[\mathcal{A}(y^\wedge)/y^\wedge] \Rightarrow \bigvee_{s \in S^2} (state^\wedge = s) \wedge \zeta^2(s, e).pre$$

Поскольку управляющее состояние может одновременно иметь только одно значение, из приведенного выше выражения следует:

$$\forall s \in S^2 \; \zeta^1(\mathcal{S}(s), \mathcal{R}(e)).pre[\mathcal{A}(y^\wedge)/y^\wedge] \Rightarrow \zeta^2(s, e).pre$$

Аналогичные рассуждения можно привести относительно постусловий:

$$e.post \Rightarrow g.post[\mathcal{A}'(u^\wedge)/u^\wedge, \mathcal{A}'(u')/u'];$$
$$\left(\bigwedge_{s \in S^2} (state^\wedge = s) \Rightarrow (\zeta^2(s, e).post \wedge (state' = \delta^2(s, e))) \right) \Rightarrow$$
$$\left(\bigwedge_{s \in S^1} (\mathcal{S}(state^\wedge) = s) \Rightarrow (\zeta^1(s, g).post[\mathcal{A}(y^\wedge)/y^\wedge, \mathcal{A}(y')/y'] \wedge (\mathcal{S}(state') = \delta^1(s, g)) \right);$$
$$\forall s \in S^2 \; \zeta^2(s, e).post \Rightarrow \zeta^1(\mathcal{S}(s), \mathcal{R}(e)).post[\mathcal{A}(y^\wedge)/y^\wedge, \mathcal{A}(y')/y']$$

и, кроме того, при абстракции состояний сохраняется функция переходов:

$$\forall s \in S^2 \; \mathfrak{S}(\delta^2(s, e)) = \delta^1(\mathfrak{S}(s), \mathcal{R}(e))$$

Рассмотрим теперь правило ограничений. В соответствии с определением требуется

$$C_{\omega'}(u) \Rightarrow C_{\tau'}[\mathcal{A}'(u_c)/u_c, \mathcal{A}'(u_d)/u_d],$$

что равносильно

$$C_\omega \Rightarrow C_\tau.$$

Исходя из приведенных рассуждений, можно сформулировать простое определение отношения подтипизации для разомкнутых автоматизированных абстрактных типов данных.

Определение 9. Разомкнутый ААТД $\langle A^2, \omega \rangle$ ($\omega = \langle W, N \rangle$) является *подтипом* разомкнутого ААТД $\langle A^1, \tau \rangle$ ($\tau = \langle V, M \rangle$, $A^i = \langle S^i, E^i, Z^i, s_0^i, \Delta^i \rangle$), если существует функция абстракции состояний $\mathfrak{S}\colon S^2 \to S^1$ и функция переименования событий $\mathcal{R}\colon E^2 \to E^1$, такие что:

- $\mathfrak{S}(s_0^2) = s_0^1$
- $\forall s \in S^2 \; \forall e \in dom(\mathcal{R}) \; \mathfrak{S}(\delta^2(s, e)) = \delta^1(\mathfrak{S}(s), \mathcal{R}(e))$

Кроме того, ω является подтипом τ (в смысле определения 8), и для их функции переименования $\mathcal{R}_{\omega\tau}$ справедливо:

- $\forall s \in S^2 \; \forall e \in dom(\mathcal{R}) \; \mathcal{R}_{\omega\tau}(\zeta^2(s, e)) = \zeta^1(\mathfrak{S}(s), \mathcal{R}(e))$

Утверждение 2. *Если разомкнутый ААТД $\langle A^2, \omega \rangle$ является подтипом разомкнутого ААТД $\langle A^1, \tau \rangle$ в смысле определения 9, то эквивалентные им абстрактные типы данных ω' и τ' также связаны отношением подтипизации (в смысле определения 8).*

Доказательство. Утверждение непосредственно следует из рассуждений, предшествующих определению подтипизации разомкнутых ААТД. □

Руководствуясь аналогичными рассуждениями, можно сформулировать определение отношения подтипизации для замкнутых автоматизированных абстрактных типов. Однако в замкнутом случае полезной представляется возможность сопоставлять одному переходу автомата, входящего в супертип, несколько

переходов автомата подтипа: по одному и тому же событию, но по разным наблюдениям. Действия на этих переходах должны быть связаны определенным образом с действием на соответствующем переходе автомата супертипа. В результате приходим к требованию отношения подтипизации между вложенными типами, в котором функция переименования не обязательно является инъекцией.

Определение 10. Замкнутый ААТД $\langle A^2, \omega \rangle$ ($\omega = \langle W, N \rangle$) является *подтипом* замкнутого ААТД $\langle A^1, \tau \rangle$ ($\tau = \langle V, M \rangle$, $A^i = \langle S^i, E^i, Z^i, P^i, s_0^i, \pi^i, \Delta^i \rangle$), если существует функция абстракции состояний $\mathcal{S}\colon S^2 \to S^1$ и функция переименования событий $\mathcal{R}\colon E^2 \to E^1$, такие что:

- $\mathcal{S}(s_0^2) = s_0^1$
- для функций переходов справедливо соотношение:

$$\forall s \in S^2\ \forall e \in dom(\mathcal{R})\ \forall x^2 \in X^2(s, e)$$
$$\exists x^1 \in X^1(\mathcal{S}(s), \mathcal{R}(e)) : \mathcal{S}(\delta^2(s, e, x^2)) = \delta^1(\mathcal{S}(s), \mathcal{R}(e), x^1)$$

Кроме того, ω является подтипом τ, их функция переименования $\mathcal{R}_{\omega\tau}$ необязательно является инъекцией. Эта функция обладает следующим свойством:

$$\forall s \in S^2\ \forall e \in dom(\mathcal{R})\ \forall x^2 \in X^2(s, e)$$
$$\exists x^1 \in X^1(\mathcal{S}(s), \mathcal{R}(e)) : \mathcal{R}_{\omega\tau}(\zeta^2(s, e, x^2)) = \zeta^1(\mathcal{S}(s), \mathcal{R}(e), x^1)$$

В качестве примера рассмотрим два подтипа автоматизированного типа `LoanableItem`, моделирующего предмет, который может быть взят на прокат (разд. 3.2).

Пример 2. Один из подтипов «прокатной единицы» — *книга* (`Book`). Этот автоматизированный тип управляется в точности тем же автоматом, что и его супертип (рис. 8), а его вложенный АТД `BookNested` является подтипом `LoanableItemNested`. Поведение книги ограничено по сравнению с поведением прокатной единицы тем, что книгу можно взять ровно на две недели.

```
class BookNested {
    // Переменные спецификации LoanableItemNested
```

```
    spec const Date twoWeeks;
public:
    void SaveUser(User user, Date currentDate) {
        requires: user != noUser;
        modifies: this;
        ensures: currentUser' == user &&
            returnDate == currentDate + twoWeeks;
    }
    // Остальные методы LoanableItemNested
}
```

Можно показать, что АТД `Book` является подтипом `LoanableItem`. Поскольку их управляющие автоматы совпадают, то функции абстракции состояний и переименования событий являются тождественными (отображают любое значение аргумента в это же значение). Функция переименования для вложенных типов также тождественна и необходимое для нее описанное в определении свойство выполняется.

Пример 3. Второй рассматриваемый автоматизированный подтип прокатной единицы — *видеомагнитофон*. Когда видеомагнитофон возвращают из проката, он должен пройти технический контроль. Поэтому к управляющему автомату АТД `VideoRecorder` добавим состояние `Maintenance` (рис. 10).

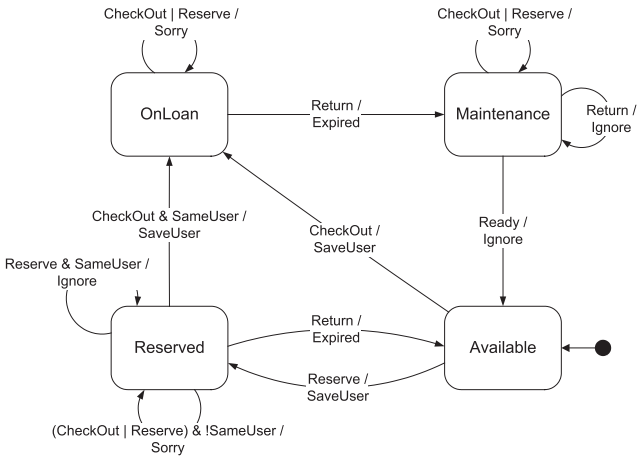

Рис. 10. Управляющий автомат АТД `VideoRecorder`

Вложенный тип `VideoRecorderNested` является подтипом вложенного АТД прокатной единицы и ограничивает срок проката видеомагнитофона тремя днями.

```
class VideoRecorderNested {
    // Переменные спецификации LoanableItemNested
    spec const Date threeDays;
public:
    void SaveUser(User user, Date currentDate) {
        requires: user != noUser;
        modifies: this;
        ensures: currentUser' == user &&
            returnDate == currentDate + threeDays;
    }
    // Остальные методы LoanableItemNested
}
```

Будем считать, что функция абстракции состояний в этом случае отображает новое управляющее состояние `Maintenance` в состояние `OnLoan` автомата супертипа.

Заключение

В работе была поставлена задача построения объектно-ориентированной модели сущности со сложным поведением. При этом были рассмотрены простейшие известные модели таких сущностей — абстрактные вычислители. Проведено их сравнение и выделены несколько разновидностей, существенных для моделирования и программирования (открытые и закрытые, разомкнутые и замкнутые, активные и пассивные).

Для придания точного смысла термину «объектно-ориентированный», в работе проведены обзор и классификация существующих теоретических и практических объектных моделей. На их основе построена собственная объектная модель, основным понятием которой является абстрактный тип данных (АТД).

В качестве модели сущности со сложным поведением в работе предложен автоматизированный абстрактный тип данных (ААТД). ААТД представляет собой тип данных, логика которого описывается конечным автоматом, а семантика — вложенным абстрактным типом данных. Предложен метод спецификации автоматизированных типов с использованием существующих нотаций описания конечных автоматов и АТД. В работе приведен алгоритм преобразования спецификации ААТД в спецификацию эквивалентного ему обыкновенного абстрактного типа.

Кроме того, в работе апробирована техника перенесения на ААТД свойств и отношений, характерных для обыкновенных типов данных. Подробно рассмотрено отношение подтипизации для автоматизированных типов. Предложено конструктивное определение этого отношения в терминах спецификаций и показана его связь с аналогичным определением для обыкновенных абстрактных типов.

Список литературы

1. *Guttag J. V., Horning J. J.* Larch: Languages and Tools for Formal Specification. New York: Springer, 1993. 571 p.

2. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002. 528 с.

3. *Буч Г., Рамбо Дж., Джекобсон И.* Язык UML: Руководство пользователя. СПб.: Питер, 2004. 430 с.

4. *Harel D., Polity M.* Modeling Reactive Systems with Statecharts. The Statemate Approach. New York: McGraw-Hill, 1998. 258 p.

5. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

6. *Туккель Н. И., Шалыто А. А.* Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. http://is.ifmo.ru (раздел «Проекты»).

7. *Шалыто А. А., Наумов Л. А.* Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов. http://is.ifmo.ru (раздел «Статьи»).

8. *Martin R.* Three-Level FSM // Pattern Languages of Program Design, 1995.

9. *Yacoub S. M., Ammar H. H.* Finite State Machine Patterns // Proceedings of EuroPLoP. 1998.

10. *Adamczyk P.* The anthology of the finite state machine design patterns // The 10th Conference on Pattern Languages of Programs, 2003.

11. *Adamczyk P.* Selected patterns for implementing finite state machines // The 11th Conference on Pattern Languages of Programs, 2004.

12. *Sane A, Campbell R.* Object-oriented state machines: Subclassing, composition, delegation, and genericity // OOPSLA, 1995.

13. *Шопырин Д. Г.* Методы объектно-ориентированного проектирования и реализации программного обеспечения реактивных систем. Диссертация.

14. *Harel D., Kupferman O.* On the Behavioral Inheritance of State-Based Objects // The 34th International Conference on Component and Object Technology. Santa Barbara, C.A. 2000.

15. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

16. *Liskov B., Wing J.* Family Values: A Behavioral Notion of Subtyping // ACM Trans. Program. Lang. Syst., 1994.

17. *Liskov B., Wing J.* Family Values: A Semantic Notion of Subtyping. Technical Report LCS TR-562, MIT, 1992.

18. *Cardelli L., Wegner P.* On Understanding Types, Data Abstraction, and Polymorphism // ACM Computing Surveys, Vol.17, No.4, pages 471522. ACM, 1985.

19. *Fisher K., Mitchell J. C.* Notes on Typed Object-Oriented Programming // TACS94, volume 789 of LNCS, pages 844–885. Springer-Verlag, 1994.

20. *Fisher K., Mitchell J. C.* On the Relationship between Classes, Objects and Data Abstraction // Theory and Practice of Object Systems, 4(1):3–25, 1998.

21. *Leavens G. T.* Larch/C++ An Interface Specification Language for C++ // http://www.cs.iastate.edu/ leavens/larchc++.html

22. *Bajaj S.* Design Patters: Solidify Your C# Application Architecture with Design Patterns // MSDN Magazine, 2001.

23. *Шалыто А. А., Туккель Н. И.* От Тьрюингова программирования к автоматному // Мир ПК, 2:144-149,2002.

24. *Dyson P., Anderson B.* State Patterns // Pattern Languages of program design 3. Addison-Wesley, 1998.

Оглавление