УДК 004.434
doi:10.15217/issn1684-8853.2016.6.50

КООПЕРАТИВНОЕ ВЗАИМОДЕЙСТВИЕ АВТОМАТНЫХ ОБЪЕКТОВ

Ф. А. Новиков[а], *доктор техн. наук, профессор*
И. В. Афанасьева[б, в], *ведущий инженер, аспирант*
[а]*Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, РФ*
[б]*Специальная астрофизическая обсерватория РАН, Нижний Архыз, РФ*
[в]*Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, Санкт-Петербург, РФ*

Цель исследования: *создание модели описания поведения, нацеленной на достижение более высоких показателей надежности и производительности при проектировании архитектуры и реализации реагирующих и распределенных систем по сравнению с традиционными методами. Побочной целью является создание удобного графического языка публикаций для описания параллельных алгоритмов и распределенных реагирующих систем.* ***Методы:*** *для описания поведения использованы диаграммы автомата (графы переходов состояний) унифицированного языка моделирования UML, расширенные специальными стереотипами, портами и интерфейсами. Описание предлагаемого графического языка проведено с помощью метамоделирования средствами диаграмм классов UML.* ***Результаты:*** *разработана модель поведения, использующая графы переходов состояний и относящаяся к парадигме автоматного программирования. Детально описаны свойства и преимущества предлагаемой модели в классе асинхронных распределенных реагирующих систем, а именно: доказана алгоритмическая полнота, определен наглядный графический язык, приведены демонстрационные примеры и указаны перспективы развития.* ***Практическая значимость:*** *представленная модель была успешно применена на практике при разработке специализированного программного обеспечения управления высокоточным научным оборудованием в наблюдательной астрономии, а также использована как высокоуровневое средство описания поведения в автоматном методе определения языков предметной области.*

Ключевые слова *— модель поведения, автоматное программирование, граф переходов состояний, унифицированный язык моделирования, диаграммы автомата, диаграммы классов, параллельное поведение, архитектура реагирующих систем.*

Введение

В современных информационных технологиях и программировании господствует взгляд на программное обеспечение как на системы взаимодействующих программных объектов. Программный объект имеет состояние и поведение. Состояние определяется текущими значениями свойств объекта, а поведение задается методами объекта. При этом практически все сходятся в том, что *свойства* удобнее всего представлять парами «ключ — значение», и по этому вопросу нет разночтений. Для задания *поведения*, напротив, используется великое множество различных подходов, и это верный признак того, что среди известных подходов нет наилучшего во всех возможных случаях. Каждый способ описания поведения программных объектов имеет свои достоинства и свои недостатки, которые поразному проявляются в различных контекстах.

Здесь мы представляем еще один способ описания поведения и соответствующий графический язык, которые имеют существенные преимущества при определенных условиях.

Предлагаемая модель возникла не на пустом месте. Уже более четверти века развивается *парадигма автоматного программирования* — подход к описанию поведения, основанный на явном выделении состояний. Несравненные заслуги в развитии и продвижении этого подхода принадлежат профессору А. А. Шалыто. Автоматное программирование имеет целый ряд неоспоримых достоинств, отмеченных в работах [1–3] и в других источниках:

— простое и ясное математическое основание в форме теории автоматов;

— устойчивую традицию использования концепции состояний при описании поведения различных устройств во многих инженерных областях;

— возможность несложной, а потому эффективной программной реализации на любой платформе.

По нашему мнению, в настоящее время автоматное программирование является наиболее перспективной парадигмой создания программного обеспечения, в особенности программного обеспечения сложных технических устройств.

Многие авторы с успехом применяли автоматное программирование в различных областях и с различными целями, что вызвало к жизни разнообразие форм автоматного программирования. При этом, естественно, менялась нотация, семантика, прагматика и т. д. при неизменности центральной идеи явного выделения состояний. Разнообразие форм является достоинством парадигмы, если известно, в каких случаях и при каких условиях целесообразно применять ту или

иную форму. В противном случае разнообразие форм может сбивать с толку.

В данной статье мы ставим цель изложить теоретические основы и предложить развитую нотацию конкретной формы автоматного программирования для *асинхронных распределенных реагирующих систем* в рамках нашего опыта практического применения автоматного программирования [4, 5] и использования предлагаемого языка [6].

Действия и состояния

Прежде всего необходимо точно определить, что есть *поведение* и что есть *состояние* программного объекта. При определении поведения мы полагаем, что имеется некоторая базовая машина, обладающая способностью производить элементарные *действия*. Например, при определении поведения с помощью машины Тьюринга элементарными действиями являются смена состояний в управляющем устройстве машины, движения ленточной головки, запись и чтение символов на ленте [7]. Каждая конкретная фактически выполненная последовательность действий называется *протоколом* (или экземпляром выполнения, или вычислением). При этом последовательности действий в протоколах отнюдь не произвольны. Например, на каждом такте работы машины Тьюринга сначала читается текущий символ на ленте, затем записывается новый символ и меняется состояние, а потом уже производится движение головки. Конечное описание некоторого потенциально бесконечного подмножества множества протоколов называется *моделью поведения*, или просто *поведением* [8]. В настоящее время доминирующим классом таких описаний являются алгоритмы [9], поскольку все множество возможных выполнений любого алгоритма как раз образует некоторое множество протоколов. При этом нас интересует описание множеств таких протоколов, которые обладают определенными свойствами. Например, в случае машины Тьюринга нас интересуют только протоколы, которые начинаются, когда машина находится в начальном состоянии и на ленте записаны исходные данные, а заканчиваются, когда машина находится в заключительном состоянии. Таким образом, в первом приближении можно считать, что *поведение* — это описание некоторого множества протоколов выполнения. Первое приближение нуждается в трех существенных уточнениях.

Во-первых, заметим, что мы считали протоколы линейно упорядоченными множествами (последовательностями) только для простоты начального изложения. На самом деле действия при выполнении упорядочены частично. Например, в машине Тьюринга чтение символа с ленты обязательно предшествует перемещению головки, но смену состояния и запись символа на ленту можно выполнять в любом порядке, т. е. параллельно. Параллельное поведение не является экзотикой, напротив, параллельное поведение встречается в жизни чаще последовательного. Мы связываем последовательное выполнение с линейным порядком, а параллельное выполнение — с частичным порядком. Известно, что всякий линейный порядок является частичным, и всякий частичный порядок может быть дополнен до линейного порядка, но не единственным образом [10]. В этой неоднозначности, по нашему мнению, кроется одна из проблем описания параллельного поведения.

Во-вторых, считать действие элементарным — это также некоторая условность. Например, в обычном компьютере при программировании машинная команда считается элементарным действием, но при конструировании процессора одна машинная команда — это сложное поведение, включающее множество действий по загрузке команды, дешифровке кода операции, вычислению адресов в памяти, загрузке данных в регистры, выполнению самой операции в процессоре, выгрузке данных из сумматора и т. д. Поэтому описание поведения никогда не может быть задано абсолютно, но только относительно набора действий базовой машины, которая выбирается произвольно.

В-третьих, описание поведения, как правило, определяет потенциально бесконечное множество протоколов, причем актуальное построение всего множества невозможно и не требуется. Требуется по заданным описанию поведения и входным данным определять частично упорядоченное множество действий, соответствующих данному описанию, т. е. требуется, иначе говоря, выполнять алгоритм или осуществлять поведение. При этом для заданного описания поведения (включая заданные входные данные, если они нужны) частично упорядоченное множество действий может определяться однозначно, как единственная последовательность действий (возможно, пустая, если поведение неосуществимо), и тогда поведение называется детерминированным, или же может определяться неоднозначно, как непустое семейство частично упорядоченных множеств действий, и тогда поведение называется недетерминированным. Таким образом, мы приходим к следующим определениям. *Поведение* — это конечное описание иерархии семейств частично упорядоченных множеств действий. *Модель поведения* — это язык для записи таких описаний.

Обратимся теперь к понятию *состояния*. Прежде всего заметим, что состояние и поведение неразрывно связаны: текущее состояние объекта определяется поведением объекта в прошлом (уже

выполненные действия) и предопределяет поведение объекта в будущем (если временно исключить из рассмотрения случайное поведение). Разумеется, значения всех переменных программного объекта полностью определяют его состояние. Например, конфигурация [7] машины Тьюринга определяется тем, какой символ на ленте обозревается, всеми символами, которые находятся слева и справа от текущего, а также тем, какая строка управляющий таблицы является текущей. Ясно, что конфигураций машины Тьюринга очень много. Следуя А. А. Шалыто [2], мы называем такие состояния *вычислительными*. Если положить в основу описания поведения именно вычислительные состояния, то естественным образом возникает теория машин абстрактных состояний [11], которая с успехом применяется для описания поведения, в частности, вычислительных программ. Однако строки управляющей таблицы машины Тьюринга также называются состояниями, и это не случайно. Такие состояния, вслед за А. А. Шалыто, мы называем *управляющими*. Заметим, что управляющих состояний намного меньше, нежели вычислительных. Конечно, когда машина Тьюринга находится в одном определенном управляющем состоянии, протоколы будущего поведения машины зависят от символов на ленте, но интуитивно мы склонны считать все протоколы в одном состоянии сходными между собой. Отношение сходства протоколов для разных базовых машин можно определять по-разному, но во всех случаях оно оказывается рефлексивным, симметричным и транзитивным, т. е. отношением эквивалентности, а значит, определяет фактор-множество [12]. Так мы приходим к определению: *множество управляющих состояний* — это фактор-множество множества вычислительных состояний по отношению сходства протоколов. Какие протоколы поведения считать сходными — вопрос субъективный, а потому множество управляющих состояний, в отличие от множества вычислительных состояний, не предопределяется поведением однозначно. В этой статье далее термин «состояние» означает управляющее состояние и слово «управляющее» опускается.

Способы описания поведения

Предметом статьи является поиск ответа на вопрос: как наиболее целесообразно выбрать модель поведения, если известен класс поведений, которые требуется описать, или хотя бы некоторые свойства этого класса. Прежде чем предлагать наше решение по выбору, необходимо ответить на вопрос, из чего, собственно, можно выбирать.

Известно множество моделей поведения [13], постоянно предлагаются новые модели и модифицируются уже известные, поэтому исчерпывающий обзор в одной статье вряд ли возможен. На фоне повсеместного распространения UML [14], для подавляющего большинства моделей поведения разработаны удобные графические языки, которыми мы также пользуемся [15]. Исходя из данных нами выше определений поведения и состояния, мы предлагаем следующую классификацию моделей поведения:

— явное выделение состояний;
— потоки управления и данных;
— последовательности сообщений.

В моделях поведения с явным выделением состояний [1] явно присутствуют управляющие состояния, а последовательность действий, напротив, в явном виде не показывается. К этому классу относятся: конечный автомат, машина Тьюринга, сеть Петри, язык SDL. Обычно в качестве графического языка в моделях поведения с явным выделением состояний используют диаграммы графа переходов состояний [16].

В моделях поведения с потоками управления явно присутствует описание класса протоколов (множества последовательностей действий) и класса вычислительных состояний (описание типизированных переменных), а управляющие состояния, напротив, скрыты. К этому классу относятся: блок-схема, диаграмма потоков данных, императивные языки программирования. В этом случае графическим языком являются диаграммы графа деятельности. Наиболее проработанная нотация для потоковых моделей поведения сегодня — это, по нашему мнению, диаграмма деятельности в UML 2 [17].

В моделях поведения с последовательностью сообщений показываются непосредственно протоколы, т. е. последовательности действий, упорядоченные во времени. Фактически поведение описывается индуктивно, на примерах. К этому классу относятся все диаграммы взаимодействия UML, нотация BPMN. Графические языки весьма разнообразны, но чаще всего используются вариации языка MSC.

Наряду с предложенной нами классификацией по внутренним свойствам моделей, давно используются классификации, основанные на свойствах классов описываемых поведений. В известной классификации Д. Харела [16] выделяются реагирующие системы, которые реагируют на изменения внешней среды, и трансформационные системы, которые преобразуют входные данные в выходные. Реагирующие системы часто являются постоянно действующими, а трансформационные системы, напротив, запускаются по мере надобности. В трансформационных системах для описания поведения достаточно описать связь между входом и выходом, т. е. связь между начальным и конечным вычислительными состо-

яниями. Последовательность действий в трансформационных системах не имеет особого значения, если результат вычислен правильно. Для описания связи между значениями переменных на входе и на выходе трансформационного алгоритма достаточно, например, языка исчисления предикатов, и предлагаемая нами модель поведения в этом случае ничего дополнительно не дает, поэтому трансформационные системы в этой статье не рассматриваются.

При реализации реагирующих систем непрерывного действия в большинстве случаев используют одну из двух схем, представленных на рис. 1. Схему на рис. 1, *а* уместно называть схемой с мониторингом, или *синхронной*, а схему на рис. 1, *б* — схемой с прерываниями, или *асинхронной*.

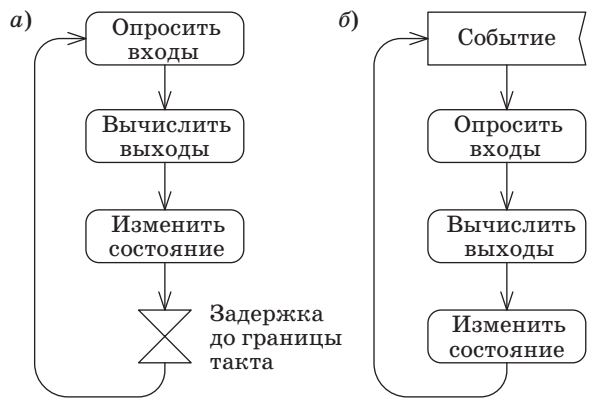
В принципе, обе схемы пригодны для реализации требуемого поведения реагирующих систем. Отличие состоит в том, что обработка прерываний, с одной стороны, требует дополнительных накладных расходов на организацию очередей событий, переключение контекстов и т. д. С другой стороны, при работе по прерываниям можно не бездействовать, ожидая события, а занять вычислительные ресурсы какой-либо другой полезной работой. Критерием для выбора одной из двух схем являются следующие количественные соотношения:

— запас производительности, т. е. насколько доступная вычислительная мощность превосходит минимальную мощность, необходимую для выполнения действий реагирующей системы (если вычислительной мощности недостаточно для выполнения необходимых действий, то реагирующая система, очевидно, невозможна);

— степень распределенности, т. е. вся реагирующая система сосредоточена в одном месте или имеется несколько взаимодействующих подсистем;

— номенклатура событий, т. е. как много различных сигналов должна обрабатывать система.

Можно заметить, что ранее запас по производительности был невелик, многие реагирующие системы работали на пределе возможностей аппаратуры, отчего распределенные системы встречались не так часто и реагирующие системы выполняли достаточно узкий набор функций. В настоящее время вычислительная мощность компьютеров выросла значительно, поэтому распределенные системы используются повсеместно и наблюдается тенденция совмещения множества разнородных функций в одной системе. Это хорошо заметно на примере таких реагирующих систем, как кнопочный телефон старого образца и современный смартфон. Поэтому сегодня реагирующие системы чаще оказываются асинхронными, нежели синхронными.



■ ***Рис. 1.*** Синхронные (*а*) и асинхронные (*б*) реагирующие системы

В этой статье предлагается модель поведения прежде всего асинхронных распределенных реагирующих систем (такие системы иногда называют мультиагентными).

При описании асинхронных распределенных реагирующих систем, как правило, применяются модели поведения с явным выделением состояний. Мы затрудняемся дать формальное истолкование причин этого явления и принимаем его как эмпирический факт. Предлагаемая модель также относится к классу моделей с явным выделением состояний и, как показано в последующих разделах статьи, проявляет свои преимущества именно в случае асинхронных распределенных многофункциональных реагирующих систем.

Моделирование поведения системами автоматных объектов

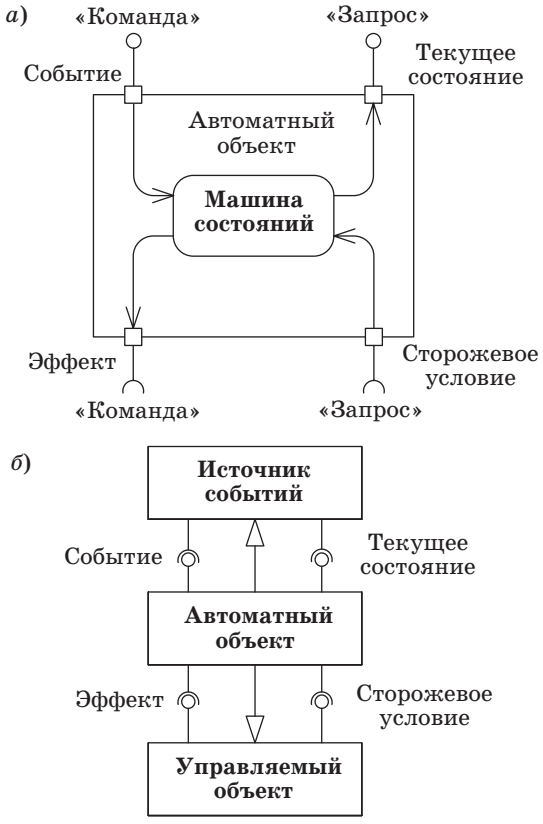
Центральным понятием предлагаемой модели поведения является *автоматный объект*. Автоматный объект имеет много общего с традиционным конечным автоматом [7], но в то же время обладает существенными отличиями. Так же, как и конечный автомат в форме Мили [18], автоматный объект задается графом переходов состояний, где на каждом переходе указывается *событие* (иногда говорят входное воздействие), инициирующее этот переход, а также могут указываться *сторожевое условие* выполнения перехода и *эффект* (иногда говорят выходное воздействие), производимый в результате перехода. Мы используем следующие расширения модели, каждое из которых выводит предлагаемую модель автоматного объекта из класса конечных автоматов:

— события являются не одиночными символами — элементами конечного алфавита, но могут иметь параметры и могут передавать в автоматный объект произвольный объем информации;

— сторожевые условия могут иметь произвольную сложность и зависеть от текущих значений произвольного числа локальных переменных, а не только от текущего состояния;

— эффекты также являются не одиночными символами — элементами конечного алфавита, но могут иметь параметры и могут записывать во внешней памяти произвольный объем информации.

Рассмотрим ключевой для дальнейшего изложения рисунок. Схема автоматного объекта, комбинирующая концепции машины состояний, компонентов, портов и интерфейсов унифицированного языка моделирования UML [19], представлена на рис. 2, *а*. Основное нововведение состоит в том, что мы мыслим машину состояний инкапсулированной в компонент и явно указываем порты и интерфейсы, через которые автоматный объект взаимодействует с внешним миром. Схема связей, традиционная для систем управления, реагирующих на внешние события [2, 3], показана на рис. 2, *б*. На среднем уровне схемы находится управляющий автомат. Входным алфавитом автомата являются события, посылаемые источником событий. В зависимости от полученных событий и результатов проверки сторожевых условий на переходах автомат меняет свое текущее состояние и выполняет действия на переходах (эффекты), адресованные объекту управления. В традиционной схеме управляющие автоматы, источники событий и объекты управления выглядят неравноправными и разнотипными объектами. В предлагаемой модели источники событий и объекты управления также являются автоматными объектами, что достигается введением соответствующих отношений обобщения (см. рис. 2, *б*).



■ ***Рис. 2***. Автоматный объект (*а*) и его связи (*б*)

Модель автоматного объекта (см. рис. 2, *а*) построена, исходя из следующих соображений: в соответствии с принципом Б. Мейера [20] операции интерфейса любого объекта следует разделить на *запросы*, доставляющие значения и не меняющие состояние объекта (стереотип «запрос»), и *команды*, меняющие состояние объекта, но не доставляющие значений (стереотип «команда»). Далее, считая обязательным указание для каждого объекта не только предоставляемых, но и требуемых интерфейсов, получаем ровно четыре возможных вида интерфейсов взаимодействия между объектами: предоставляемые команды и требуемые команды, предоставляемые запросы и требуемые запросы. Перечень комбинаций является исчерпывающим, и в этом смысле предлагаемая модель является окончательной и совершенной, поскольку учитывает вообще все возможные типы взаимодействия объектов, удовлетворяющие принципу Б. Мейера.

Применяя это наблюдение к автоматам, получаем ровно четыре возможных вида интерфейсов между автоматом, его источником событий и объектом управления (см. рис. 2, *а*):

— события на переходах автомата являются предоставляемыми командами автомата, аргументы событий, если они есть, инициализируют локальные переменные автомата;

— сторожевые условия на переходах являются логическими выражениями над значениями, которые доставляют требуемые запросы к объекту управления;

— эффекты — это требуемые команды объекта управления, в качестве аргументов могут передаваться значения локальных переменных автомата;

— автомат может предоставлять запросы о своем текущем состоянии и значениях других локальных переменных.

Мы утверждаем, что в практических случаях из соображений декомпозиции удобнее описывать поведение не одним автоматным объектом, а системой взаимодействующих автоматных объектов. Все автоматные объекты в системе равноправны, и каждый из них взаимодействует с другими автоматными объектами и с внешней средой через интерфейсы четырех типов (см. рис. 2, *а*). Источником событий и объектом управления для данного автоматного объекта могут быть как внешние объекты, так и этот же или другой автомат системы. Таким образом, источники событий, объекты управления, управляющие автоматы — это все автоматные объекты, а источники событий и объекты управления — не более чем автоматные

объекты со специализированными интерфейсами (см. рис. 2, *б*). Унифицированная трактовка управляющих автоматов, объектов управления и источников событий как автоматных объектов является самым важным преимуществом предложенной модели по сравнению со всеми ранее предложенными моделями автоматного программирования [1–3].

Далее, в предлагаемой автоматной модели мы можем связывать автоматные объекты через соответствующие интерфейсы произвольным образом. Можно предоставляемый командный интерфейс одного автомата связать с требуемыми командными интерфейсами других автоматов, и первый автомат станет «слушаться нескольких командиров». Можно требуемый командный интерфейс одного автомата связать с предоставляемыми командными интерфейсами других автоматов, и первый автомат станет «командовать несколькими подчиненными». Можно связывать предоставляемые интерфейсы с требуемыми интерфейсами того же автомата, и автомат сам себе начнет посылать события — возможности автоматной модели ничем не ограничены!

Более того, предлагаемая автоматная модель естественным образом подразумевает асинхронность, параллельность и недетерминированность. Совсем нетрудно получить привычную последовательную схему: достаточно вытянуть автоматы в линейку, связывая предоставляемые интерфейсы предыдущего с требуемыми интерфейсами последующего. Но если нет ограничений последовательного выполнения, то в предлагаемой модели можно строить естественные параллельные схемы, весьма эффективные на современных платформах.

Метамодель языка автоматных объектов

Нотация предлагаемой модели поведения образует самостоятельный графический язык, который мы назвали ЧАО (CIAO — Cooperative Interaction of Automata Objects). Графический язык ЧАО предназначен для:

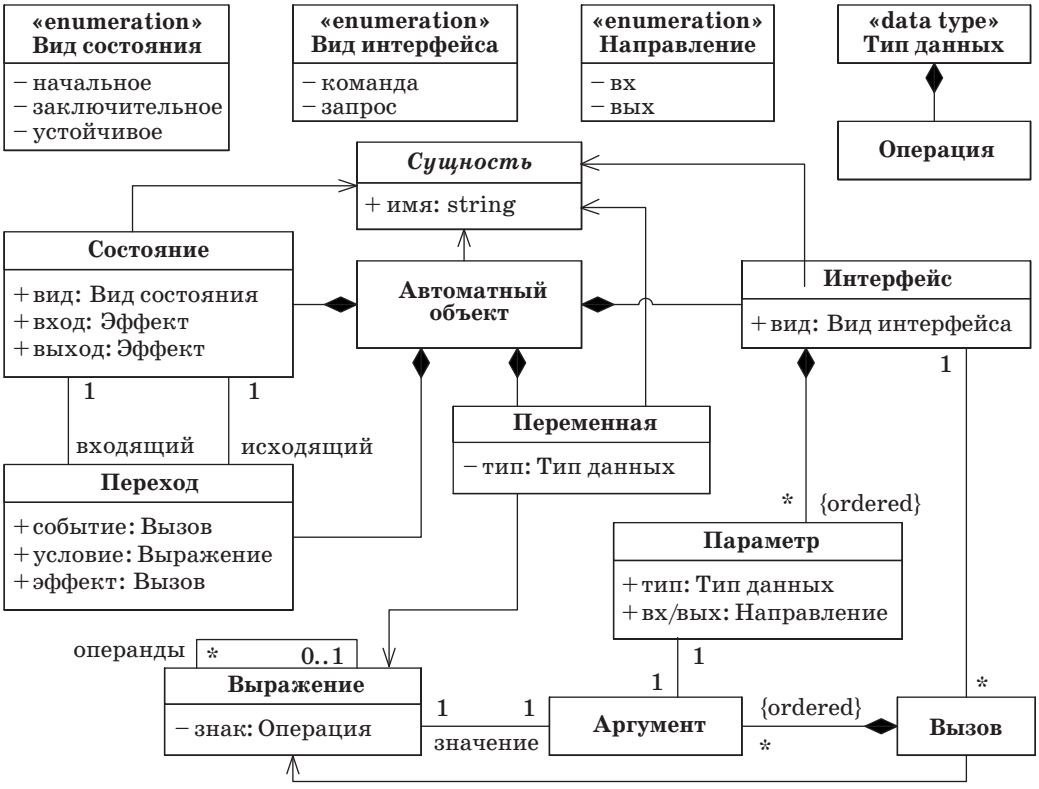
— наглядного описания и визуализации сложного поведения;

— публикации (параллельных) алгоритмов;

— построения имитационных моделей сложного поведения и доказательного исследования этих моделей;

— быстрого прототипирования программного обеспечения распределенных реагирующих систем.

Графический язык ЧАО является совместной комбинацией и расширением диаграммы автомата и диаграммы компонентов унифицированного языка моделирования UML [19]. Для определения языка, в особенности графического, мы предпочитаем использовать метамодель в форме диаграммы классов [21–23]. На рис. 3 представлена метамодель языка ЧАО.



■ ***Рис. 3.*** Метамодель языка описания поведения ЧАО

Язык имеет четыре основные именованные сущности: автоматный объект, состояние, интерфейс и переменную. Автоматный объект является классификатором в смысле языка UML и потому обозначается прямоугольником, состояния и интерфейсы трактуются и обозначаются в точности так, как это принято в языке UML [19]. Переменные обозначаются в текстовом виде, как это принято в языках программирования. Заметим, что переменная — это примитивный автоматный объект с командой присваивания значения (*set*) и с запросом получения присвоенного значения (*get*). Таким образом, использование переменных не нарушает «чистоту» автоматного программирования и является не более чем «синтаксическим сахаром», введенным для удобства.

Переходы между состояниями изображаются в виде стрелочек, нагруженных указанием события, сторожевого условия и эффекта перехода в текстовом виде. События обязаны быть предоставляемыми командами, эффекты обязаны быть требуемыми командами, сторожевые условия строятся из требуемых запросов. Синтаксис всех этих конструкций принят такой же, как в обычных языках программирования: выражения, в частности сторожевые условия, составлены из имен переменных и вызовов функций с помощью знаков операций, события и эффекты оформляются как операторы вызова функций без результата (процедур). Особо оговорим синтаксис предоставляемых запросов. По определению, запрос не меняет состояния, поэтому запрос — это петля, присущая всем состояниям. Мы изображаем ее единожды, как петлю объемлющего составного состояния, т. е. петлю самого автоматного объекта.

В целом нотация языка ЧАО нам представляется достаточно простой и наглядной, поэтому не стоит здесь останавливаться на деталях — они проясняются в последующих примерах.

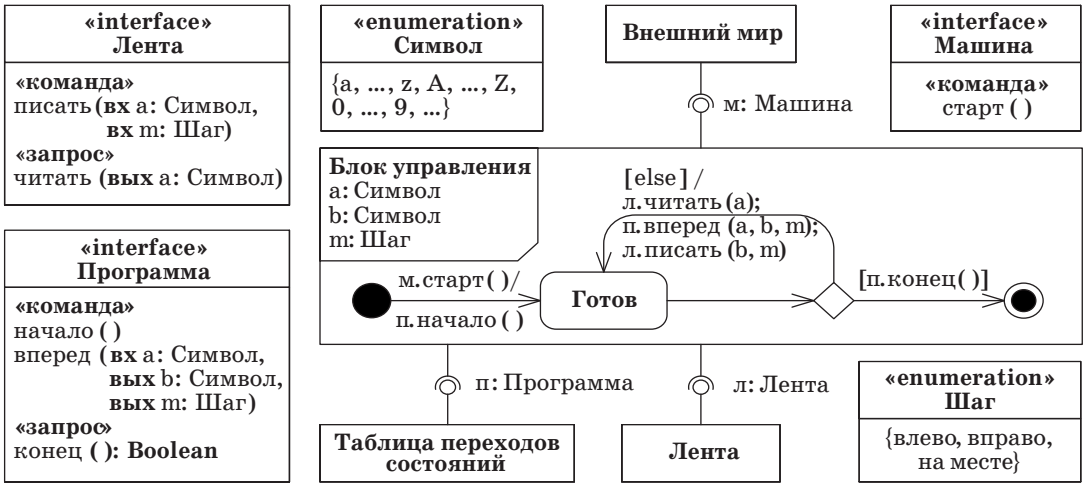
Алгоритмическая полнота модели поведения

Предлагаемая модель поведения алгоритмически полна по Тьюрингу [7]. Этот факт подтверждается эмулятором машины Тьюринга (рис. 4).

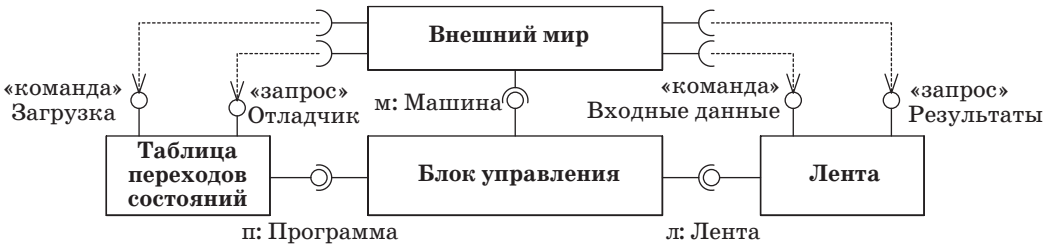
Система автоматных объектов, приведенная на рис. 4, — не самый лаконичный вариант реализации машины Тьюринга, но эта диаграмма прямо соответствует обычным словесным описаниям машины [7], а потому подходит для демонстрации алгоритмической полноты. Эмулятор машины Тьюринга состоит из трех автоматных объектов: блока управления, программной памяти, хранящей таблицу переходов состояний, и ленты. Кроме того, на диаграмме присутствует объект, поведение которого не определено, — это внешний мир, про существование которого часто забывают упомянуть, описывая машину Тьюринга. Автоматные объекты связаны интерфейсами **Машина**, **Программа** и **Лента**, операции этих интерфейсов полностью специфицированы. При этом автоматный объект **Блок управления** раскрыт, т. е. показан граф переходов состояний и локальные переменные, а прочие объекты не раскрыты.

Поясним используемые обозначения. Диаграмма автомата заключена в рамку по правилам UML [19]. В ярлычке рамки написано название автоматного объекта и перечислены локальные переменные. Предоставляемые и требуемые интерфейсы обозначены «шарнирами» в соответствии с нотацией UML 2, кроме того, указаны имена и типы интерфейсов. Имена интерфейсов и операций используются на переходах автомата для указания переключающих событий, сторожевых условий и выполняемых действий.

Опишем содержание рис. 4 на естественном языке. Блок управления предоставляет только од-



■ ***Рис. 4.*** Описание поведения машины Тьюринга



■ ***Рис. 5.*** Дополнительные интерфейсы машины Тьюринга

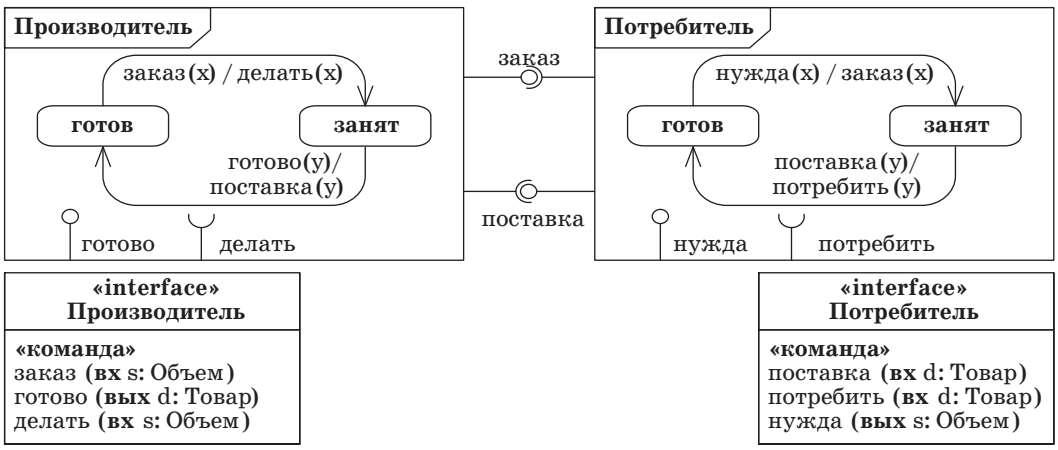
ну команду: **старт.** После старта память программы машины Тьюринга переводится в исходное состояние, машина переходит в состояние **Готов** и сразу (спонтанно, как это предусмотрено в UML), считывает символ **a** с ленты, движется вдоль ленты в направлении **m**, записывает на ленту символ **b** и т. д. Не имеет значения, что находится внутри «черных ящиков» **Внешний мир**, **Таблица переходов состояний** и **Лента**. Там могут быть автоматные объекты любого рода, электронные устройства или люди и т. д. Все работает отлично, если объявленные интерфейсы предоставляются и подписанные контракты соблюдаются.

Мы видим, что неформальное описание поведения машины Тьюринга схвачено довольно хорошо. Но представляет ли эта схема практический интерес? Может ли это быть полезным при анализе, сравнении и согласовании описаний поведения? Мы считаем, что да. Даже поверхностный анализ диаграммы показывает, что обычное описание поведения машины Тьюринга неполно и недостаточно для того, чтобы машину Тьюринга можно было бы реально использовать для программирования. Например, нам нужна возможность заполнить ленту исходными символами перед запуском машины, а также нужна возможность рассмотреть результаты после того, как машина Тьюринга достигла конечного состояния. Кроме того, мы должны иметь возможность загрузить таблицу переходов состояний перед запуском машины, и мы нуждаемся в возможности наблюдать за текущим состоянием, если собираемся отлаживать машину Тьюринга. Таким образом, чтобы описать требуемое поведение машины Тьюринга более точно, формально и детально, мы можем ввести дополнительные контракты и интерфейсы (рис. 5). Язык ЧАО это позволяет.

Командное взаимодействие автоматных объектов

Выразительную силу предлагаемой модели поведения мы демонстрируем на известном примере взаимодействующих процессов производителя и потребителя. Производитель последовательно производит порции некоторого товара, а потребитель потребляет их в той же последовательности. Оба процесса независимы и обладают «свободой воли», но они ограничены контрактом.

Мы начнем рассмотрение примера с простейшего контракта: производитель должен произвести товар, если он получил заказ от потребителя, а потребитель должен потребить произведенный товар, если он разместил заказ. Конечно, потребитель не может потребить товар, если он не был произведен, а производитель не должен производить новый товар, если предыдущий не был потреблен. Наше решение не претендует на научную новизну, но оно просто (если не сказать тривиально) и представляется нам очевидным (рис. 6).



■ ***Рис. 6.*** Взаимодействие производителя и потребителя

Не требуется никаких ухищрений для организации взаимодействия. Единственное предположение состоит в том, что обработка любого события является атомарной операцией.

Производитель и потребитель связаны через интерфейсы **заказ** и **поставка**. Интерфейс **заказ** является требуемой командой для потребителя и событием для производителя, а интерфейс **поставка** является требуемой командой для производителя и событием для потребителя. Получив событие **заказ** в состоянии **готов**, производитель дает команду **делать** и переходит в состояние **занят** до тех пор, пока не наступит событие **готово**, после чего отправляет произведенный товар потребителю и возвращается в состояние **готов**. Потребитель же, находясь в состоянии **готов**, в случае события **нужда** дает команду **заказ** и переходит в состояние **занят** до тех пор, пока не наступит событие **поставка**, после чего товар потребляется, и потребитель возвращается в состояние **готов**.

Необходимо обратить внимание на две особенности этой модели.

1. Интерфейсы **заказ** и **поставка** служат для взаимодействия производителя и потребителя, поэтому мы изображаем их снаружи прямоугольников, обрамляющих соответствующие автоматные объекты. Интерфейсы **делать** и **готово** производителя (аналогично интерфейсы **нужда** и **потребить** потребителя) являются интерфейсами для взаимодействия с внутренними автоматными объектами (может статься, что это элементарные действия базовой машины), поэтому мы изображаем их внутри прямоугольников автоматных объектов. Именно таким образом в языке ЧАО реализуется важная концепция вложенности автоматов [5].

2. Все команды снабжены параметрами, которые позволяют передавать дополнительную информацию. Вообще говоря, в рассматриваемом простейшем случае можно было бы обойтись без параметров. Однако использование параметров (и переменных) мы считаем важным и полезным свойством языка ЧАО: например, заказанный товар и произведенный товар — это разные вещи, и они обозначены у нас разными буквами.

Наиболее значительным преимуществом предлагаемого способа описания поведения является то обстоятельство, что мы можем формально доказать правильность описания. Действительно, если до начала работы оба автоматных объекта находятся в состоянии **готов**, то любой протокол выполнения является многократным повторением последовательности действий

$$<\textbf{нужда}(x) \rightarrow \textbf{заказ}(x) \rightarrow \textbf{делать}(x) \rightarrow \textbf{готово}(y) \rightarrow \textbf{поставка}(y) \rightarrow \textbf{потребить}(y)>.$$

Справедливость этого утверждения следует из однозначности маршрутов в графах переходов состояний. Доказанное формальное утверждение с лихвой перекрывает неформальную спецификацию контракта, с которой мы начали этот пример. Таким образом, в данном случае описание поведения не нуждается в отладке — оно правильно по построению. Более того, если действие **делать** фактически делает то, что требуется (т. е. y точно соответствует x, что требует отдельной проверки), то можно доказательно утверждать, что каждый раз потребляется в точности то, в чем была нужда. Это полезное свойство описания поведения, которое не было явно затребовано в неформальном описании контракта. В данном случае доказательство правильности получилось очень простым из-за однозначности путей в графах, в других случаях доказательство правильности может потребовать определенных усилий, но сама возможность строить доказательно правильные описания стоит того.

Обработка исключительных ситуаций

В уже рассмотренном простейшем случае предполагалось, что процессы производителя и потребителя работают штатно, без сбоев. Но в жизни бывают нештатные ситуации. Разумная обработка исключений — ключ к надежности. Разумно обработать можно только предусмотренные исключительные ситуации. Допустим, что в процессах производителя и потребителя возможно возникновение следующих исключительных ситуаций:

1) производитель не может принять заказ (например, перегружен другими заказами);

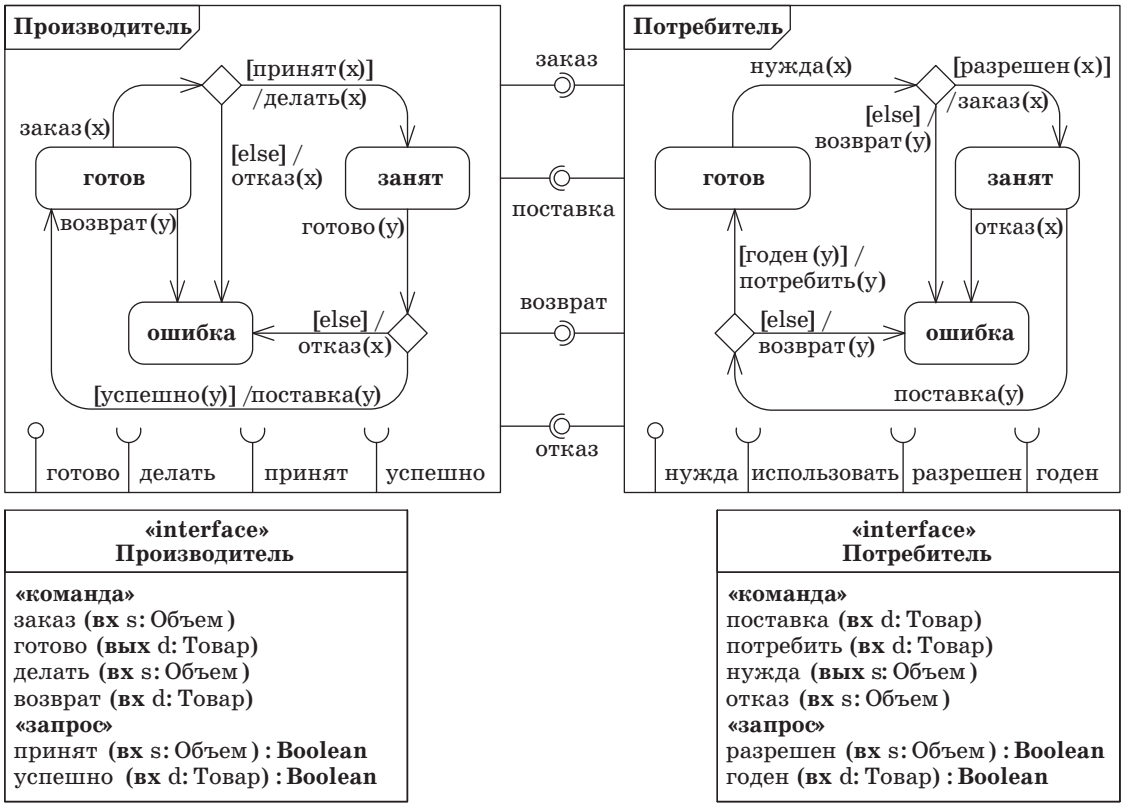
2) производитель не может выполнить заказ (сломалось оборудование);

3) потребитель не может разместить заказ (нет денег);

4) потребитель не может потребить поставленный товар (несоответствие качества).

Обработка этих исключений может быть сделана консервативным расширением ранее построенных диаграмм. На диаграммах появляются два состояния **ошибка** и дополнительные командные интерфейсы **возврат** и **отказ**. Наряду с командами мы используем запросы **принят** (проверка возможности принять заказ), **успешно** (результат производства товара), **разрешен** (проверка возможности разместить заказ), **годен** (проверка качества товара) и сегментированные переходы [19] со сторожевыми условиями (рис. 7).

В этом примере мы выбрали простейший вариант обработки исключений, когда при возникновении исключения работа просто останавливается, и главная задача обработчика — не допустить выполнения «лишних» действий, ненужных или даже вредных в исключительной ситуации. Например, если заказ x по каким-то причинам не может быть выполнен, то нежелательно отдавать команду **делать**(x) — это может привести к еще худшим последствиям.



■ ***Рис. 7.*** Обработка исключений при взаимодействии производителя и потребителя

В результате получилось достаточно надежное решение, а именно, можно показать, что имеет место расширенное свойство правильности. Если до начала работы оба автоматных объекта находятся в состоянии **готов**, то либо исключений не случается, и тогда любой протокол выполнения является многократным повторением последовательности действий

$$<\textbf{нужда}(x) \rightarrow \textbf{заказ}(x) \rightarrow \textbf{делать}(x) \rightarrow \textbf{готово}(y) \rightarrow \textbf{поставка}(y) \rightarrow \textbf{потребить}(y)>,$$

либо случается ровно одно исключение, протокол завершается событием **отказ** или **возврат**, и оба автоматных объекта переходят в заключительное состояние **ошибка**. Доказательство проводится аналогично предыдущему, но оказывается достаточно пространным, поскольку придется провести разбор случаев.

По нашему мнению, приведенный пример показывает, что предлагаемая модель поведения позволяет наглядно и доказательно описывать достаточно сложное поведение.

Учет ограничений времени

В предыдущем примере события исключительных ситуаций возникают по внутренним причинам автоматных объектов. Однако существуют события, возникающие по внешним причинам. Наиболее часто встречающийся случай — прерывание по времени. Ясно, что реальные производители и реальные потребители не могут находиться в состояниях **готов** и **занят** бесконечно долго.

Пусть заданы следующие четыре интервала времени:

1) t_1 — максимальное время, которое производитель может находиться в состоянии **готов**, не получая новых заказов;

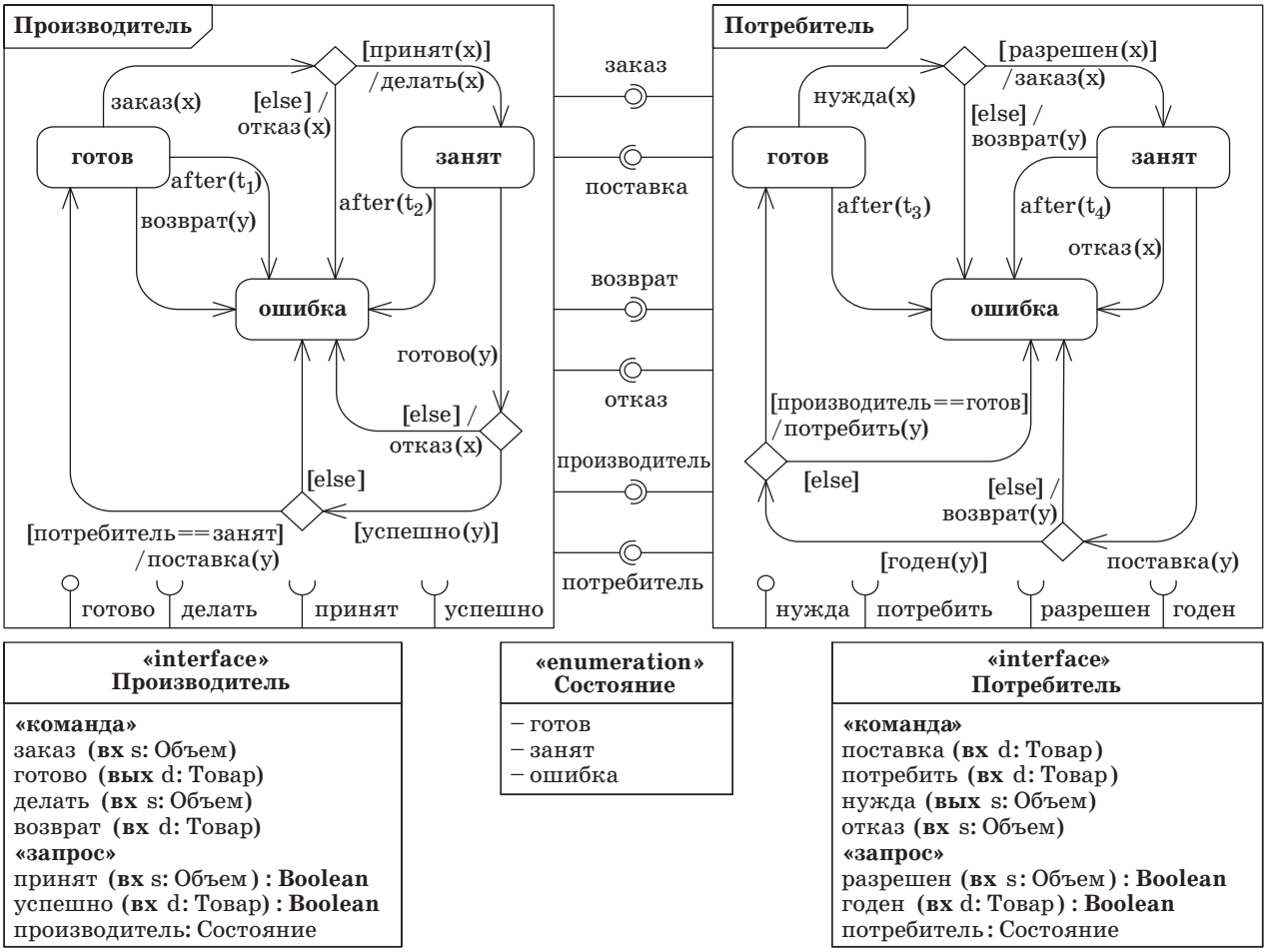
2) t_2 — максимальное время, которое производитель может находиться в состоянии **занят**, не производя новых товаров;

3) t_3 — максимальное время, которое потребитель может находиться в состоянии **готов**, не испытывая нужды;

4) t_4 — максимальное время, которое потребитель может находиться в состоянии **занят**, не получая заказанного товара.

Мы считаем, что если любое из этих ограничений нарушено, то возникает исключительная ситуация и автоматный объект переходит в состояние **ошибка**. Решение для этого случая показано на рис. 8.

На диаграммах рис. 8 следует обратить внимание на следующее важное обстоятельство, связанное со временем. Мы используем событие таймера UML (ключевое слово **after** [19]), которое возникает, когда истечет заданный интервал времени. Это очень удобно и естественно при



«interface» Производитель

«команда»
заказ **(вх** s: Объем**)**
готово **(вых** d: Товар**)**
делать **(вх** s: Объем**)**
возврат **(вх** d: Товар**)**
«запрос»
принят **(вх** s: Объем**) : Boolean**
успешно **(вх** d: Товар**) : Boolean**
производитель: Состояние

«enumeration» Состояние

– готов
– занят
– ошибка

«interface» Потребитель

«команда»
поставка **(вх** d: Товар**)**
потребить **(вх** d: Товар**)**
нужда **(вых** s: Объем**)**
отказ **(вх** s: Объем**)**
«запрос»
разрешен **(вх** s: Объем**) : Boolean**
годен **(вх** d: Товар**) : Boolean**
потребитель: Состояние

■ ***Рис. 8.*** Взаимодействие с учетом ограничений по времени

описании локальных изменений состояний по времени. Однако из этого следует, что каждый из автоматных объектов не может знать, что в другом объекте произошел переход по времени, поскольку автоматные объекты асинхронны, и время течет в них независимо. Необходимо как-то известить другой автоматный объект. Это можно сделать разными способами. Можно было бы завести специальные события, по одному на каждый возможный переход по времени, и передавать их через специальные интерфейсы, аналогично тому, как это сделано ранее (см. рис. 7). Однако такое решение выглядит тяжеловесным и неубедительным. Можно было бы использовать глобальные часы (ключевое слово **at** [19]) и событие изменения (ключевое слово **when** [19]), чтобы проверять наступление событий в одном автоматном объекте из другого автоматного объекта. Однако такое решение прямо противоречит принципам инкапсуляции. Мы предлагаем и рекомендуем другое решение, основанное на следующем полезном приеме: каждый автоматный объект предоставляет запрос без параметров, доставляющий текущее состояние, а другие автоматные объекты могут проверять это состояние. В парадигме автоматного программирования это называется *наблюдаемостью* [1] и обоснованно считается важным преимуществом. В данном случае, прежде чем отдать команду, мы проверяем, готов ли объект выполнить команду, т. е. находится ли он в подходящем состоянии.

Заметим, что новые диаграммы на рис. 8 опять получены консервативным расширением предыдущих диаграмм на рис. 7, т. е. реализуют принцип повторного использования и позволяют провести формальное доказательство правильности методом разбора случаев, хотя разобрать придется достаточно много случаев. В доказательном программировании имеет место следующая принципиальная проблема: как убедиться, что формальное доказательство безошибочности некоторой программы само не содержит ошибок? Общим ответом является построение метода автоматического доказательства правильности. Для систем взаимодействующих автоматных объектов метод автоматического доказательства правильности пока не построен, но мы надеемся, что он возможен и будет предложен в ближайшем будущем.

Параллельное программирование

Мы рассматриваем вопросы параллельного программирования в предлагаемой модели поведения на примере задачи Э. Дейкстры об обедающих философах [24]. Пять философов сидят за круглым столом, перед каждым стоит тарелка спагетти. Вилки лежат на столе между каждой парой тарелок (всего пять вилок). Каждый философ может взять вилку (если она доступна) или положить (если он уже держит ее). Взятие каждой вилки и возвращение ее на стол являются раздельными действиями. Каждый философ может либо есть, либо размышлять. Философ может есть только тогда, когда держит две вилки одновременно. Прием пищи не ограничен (бесконечный запас). Если требуемая вилка занята соседом, голодный философ вынужден ждать — он не может вернуться к размышлениям, не поев. После окончания еды философ кладет вилки на стол для того, чтобы ими могли воспользоваться другие философы. Задача состоит в разработке модели поведения, при котором:

— ни один из философов не голодает (будет вечно чередовать приемы пищи и размышления при бесконечном выполнении программы);

— ресурсы равномерно распределяются между философами (при одинаковом поведении философы едят примерно одинаковое количество времени, никто не получает преимущества);

— философы наделены свободой воли и молчаливы (не общаются непосредственно и нет внешнего агента, который бы командовал ими).

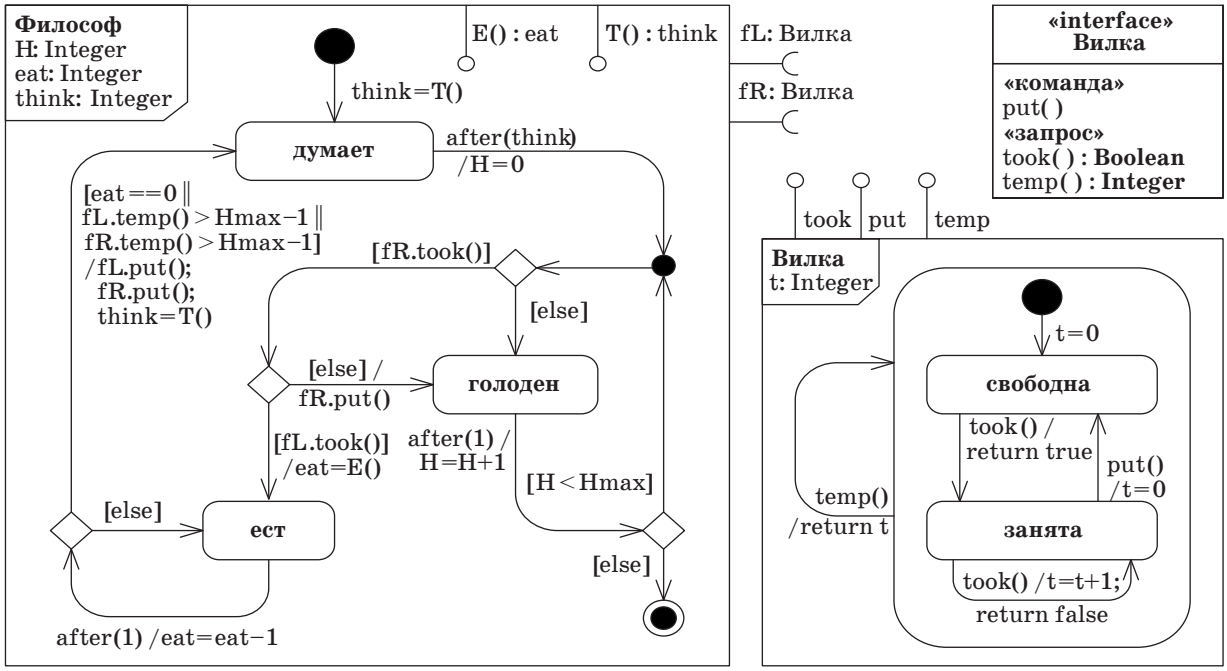
Если бы вилок было с избытком, то жизненный цикл каждого философа был бы очевиден: размышляет → берет правую вилку → берет левую вилку → ест → кладет левую вилку → кладет правую вилку → размышляет. Однако вилок только пять, и возможна взаимная блокировка: философы проголодаются примерно одновременно, возьмут по одной вилке (правой), и умрут от голода в ожидании, когда освободится вторая вилка. Возможны и другие коллизии, многие из которых исследованы в книге [24].

Заметим, что решение задачи об обедающих философах не всегда возможно — многое зависит от соотношения времени размышления и приема пищи. Например, если все философы дольше едят, нежели размышляют, то производительности системы заведомо не хватит, поскольку в лучшем случае одновременно принимать пищу могут два философа, а всего их пять. Случай, когда длительность каждого состояния для каждого философа фиксирована, в нашем контексте не очень интересен, поскольку решается предварительным расчетом и статическим составлением расписания. С точки зрения параллельного программирования более интересен случай, когда потребность в ресурсах (вилках) возникает асинхронно в неизвестные заранее моменты времени, как это обычно бывает в жизни.

Положим, что каждый философ пребывает в одном из трех ортогональных состояний: размышляет, ест или голодает, т. е. закончил размышлять, но не начал есть, поскольку нет свободных вилок. Как обычно в модельных задачах, время считаем дискретным и измеряем натуральными числами. Допустим, что продолжительность размышления T и продолжительность приема пищи E — псевдослучайные величины, распределение которых задано извне (или неизвестно), а продолжительность голодания H определяется нашим алгоритмом. Естественно считать, что величины T, E и H ограничены, причем если величина H достигает верхней границы $H_{\max}$, то в работе системы возникает исключительная ситуация — несчастный философ умирает от голода.

Голодная смерть представляется нам крайне нежелательным исходом, поэтому мы считаем, что философы из гуманных соображений могут прекратить прием пищи и освободить вилку, когда сосед умирает от голода. Хотя философы между собой не общаются, они могут взаимодействовать через вилки, которые тоже являются автоматными объектами. У вилки есть локальная переменная t — счетчик отказов, который увеличивается на единицу каждый раз, когда была попытка взять вилку в тот момент, когда она занята. Нетрудно видеть, что t — это время голодания соседа, которому вилка нужна, но недоступна. Если время голодания соседа (слева или справа) приблизилось к пределу, то каждый философ прекращает есть спагетти и освобождает вилки. Наше решение приведено на рис. 9.

Из трех требований к решению приоритет отдан первому: ценой некоторых неудобств, связанных с недоеданием и прерыванием трапез, философы не будут умирать с голоду, если это возможно. Второе требование требует знания свойств функций T и E — имеют ли они заданное распределение вероятности или же они зависят от предыстории, и т. д. В этом случае средствами теории вероятностей возможно математически исследовать описание поведения, например, определить математическое ожидание времени голодания и т. д. Третье требование выполнено, философы прямо не общаются, и нет координирующего центра. Это достигнуто за счет того, что вилки наделены не только памятью (как в известном решении с присвоением ресурсам частичного порядка [24]), но и *поведением*. По нашему мнению, пример убедительно демонстрирует пригодность языка ЧАО для публикации, обсуждения и исследования параллельных и распределенных систем.



■ ***Рис. 9.*** Описание алгоритма поведения обедающих философов

Заключение

Предложенная модель уже несколько раз была опробована на практике и показала хорошую применимость. В частности, в работах [21–23] описан автоматный метод определения языков предметной области с помощью систем взаимодействующих автоматных объектов. В автоматном методе непосредственно используется предлагаемая модель, хотя язык ЧАО еще имеет латентную форму. В статьях [5, 6] выполнено моделирование специального программного обеспечения для систем реального времени научного назначения (астрономия) с непосредственным использованием данного языка.

Предложенная модель применяется в ряде текущих проектов Специальной астрофизической обсерватории РАН, в Санкт-Петербургском политехническом университете Петра Великого, и мы надеемся на положительные результаты.

Рамки одной статьи не позволили охватить важные теоретические вопросы данной модели:

— кооперация взаимодействующих автоматных объектов (схема связей в смысле книги [2]) позволяет устанавливать взаимодействие между автоматными объектами как статически, так и динамически, во время выполнения, что особенно важно для реагирующих систем, поведение которых должно меняться в зависимости от окружающей среды;

— автоматные объекты допускают неограниченную вложенность при сохранении объектно-ориентированной инкапсуляции. Использованная концепция локальных переменных является примером, но не исчерпывает возможности в этом направлении;

— описание поведения взаимодействующими автоматными объектами более благоприятно для применения математических методов по сравнению с императивным программированием: явное выделение состояний индуцирует явный протокол выполнения, свойства которого можно математически доказывать или количественно оценивать.

Затронутые, но не раскрытые аспекты модели поведения взаимодействующих автоматных объектов, а также практические аспекты реализации системы программирования на основе языка ЧАО мы опишем в последующих статьях в ближайшем будущем.

Литература

1. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. — СПб.: Наука, 1998. — 628 с.
2. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. — СПб.: Питер, 2011. — 176 с.
3. **Шалыто А. А.** Парадигма автоматного программирования // Научно-технический вестник СПбГУ ИТМО. 2008. Вып. 53. С. 3–24.
4. **Atiskov A. Y.** et al. Ontology-Based Analysis of Cryptography Standards and Possibilities of Their Harmonization / A. Y. Atiskov, F. A. Novikov, L. N. Fedorchenko, V. I. Vorobiev, N. A. Moldovyan // Theory and

Practice of Cryptography Solutions for Secure Information Systems. — Hershey: IGI Global, 2013. — P. 1–33. doi:10.4018/978-1-4666-4030-6.ch001
5. **Afanasieva I. V.** Data Acquisition and Control System for High-Performance Large-Area CCD Systems // Astrophysical Bulletin. 2015. Vol. 70. N 2. P. 232–237. doi:10.1134/S1990341315020108
6. **Афанасьева И. В., Новиков Ф. А.** Архитектура программного обеспечения систем оптической регистрации // Информационно-управляющие системы. 2016. № 3. С. 51–63. doi:10.15217/issn1684-8853.2016.3.51
7. **Hopcroft J. E., Motwani R., Ullman J. D.** Introduction to Automata Theory, Languages, and Computation. — Addison-Wesley, 2001. — 521 p.
8. **Bock C., Odell J.** Ontological Behavior Modeling // Journal of Object Technology. 2011. N 10. P. 1–36. doi:10.5381/jot.2011.10.1.a3
9. **Cormen T. H.,** et al. Introduction to Algorithms/ T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein. Third Ed. — Cambridge: The MIT Press, 2009. — 1312 p.
10. **Knuth D. E.** The Art of Computer Programming. — Addison-Wesley Professional, 2011. — 3168 p.
11. **Börger E., Stärk R.** Abstract State Machines. A Method for High-Level System Design and Analysis. — Berlin: Springer, 2003. — 438 p. doi:10.1007/978-3-642-18216-7
12. **Новиков Ф. А.** Дискретная математика. — СПб.: Питер, 2013. — 432 с.
13. **Bock C.** Three Kinds of Behavior Models // Journal of Object-Oriented Programming. 1999. N 12 (4). P. 36–39.
14. **Booch G., Rumbaugh J., Jacobson I.** The Unified Modeling Language Reference Manual. — Addison-Wesley, 2010. — 721 p.
15. **Новиков Ф. А.** Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6. С. 9–22.
16. **Harel D.** Statecharts: a Visual Formalism for Complex Systems // Science of Computer Programming. 1987. Vol. 8. P. 231–274. doi:10.1016/0167-6423(87)90035-9
17. **Bock C.** UML 2 Activity and Action Models // Journal of Object Technology. 2003. Vol. 2. N 4. P. 43–53. doi:10.5381/jot.2003.2.4.c3; Part 2: Actions // Journal of Object Technology. 2003. Vol. 2. N 5. P. 41–56. doi:10.5381/jot.2003.2.5.c4; Part 3: Control Nodes // Journal of Object Technology. 2003. Vol. 2. N 6. P. 7–23. doi:10.5381/jot.2003.2.6.c1; Part 4: Object Nodes // Journal of Object Technology. 2004. Vol. 3. N 1. P. 27–41. doi:10.5381/jot.2004.3.1.c3; Part 5: Partitions // Journal of Object Technology. 2004. Vol. 3. N 7. P. 37–56. doi:10.5381/jot.2004.3.7.c4; Part 6: Structured Activities // Journal of Object Technology. 2005. Vol. 4. N 4. P. 43–66. doi:10.5381/jot. 2005.4.4.c4
18. **Карпов Ю. Г.** Теория автоматов. — СПб.: Питер, 2002. — 224 с.
19. **Новиков Ф. А., Иванов Д. Ю.** Моделирование на UML. Теория, практика, видеокурс. — СПб.: Профессиональная литература, Наука и Техника, 2010. — 640 с.
20. **Meyer B.** Object-Oriented Software Construction. — Prentice-Hall, 2000. — 1406 p.
21. **Новиков Ф. А., Тихонова У. Н.** Автоматный метод определения проблемно-ориентированных языков. Ч. 1 // Информационно-управляющие системы. 2009. № 6. С. 34–40.
22. **Новиков Ф. А., Тихонова У. Н.** Автоматный метод определения проблемно-ориентированных языков. Ч. 2 // Информационно-управляющие системы. 2010. № 2. С. 31–37.
23. **Новиков Ф. А., Тихонова У. Н.** Автоматный метод определения проблемно-ориентированных языков. Ч. 3 // Информационно-управляющие системы. 2010. № 3. С. 29–37.
24. **Hoare C. A. R.** Communicating Sequential Processes. — Prentice-Hall, 1985. — 256 p. doi:10.1145/357980.358021

UDC 004.434
doi:10.15217/issn1684-8853.2016.6.50

Cooperative Interaction of Automata Objects

Novikov F. A.[a], Dr. Sc., Tech., Professor, fedornovikov51@gmail.com
Afanasieva I. V.[b,c], Senior Engineer, Post-Graduate Student, riv@sao.ru
[a]Peter the Great St. Petersburg Polytechnic University, 29, Politekhnicheskaia St., 195251, Saint-Petersburg, Russian Federation
[b]Special Astrophysical Observatory, Russian Academy of Sciences, Nizhnii Arkhyz, 369167, Russian Federation
[c]Saint-Petersburg National Research University of Information Technologies, Mechanics and Optics, 49, Kronverkskii St., 197101, Saint-Petersburg, Russian Federation

Purpose: We propose a behavior description model which would help to achieve higher parameters of reliability and performance as compared to the conventional methods of developing reactive and distributed systems. Our secondary purpose is creating a user-friendly graphical language to describe parallel algorithms and distributed reactive systems. **Methods:** To describe the behavior, we use state machine diagrams (state transition graphs) of Unified Modeling Language (UML) enhanced with special stereotypes, ports and interfaces. For the description of the proposed graphical language, we use UML class diagrams as a metamodel. **Results:** A behavior model has been developed which uses state transition graphs and relates to the paradigm of automata-based programming. The features and advantages of the proposed model in the class of asynchronous distributed reactive systems are discussed in full details: its algorithm

completeness is proved, the graphical language is defined, examples are given, and the directions for further development are specified. **Practical relevance:** The proposed model was successfully applied to developing specialized software for precision scientific equipment control in observational astronomy. Besides, it was used as a high-level behavior description tool in an automata-based method of determining domain-specific languages.

Keywords — Behavior Model, Automata-Based Programming, State Transition Graph, Unified Modeling Language, Statechart, Class Diagram, Parallel Behavior, Reactive System Architecture.

References

1. Shalyto A. A. *SWITCH-technologia. Algoritmizatsiia i programmirovanie zadach logicheskogo upravleniia* [Switch Technology. Algorithmization and Programming of Logical Control Problems]. Saint-Petersburg, Nauka Publ., 1998. 628 p. (In Russian).
2. Polikarpova N. I., Shalyto A. A. *Avtomatnoe programmirovanie* [Automata-Based Programming]. Saint-Petersburg, Piter Publ., 2011. 176 p. (In Russian).
3. Shalyto A. A. The Paradigm of Automata-Based Programming]. *Nauchno-tekhnicheskii vestnik SPbGU ITMO* [Scientific and Technical Journal of Information Technologies, Mechanics and Optics], 2008, vol. 53, pp. 3–24 (In Russian).
4. Atiskov A. Y., Novikov F. A., Fedorchenko L. N., Vorobiev V. I., Moldovyan N. A. Ontology-Based Analysis of Cryptography Standards and Possibilities of their Harmonization. In: *Theory and Practice of Cryptography Solutions for Secure Information Systems*. Hershey, IGI Global, 2013, pp. 1–33. doi:10.4018/ 978-1-4666-4030-6.ch001
5. Afanasieva I. V. Data Acquisition and Control System for High-Performance Large-Area CCD Systems. *Astrophysical Bulletin*, 2015, vol. 70, no. 2, pp. 232–237. doi:10.1134/ S1990341315020108
6. Afanasieva I. V., Novikov F. A. Software Architecture for Optical Detector Systems. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2016, no. 3, pp. 51–63 (In Russian). doi:10.15217/issn1684-8853.2016.3.51
7. Hopcroft J. E., Motwani R., Ullman J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001. 521 p.
8. Bock C., Odell J. Ontological Behavior Modeling. *Journal of Object Technology*, 2011, no. 10, pp. 1–36. doi:10.5381/ jot.2011.10.1.a3
9. Cormen T. H., Leiserson Ch. E., Rivest R. L., Stein C. *Introduction to Algorithms* (Third Edition). Cambridge, The MIT Press, 2009. 1312 p.
10. Knuth D. E. *The Art of Computer Programming*. Addison-Wesley Professional, 2011. 3168 p.
11. Börger E., Stärk R. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Berlin, Springer, 2003. 438 p. doi:10.1007/978-3-642-18216-7
12. Novikov F. A. *Diskretnaia matematika* [Discrete Mathematics]. Saint-Petersburg, Piter Publ., 2013. 432 p. (In Russian).
13. Bock C. Three Kinds of Behavior Models. *Journal of Object-Oriented Programming*, 1999, no. 12 (4), pp. 36–39.
14. Booch G., Rumbaugh. J., Jacobson I. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2010. 721 p.
15. Novikov F. A. Visual Software Gesign. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2005, no. 6, pp. 9–22 (In Russian).
16. Harel D. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987, vol. 8, pp. 231–274. doi:10.1016/0167-6423(87)90035-9
17. Bock C. UML 2 Activity and Action Models. *Journal of Object Technology*, 2003, vol. 2, no. 4, pp. 43–53. doi:10.5381/ jot.2003.2.4.c3; Part 2: Actions. *Journal of Object Technology*, 2003, vol. 2, no. 5, pp. 41–56. doi:10.5381/ jot.2003.2.5.c4; Part 3: Control Nodes. *Journal of Object Technology*, 2003, vol. 2, no. 6, pp. 7–23. doi:10.5381/ jot.2003.2.6.c1; Part 4: Object Nodes. *Journal of Object Technology*, 2004, vol. 3, no. 1, pp. 27–41. doi:10.5381/ jot.2004.3.1.c3; Part 5: Partitions. *Journal of Object Technology*, 2004, vol. 3, no. 7, pp. 37–56. doi:10.5381/ jot.2004.3.7.c4; Part 6: Structured Activities. *Journal of Object Technology*, 2005, vol. 4, no. 4, pp. 43–66. doi:10.5381/jot.2005.4.4.c4
18. Karpov Iu. G. *Teoriia avtomatov* [Automata Theory]. Saint-Petersburg, Piter Publ., 2002. 224 p. (In Russian).
19. Novikov F. A., Ivanov D. Iu. *Modelirovanie na UML. Teoriia, praktika, videokurs* [Modeling in UML. Theory, Practice, Video Course]. Saint-Petersburg, Professional'naia literatura, Nauka i Tekhnika Publ., 2010. 640 p. (In Russian).
20. Meyer B. *Object-Oriented Software Construction*. Prentice-Hall, 2000. 1406 p.
21. Novikov F. A., Tikhonova U. N. An Automata Based Method for Domain Specific Languages Definition. Part 1. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2009, no. 6, pp. 34–40 (In Russian).
22. Novikov F. A., Tikhonova U. N. An Automata Based Method for Domain Specific Languages Definition. Part 2. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2010, no. 2, pp. 31–37 (In Russian).
23. Novikov F. A., Tikhonova U. N. An Automata Based Method for Domain Specific Languages Definition. Part 3. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2010, no. 3, pp. 29–37 (In Russian).
24. Hoare C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985. 256 p. doi:10.1145/357980.358021