ISBN 978-5-91601-086-2

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ НАУКИ
ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР им. А.А. ДОРОДНИЦЫНА
РОССИЙСКОЙ АКАДЕМИИ НАУК

Ю.И. БРОДСКИЙ

МОДЕЛЬНЫЙ СИНТЕЗ И МОДЕЛЬНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР ИМ. А.А. ДОРОДНИЦЫНА
РОССИЙСКОЙ АКАДЕМИИ НАУК
МОСКВА 2013

УДК 519.876

Ответственный редактор
член-корр. РАН Ю.Н. Павловский

Предлагается новый подход к проектированию и компьютерной реализации имитационных моделей сложных многокомпонентных систем, отличающийся от объектного подхода. Он назван в работе модельным синтезом и модельно-ориентированным программированием. Центральным понятием этого подхода и в то же время элементарным кирпичиком для построения любых более сложных конструкций является понятие модели-компоненты.

Организация имитационных вычислений модели-компоненты оказывается инвариантной относительно операции объединения моделей-компонент в модель-комплекс. Это позволяет строить фрактальные модели любой сложности и реализовывать вычислительный процесс даже очень сложных моделей единообразно. Предлагаемый подход позволяет исключить императивное программирование и производит программный код высокой степени параллельности.

Ключевые слова: имитационное моделирование, сложные многокомпонентные системы, модельный синтез, модельно-ориентированное программирование, параллельные и распределенные вычисления.

Работа выполнена при финансовой поддержке РФФИ, проект 13-01-00499-а, и РГНФ, проект 12-06-00932-а.

Рецензенты: И.Г. Поспелов,
Ю.И. Димитриенко

Научное издание

© Федеральное государственное бюджетное учреждение
науки Вычислительный центр им. А.А. Дородницына
Российской академии наук, 2013

Оглавление

Предисловие 5
Введение 7
1. Роды структур и элементы геометрической теории декомпозиции 17
1.1. Определение рода структуры 17
1.2. Примеры родов структур 20
1.3. Изоморфизмы, естественные канонические морфизмы 23
1.4. P- и F- редукции и декомпозиции 24
1.5. Редукции высших уровней 27
2. Математические модели. Внутренние и внешние характеристики. Гипотеза о замкнутости 28
2.1. Математические модели 28
2.2. Внутренние и внешние характеристики 30
2.3. Гипотеза о замкнутости 32
2.4. Гипотеза о замкнутости применительно к моделям сложных многокомпонентных систем 33
2.5. Особенности моделирования сложных многокомпонентных систем 37
2.6. Основные выводы из гипотезы о замкнутости 42
3. Модельный синтез – концепция описания и реализации имитационных моделей сложных многокомпонентных систем 44
3.1. Неформальное описание моделей-компонент и моделей-комплексов 44
3.2. Семейство родов структур «модель-компонента» 51
3.3. Синтез модели-комплекса из моделей-компонент 65

4. Модельно-ориентированное программирование.. 75
4.1. Декларативное и императивное программирование 75
4.2. Декомпозиция и инкапсуляция 81
4.3. Поведение математических моделей сложных систем 83
4.4. Модельно-ориентированная парадигма 88
4.5. Модельно-ориентированный декларативный язык описания компонент и комплексов (ЯОКК) 92
4.5.1. Пример использования ЯОКК – пешеходы и муха 94
4.5.2. Общий синтаксис ЯОКК 97
4.5.3. Описатель типа данных 100
4.5.4. Описатель комплекса 102
4.5.5. Описатель метода 104
4.5.6. Описатель компоненты 107
4.5.7. Компиляция описателей ЯОКК 115
5. Модельный синтез и объектный анализ 121
5.1. Элементы модельного анализа и проектирования имитационных моделей сложных систем 122
5.2. Модельный синтез vs объектный анализ 124
Заключение 135
Литература 140

Памяти Владимира Юрьевича Лебедева.

Предисловие

Работа посвящена проблеме описания и моделирования сложных систем, таких, про которые хорошо известно, из каких компонент они состоят, что эти компоненты умеют делать, по каким правилам взаимодействуют между собой. Проблемой моделирования, притом весьма непростой, является воспроизведение поведения и оценка возможностей сложной системы в целом.

Предлагается новый подход к проектированию и компьютерной реализации имитационных моделей сложных многокомпонентных систем, отличающийся от господствующего в последние десятилетия объектного подхода. Он назван в работе модельным синтезом, а его программная реализация – модельно-ориентированным программированием. Под модельным синтезом, в первую очередь, понимается подход к решению сформулированной выше проблемы построения модели сложной системы на основе моделирования составляющих ее компонент. Кроме того, модельный синтез – это еще и альтернатива объектному анализу – основной парадигме современного проектирования и программирования больших программных комплексов, если речь идет о создании программных комплексов, имеющих выраженное «атомистическое» строение.

Центральным понятием предлагаемого подхода и в то же время элементарным кирпичиком для построения любых более сложных конструкций является понятие модели-компоненты. Модель-компонента наделена более сложной структурой, чем, например, объект объектного анализа. Эта структура обеспечивает модели-компоненте поведение – способность стандартным обра-

зом отвечать на стандартные запросы ее внутренней и внешней среды.

Организация имитационных вычислений модели-компоненты оказывается инвариантной относительно операции объединения моделей-компонент в модель-комплекс, поскольку модель-комплекс сохраняет структуру модели-компоненты. Это позволяет, во-первых, строить фрактальные модели любой сложности и, во-вторых, реализовывать вычислительный процесс даже очень сложных моделей единообразно – единой программой. Кроме того, предлагаемый подход к компьютерной реализации многокомпонентных имитационных моделей полностью исключает императивное программирование и позволяет производить программный код высокой степени параллельности.

Данная работа является дальнейшим развитием работы [8]. Ее можно считать взглядом на проблематику имитационного моделирования сложных многокомпонентных систем с позиций аппарата родов структур Н. Бурбаки [13] и геометрической теории декомпозиции [23].

Автор благодарен Юрию Николаевичу Павловскому за многолетние, но ненавязчивые попытки (без малейшей надежды на успех) – приобщения его к аппарату родов структур Н. Бурбаки и геометрической теории декомпозиции.

Введение

> «По ГОСТ 23501.101-87, составными структурными частями САПР являются подсистемы, обладающие всеми свойствами систем и создаваемые как самостоятельные системы. Каждая подсистема – это выделенная по некоторым признакам часть САПР, обеспечивающая выполнение некоторых функционально-законченных последовательностей проектных задач с получением соответствующих проектных решений и проектных документов».
> *Википедия, статья САПР*

Во введении на неформальном, «гуманитарном» уровне дается обзор проблематики, основных идей, методов и результатов данной работы.

Предметом данной работы будут проблемы имитационного моделирования сложных систем. При этом, таких сложных систем, которые в предметной области имеют явно выраженную составную структуру, причем их компоненты сами могут быть сложными системами. Тем не менее, продвигаясь вниз в анализе устройства таких компонент, однажды мы доходим до «атомов», про которые известно все: как они устроены, что умеют делать, по каким правилам взаимодействуют между собой.

Проблемами имитационного моделирования, притом весьма непростыми, являются:

1. Зафиксировать это наше знание в виде полного и достаточно формального описания устройства сложной системы.
2. На основе такого описания попытаться воспроизвести поведение сложной системы (построить ее имитационную модель), например, с целью изучения и оценки возможностей такой системы в целом.

В плане неформального понимания того, что такое сложная многокомпонентная система, автору очень близки следующие три высказывания видного специалиста в области сложных систем и их имитационного моделирования, член.-корр. АН СССР Н.П. Бусленко, почерпнутые из работ [14, 15]:

«Сложная система – составной объект, части которого можно рассматривать как системы, закономерно объединённые в единое целое в соответствии с определенными принципами или связанные между собой заданными отношениями».

«В каждый момент времени элемент сложной системы находится в одном из возможных состояний; из одного состояния в другое он переходит под действием внешних и внутренних факторов».

«Для построения синтеза поведения сложной системы необходимо дать ее компонентам возможность в полной мере проявить себя».

В некотором смысле все, что предложено в данной работе, является одной из возможных реализаций приведенных выше высказываний.

Со времен античности и до наших дней существует два основных подхода к моделированию сложных систем:

1. Один из них, называвшийся в разное время и в различных ситуациях «феноменологическим», «волновым», «системно-динамическим» состоит в том, что у изучаемой системы выделяются поддающиеся измерению характеристики, описывающие эту систему в целом. Постулируется достаточность выделенных характеристик для описания системы (гипотеза о замкнутости) и ищутся связи между этими характеристиками. В рамках данного

подхода работали Гераклит и милетские философы, Гюйгенс в споре с Ньютоном о природе света, создатели классической термодинамики.

2. Второй подход, называвшийся «агентным», «объектным», «атомистическим», «корпускулярным», предполагает выводить свойства сложной системы из свойств и способов взаимодействия «атомов» – неких простейших объектов, эту систему составляющих. Этот подход развивали античные атомисты Левкипп и Демокрит, в новое время – Ньютон, в упоминавшемся выше споре с Гюйгенсом о природе света, создатели статистической физики.

В настоящее время общепринятой точкой зрения по данному вопросу является признание равноценности и взаимной дополняемости «корпускулярного» и «волнового» описания реальности. Некоторые явления удобнее описывать «феноменологически», другие – «атомистически», а наиболее полное знание мы имеем в тех областях, где оба эти описания проникают друг в друга, как, например, истолкование классических термодинамических потенциалов методами статистической физики.

В данной работе будет рассматриваться исключительно «атомистический» подход. Не потому, что автор считает его «важнее» системно-динамического, а потому, что он оказывается примерно в половине случаев удобнее для моделирования реальных систем, и, следовательно, вполне заслуживает отдельного внимательного изучения (впрочем, как и системно-динамический).

Развивать атомистический подход к созданию сложных систем также можно двумя способами:

1. Дедуктивный способ. Предположим, что нужно создать сложную систему с заданной функциональ-

ностью (например, операционную систему или компилятор), а в выборе «атомов», реализующих различные функциональности мы свободны – разработка ведется с чистого листа. Тогда естественно поступить по-платоновски – проектировать систему сверху вниз, облекая основные идеи ее построения в соответствующие им формы, и воплощать эти формы в программный код. Идеалисты утверждают, что именно так и был создан этот мир. У программистов, для осуществления дедуктивного проектирования системы есть мощное средство – объектно-ориентированное программирование (ООП), чье наследование с возможностью переопределения методов, позволяет построить иерархию классов вполне в духе Платона.

2. Индуктивный способ. Предположим, что мы не создаем новые миры, а моделируем уже созданные. Тогда «атомы» системы жестко заданы предметной областью. Они определены и не могут быть иными, их свойства и способы взаимодействия также хорошо известны. Задачей является воспроизведение и изучение поведения всей сложной системы, состоящей из этих «атомов». Заметим, что в данном случае обычно иерархическая структура классов ООП мало чем может помочь – нужно уметь работать с теми «атомами», которые есть в системе, а они далеко не всегда представляют собой строгую иерархию. Подробно эта проблема будет освещена далее.

Заметим, что в жизни дедуктивный и индуктивный способы построения сложных систем связаны своего рода диалектикой. Допустим, мы далеко продвинулись по дедуктивному пути, например, построили .NET – иерар-

хическую систему из десятков тысяч классов. Теперь она всегда с нами, все классы отлажены, следовательно, вполне логично использовать их в своей работе, т. е., применять индуктивный способ, пользуясь кирпичиками .NET. Если же мы долгое время развивали индуктивный способ – со временем оказывается, что мы располагаем достаточно хаотичной коллекцией «атомов» с частично дублирующей друг друга функциональностью. Возникает естественное желание навести в этом богатстве определенный порядок – произвести дедуктивный реинжиниринг системы.

В данной работе будет рассматриваться исключительно индуктивный способ построения системы. Не потому, что он чем-то лучше дедуктивного, а потому, что класс задач мощностью примерно 25% (половина от половины) от всего на свете, заслуживает специального рассмотрения. Тем более что такое мощное средство как ООП, работает для него гораздо хуже, чем для дедуктивного способа.

В качестве примеров корпускулярных индуктивных моделей сложных систем приведем моделирование боевых действий в ходе штабных игр с уровнем детализации полк – дивизия – армия – фронт; изучение возможностей стратегической оборонной инициативы (СОИ) – исследование, в котором участвовал автор в конце 80-х; моделирование работы холдинга, объединяющего ряд промышленных и финансовых предприятий.

Полностью соглашаясь на интуитивном уровне с тем, что сложная система может сама состоять из сложных систем, мы не беремся давать здесь строгое определение этому понятию. Тем не менее понятие модели сложной системы будет введено вполне формально, как семейство родов структур в смысле Н. Бурбаки [13].

С этой точки зрения, сложная система – это то, что достаточно адекватно может быть представлено моделью сложной системы.

Род структуры – развитие понятия множества. Базисное множество снабжается структурой некоторого рода – вводится определенный тип отношений между его элементами, и в зависимости от этого типа отношений, множество может стать, например, группой или решеткой, или векторным пространством, или же в нашем случае – имитационной моделью сложной системы. При этом математический объект, например конкретное линейное пространство, является экземпляром структуры соответствующего рода.

Метод структурализма [17], идейно восходящий к Эрлангенской программе Ф. Клейна [21] и оказавшийся достаточно популярным и продуктивным в XX веке, в том числе и в гуманитарных науках, предлагает далее рассматривать различные преобразования базисных множеств и искать инварианты этих преобразований. Например, роды структур сохраняются при биекциях базисных множеств.

Школа член-корр. РАН Ю.Н. Павловского в ВЦ РАН в последние десятилетия развивала геометрическую теорию декомпозиции [23 – 25, 18] где с помощью морфизмов пытаются найти более простые представления различных математических объектов – их редукции и декомпозиции.

В данной работе, посвященной синтезу модели сложной системы из моделей ее компонент, нас более всего будет интересовать возможность распространения рода структуры на объединение базисных множеств математических объектов, наделенных этим родом структуры. Инвариантом относительно объединения базис-

ных множеств имитационных моделей, оказывается предложенный ниже способ организации имитационных вычислений.

Модельный синтез и анализ, как способ описания и синтеза имитационных моделей сложных многокомпонентных систем, развивался в отделе имитационных систем ВЦ РАН с конца 80-х гг. Основные его идеи и методы изложены в работах [1, 2, 8 – 12], однако сам термин «модельный синтез» впервые предлагается в данной работе. В основе модельного синтеза и анализа лежит понятие модели-компоненты.

С точки зрения построения программных систем, модель-компонента подобна объекту объектного анализа, но помимо характеристик снабженному не только методами, способными делать что-то полезное, если их вызовут, а неким аналогом операционной системы, всегда готовым давать стандартные ответы на стандартные запросы внутренней и внешней среды модели.

С формальной точки зрения, модель-компонента есть математический объект, базисным множеством которого является совокупность множеств внутренних и внешних характеристик модели, методов (того, что модель умеет делать) и событий (того, на что модель должна уметь реагировать). На базисном множестве вводится род структуры «модель-компонента», который обладает двумя замечательными свойствами:

1. Род структуры «модель-компонента» позволяет стандартным и однозначным образом организовать вычислительный процесс моделирования для всех объектов, снабженных структурой этого рода. Это означает возможность создания универсальной программы, способной запустить на выполнение любую имитационную модель, если та являет-

ся математическим объектом, снабженным структурой рода «модель-компонента».

2. Вообще говоря, если рассмотреть два произвольных математических объекта снабженных структурой одного рода (например, структурой абстрактной группы), то распространение этой структуры на объединение их базисных множеств возможно далеко не всегда. Тем не менее для математических объектов, наделенных родами структур из семейства родов структур «модель-компонента», наделение объединения их базисных множеств родом структуры из того же семейства родов структур «модель-компонента» или возможно (если подмножества характеристик их базисных множеств не имеют попарных пересечений), или возможно с некоторыми оговорками (например, при условии пополнения исходных объектов-компонент некоторым количеством дополнительных объектов-компонент, снабженных той же структурой).

Второе свойство позволяет образовывать из моделей-компонент путем объединения их базисных множеств модели-комплексы, которые после распространения общей структуры компонент на объединение их базисных множеств оказываются математическими объектами того же самого семейства родов структур «модель-компонента» и стало быть, снова могут объединяться в модели-комплексы. Первое свойство позволяет не впадать в отчаяние от сложности вычислительного процесса, получающейся в результате таких объединений сверхсложной фрактальной модели.

Для программной реализации сложной многокомпонентной модели предлагается модельно-ориентированная парадигма программирования, где единицей проектирования программного комплекса является модель-компонента конструкция более агрегированная по сравнению с объектом объектного анализа.

Ниже будет показано, что модельно-ориентированное программирование позволяет исключить наиболее сложное как для разработки, так и для отладки императивное программирование [11]. Кроме того, получаемый исполняемый код отличается высокой степенью параллельности, при этом степень параллельности кода возрастает при росте сложности модели. Данный факт может открыть перспективы для применения методов модельно-ориентированного программирования на высокопроизводительных вычислительных системах, в том числе и для задач, не связанных с имитационным моделированием, но имеющих многокомпонентную организацию.

Относительно предлагаемого в работе модельно-ориентированного подхода к программированию следует отметить два важных момента:

1. Несмотря на внешнее сходство терминов под этим подходом подразумевается нечто вполне отличное от MDA (Model Driven Architecture), MDE (Model Driven Engineering) и MDD (Model Driven Development), – различных вариантов концепции разработки программных систем, управляемой моделями. Подробнее об отличиях будет сказано в специальном разделе, посвященном сравнению модельного синтеза с объектным анализом.
2. Автор (хотя ему и приходилось программировать, быть может, больше и чаще, чем хотелось бы) от-

нюдь не считает себя профессиональным специалистом в области программирования. Поэтому пропагандирует модельно-ориентированный подход к программированию достаточно осторожно – не как универсальный метод решения всех на свете программистских проблем, а как метод, хорошо зарекомендовавший себя в четко ограниченной сфере его профессиональной деятельности – имитационном моделировании сложных многокомпонентных систем. Тем не менее, с определенной обосновываемой в работе надеждой, что предлагаемый метод, быть может, окажется хорош также и для некоторого более широкого класса концептуально близких программных систем, возможно и не связанных с имитационным моделированием.

1. Роды структур и элементы геометрической теории декомпозиции

«Математика – это искусство называть разные вещи одним и тем же именем».
А. Пуанкаре

Оригинальное изложение аппарата родов структур, содержащееся в работе [13], опирается на специфическую «бурбаковскую» терминологию и аксиоматику, отличающуюся от принятой в большинстве учебников. Тем не менее существуют работы, например [25] и [26], где этот аппарат излагается на основании обычной терминологии и аксиоматики. В русле именно этих работ, зачастую прямо цитируя их, мы и дадим здесь определение рода структуры и примеры математических объектов, наделенных различными родами структур. Начальные понятия геометрической теории декомпозиции и касающиеся их утверждения приводятся, следуя работам [23, 25]. Подробно познакомиться с теорией родов структур и с геометрической теорией декомпозиции, в том числе с доказательствами приводимых утверждений, можно в работах [13, 23-26].

1.1. Определение рода структуры

Род структуры объявляет о том, что математический объект будет состоять из частей $\sigma_1,\ldots,\sigma_r$ некоторых множеств. Это записывается следующим образом

$$\sigma_1 \subset S_1(X_1,\ldots,X_n,A_1,\ldots,A_m),\ldots,\ \sigma_r \subset S_r(X_1,\ldots,X_n,A_1,\ldots,A_m).$$

Здесь множества $X_1,...,X_n$ называются «базисными», множества $A_1,...,A_m$ – вспомогательными (употребляемая терминология почерпнута из [13]).

Множества $S_k(X_1,...,X_n,A_1,...,A_m)$, $1 \le k \le r$, – так называемые ступени, построенные по схеме S_k из базисных и вспомогательных множеств $X_1,...,X_n,A_1,...,A_m$. Ступени получаются путем применения к исходным множествам и/или уже имеющимся ступеням операций декартова произведения $\times$ и взятия множества всех подмножеств $\beta(\cdot)$. А именно:

1. По определению, X_i – ступень при любом $1 \le i \le n$. Аналогично, A_j – ступень при любом $1 \le j \le m$.
2. Если S – ступень, то и $\beta(S)$ – ступень.
3. Если S и S' – ступени, то и $S \times S'$ – ступень.
4. Других ступеней нет.

Схема S_k фиксирует исходные множества и порядок применения к ним двух указанных выше операций.

Базисные и вспомогательные множества играют разную роль в построениях родов структур (например, в построениях данной работы вспомогательные множества вообще не используются). Базисные множества должны быть обозначены разными буквами. На обозначения вспомогательных множеств таких ограничений не накладывается. Соотношения

$$\sigma_1 \subset S_1(X_1,...,X_n,A_1,...,A_m),...,\sigma_r \subset S_r(X_1,...,X_n,A_1,...,A_m)$$

называются соотношениями типизации. В род структур, кроме соотношений типизации может входить еще некоторое соотношение

$$R(X_1,...,X_n,\sigma_1,...,\sigma_r,A_1,...,A_m,\xi_1,...,\xi_s).$$

Это соотношение предъявляет к роду структуры некоторые требования. Здесь $\xi_1,...,\xi_s$ – соотношения, касающиеся множеств $A_1,...,A_m$. Вспомогательные множества $A_1,...,A_m$ и соотношения $\xi_1,...,\xi_s$ не преобразуются – отображения рассматриваются только над базисными множествами.

Соотношение $R(X_1,...,X_n,\sigma_1,...,\sigma_r,A_1,...,A_m,\xi_1,...,\xi_s)$ называется «аксиомой» данного рода структуры. К нему предъявляется требование: оно должно быть переносимо при биекциях. Это означает, что

$$(f_i;X_i \to X_i', i=1,2,...,n,) - \text{биекции}) \Rightarrow$$
$$(R(X,\sigma,A,\xi) \Rightarrow R(X'\sigma',A,\xi)), \text{ где } \sigma' = S(f,id_A).$$

Здесь $S(f,id_A)$ – так называемое «распространение» [13, 23] отображений f базисных множеств и тождественное распространение вспомогательных множеств по схеме S.

Род структуры будет обозначаться далее следующим образом:

$$\Sigma(A_1,...,A_n,\xi_1,...,\xi_s) =$$
$$= < X_1,...,X_n;[\sigma_1 \subset S_1(X,A),...,\sigma_r \subset S_r(X,A)];[R(X,\sigma,A,\xi)] >$$

или более коротко:

$$\Sigma[(A,\xi)] = < X;[\sigma \subset S(X,A)];[R(X,\sigma,A,\xi)] >.$$

Ломаные скобки «<» и «>» здесь играют роль ограничителя. Необязательные элементы рода структуры, как это принято при записи такого сорта конструкций, поставлены в квадратные скобки. Далее по возможности будет использоваться короткая форма всех соотношений.

Пусть имеются множества (E, τ) и выполняются соотношения $\tau \subset S(E, A)$ и $R(E, \tau, A, \xi)$, т. е. соотношение типизации и аксиома рода структуры $\Sigma[(A, \xi)]$ для множеств (E, τ). Тогда говорят, что объект (E, τ) снабжен структурой τ рода $\Sigma[(A, \xi)]$ или что τ есть структура рода $\Sigma[(A, \xi)]$ на множестве E. Обозначение $\Sigma[(A, \xi)]$ содержит то, что не меняется при переходе от одного $\Sigma[(A, \xi)]$-объекта к другому.

Это означает, что множества $A_1, \dots, A_m$ и соотношения $\xi_1, \dots, \xi_s$ являются одними и теми же для всех объектов данного рода структуры.

Тем самым, как уже говорилось, математические объекты делятся на классы. К классу относятся объекты, снабженные структурой данного рода $\Sigma(A, \xi)$.

1.2. Примеры родов структур

Приведем примеры родов структур.

$$MAPS = < X; \sigma \subset X \times X; (\forall x \in X)(\exists ! x' \in X)((x, x') \in \sigma)) > -$$

род структуры графика отображения множества в себя. Далее будет удобно оперировать краткими обозначениями аксиом родов структур. Как правило, такие обозначения будут образовываться следующим образом: обозначение будет начинаться с буквы А, далее будет ставиться обозначение рода структуры, далее в скобках будут ставиться не связанные кванторами буквы, фигурирующие в соотношениях, определяющих род структуры.

Например, обозначение для аксиомы

$$(\forall x \in X)(\exists ! x' \in X)((x, x') \in \sigma))$$

рода структуры графика отображения в себя в соответствии с этим правилом имеет вид $AMAPS(X,\sigma)$. MAPS-объект – это пара (E,τ), где $\tau \subset E \times E$ – бинарное отношение, называемое графиком отображения и удовлетворяющее аксиоме $AMAPS(E,\tau)$. В «обычных» математических текстах MAPS-объекты обозначаются как $f: E \to E$. Буква f в этих обозначениях иногда трактуется как график отображения, иногда как само отображение, т. е. пара (E,τ).

$$EQ = < X; \sigma \subset X \times X; AEQ(X,\sigma) > -$$

род структуры отношения эквивалентности. Аксиома $AEQ(X,\sigma)$ требует, чтобы отношение σ на X было рефлексивно, симметрично, транзитивно. Для того чтобы не загромождать изложение сложными формулами, здесь и в некоторых случаях далее аксиомы рода структуры формулируются словами. EQ-объект есть пара (E,Q), где отношение Q есть отношение эквивалентности.

В «обычных» математических текстах принадлежность пары (x,x') отношению эквивалентности Q принято обозначать $x \approx x'$, что читается как x эквивалентно x'. При этом в обычных математических текстах, как правило, не указывается, о каком отношении Q идет речь. Это должно быть ясно из контекста. Множество $\{x' | (x',x) \in Q\}$ называется классом эквивалентности, соответствующим x и обозначается x_Q. Множество $\{x_Q | x \in E\}$ называется фактор - множеством множества E по отношению эквивалентности Q и обозначается E_Q или E/Q.

$$PO = < X; \sigma \subset X \times X; APO(X,\sigma) > -$$

род структуры отношения частичного порядка. Аксиома $APO(X,\sigma)$ требует, чтобы отношение σ X было рефлексивно, антисимметрично, транзитивно. PO-объект есть пара (E,τ). В «обычных' математических текстах принадлежность пары (x,x') отношению частичного порядка τ принято обозначать $(x \le x')$, что читается как x предшествует x' или как x' следует за x. О каком отношении τ идет речь, обычно не указывается. Считается, что это должно быть ясно из контекста. Если к аксиоме APO добавить условие, для всякой пары или $x \le x'$, или $x' \le x$, то получится род структуры LP линейного порядка.

$$\text{GR}=< X;\sigma \subset X \times X \times X : AGR(X,\sigma) > -$$

род структуры абстрактной группы. GR-объект есть пара (E,τ), где (E,τ) – тернарное отношение на E, удовлетворяющее аксиоме $AGR(E,\tau)$. В «обычных» математических текстах принадлежность тройки (x,y,z) отношению τ обозначается как $x \cdot y = z$ и трактуется как постулирование на X ассоциативной алгебраической операции с нейтральным элементом, обозначаемым обычно через e, относительно которого каждый элемент обратим. Какому именно отношению τ, обычно не указывается.

В этих обозначениях $AGR(E,\tau)$ аксиома есть конъюнкция соотношений

$$(\forall x \in E)(\forall y \in E)(\ x \cdot y \in E), (\forall x \in X)(x \cdot e = x),$$
$$(\forall x \in X)(\exists x^{-1})(x \cdot x^{-1} = e).$$

Род структуры AGR абелевой группы получается из рода структуры GR «прибавлением» к аксиоме AGR условия коммутативности алгебраической операции.

$$\text{FLD}=< X;\sigma \subset X \times X \times X, \tau \subset X \times X \times X; AFLD(X,\sigma,\tau) > -$$

род структуры поля. Аксиома $AFLD(X,\sigma,\tau)$ утверждает, что $Card(X)\geq 2$, что σ и τ являются коммутативными, ассоциативными тернарными отношениями, первое из которых принято записывать аддитивно, второе – мультипликативно. Оба отношения обладают нейтральными элементами. Тернарные отношения связаны формулами дистрибутивности.

Самый важный для данной работы пример рода структуры, – семейство родов структур «модель-компонента» будет приведен ниже, в разд. 3.2.

1.3. Изоморфизмы, естественные канонические морфизмы

Определение 1.3.1 Пусть (E,τ) и (E',τ') - $\Sigma[(A,\xi)]$-объекты и $i:E\to E'$ отображение. Если это отображение биективно и $S(i,id_A)(\tau)=\tau'$, то i называется изоморфизмом объекта (E,τ) на объект (E',τ'). Если – $f:E\to E'$ произвольное отображение (не обязательно биекция) и имеет место $S(f,id_A)(\tau)\subset\tau'$, то f называется естественным каноническим морфизмом. Естественные канонические морфизмы для сокращения речи далее будут называться ЕКМ. Из этих определений следует, что всякий изоморфизм является естественным каноническим морфизмом.

Для многих родов структур естественные канонические морфизмы имеют специальные названия, возникшие в соответствующих этим родам структур теориях конкретных математических объектов.

Приведем примеры:

1. Для рода структуры SET ЕКМ — произвольные отображения.

2. Для родов структуры PO, LP, L частичного порядка, линейного порядка, решетки ЕКМ — монотонные неубывающие отображения.
3. Для рода структуры MAPS отображений в себя ЕКМ из объекта $f : E \to E$ в объект $f' : E' \to E'$ есть отображение $m : E \to E'$, такое, что $f' \circ m = m' \circ f$.
4. Аналогично определяются ЕКМ для рода структуры MAP.
5. Для родов структур GR, AL абстрактных групп, алгебраических решеток и, вообще, для всюду определенных алгебраических структур ЕКМ являются гомоморфизмы.

1.4. P- и F- редукции и декомпозиции

Определение 1.4.1 Пусть (E,τ) — $\Sigma(A,\xi)$-объект, $\tilde{E}$ — подмножество множества E, $\omega : \tilde{E} \to E$ — каноническая инъекция. $\Sigma(A,\xi)$-объект $(\tilde{E},\tilde{\tau})$ называется P-редукцией (или P-объектом) объекта (E,τ), если $\omega : \tilde{E} \to E$ является ЕКМ и для любого $\Sigma(A,\xi)$-объекта и любого отображения $g : E \to \tilde{E}$ из того, что $g \circ \omega$ является ЕКМ вытекает, что g является ЕКМ. Подмножество $\tilde{E}$ в этом случае называется P-множеством, объект $(\tilde{E},\tilde{\tau})$ вместе с названием P-редукция называется также подобъектом объекта (E,τ), структура $\tilde{\tau}$ — подструктурой структуры τ .

Множество P-множеств будет обозначаться $P(E,\tau)$. Также будет обозначаться множество P-редукций или P-объектов, поскольку между P-множествами и P-объектами имеется каноническая биекция.

Определение 1.4.2 Пусть (E,τ) – $\Sigma(A,\xi)$-объект, Q – отношение эквивалентности на E. $\pi: E \to E_Q$ – каноническая проекция. $\Sigma(A,\xi)$-объект (E_Q,τ_Q) называется F-редукцией (или F-объектом) объекта (E,τ), если $\pi: E \to E_Q$ является ЕКМ и для любого $\Sigma(A,\xi)$-объекта (E',τ') и любого отображения $g: E_Q \to E$ из того, что $\pi \circ g$ является ЕКМ вытекает, что g является ЕКМ. Множество E_Q в этом случае называется F-множеством, объект (E_Q,τ_Q) вместе с названием F-редукция называется также фактор-объектом объекта (E,τ), структура τ_Q называется фактор-структурой структуры τ. Множество F-множеств будет обозначаться $F(E,\tau)$, Также будет обозначаться множество F-редукций.

Понятие «редукция» употреблялось многими математиками. Каждый раз при употреблении этого понятия можно было вывести, каким точным смыслом оно наделялось. Впервые как переход от исходного объекта к «такому же объекту», определенному на подмножестве (фактор-множестве) того множества, на котором определен исходный объект, предложено в [18].

Предложение 1.4.1

Если Р-объект (F-объект) $\Sigma(A,\xi)$-объекта (E,τ) на множестве $\tilde{E}$ существует, то он единственен. ■

Данное утверждение кратко формулируется следующим образом: если подструктура существует, то она единственна.

Множества Р-объектов (F-объектов) $\Sigma(A,\xi)$-объекта (E,τ) не пусты, поскольку по определению (E,τ) является своим подобъектом (знак $\subset$ означает у нас или

принадлежность или равенство), а E_{id_E} отождествляется с E.

Множество $\mathrm{P}(E,\tau)$ снабжено структурой частичного порядка по включению Р-множеств. Точно такая же структура вводится на множестве $\mathrm{F}(E,\tau)$. Структуры частичного порядка, (часто это не только частичный порядок, но и решетка) возникающие на $\mathrm{P}(E,\tau)$ и $\mathrm{F}(E,\tau)$, сохраняются при ЕКМ.

Более точно, справедливо следующее

Предложение 1.4.2

Пусть (E,τ) – $\Sigma(A,\xi)$-объект, $(\tilde{E},\tilde{\tau})$ – его Р-редукция, $f:(E,\tau)\to(E_1,\tau_1)$ – изоморфизм, $\tilde{f}:\tilde{E}\to\tilde{E}_1$ – ограничение f на $\tilde{E}$, $\tilde{\tau}_1 = S(\tilde{f}, id_A)\ (\tilde{\tau})$. Тогда

$$i_1 = f\circ\omega\circ\tilde{f}^{-1}:\tilde{E}_1\to E_1 \text{ – ЕКМ.} \blacksquare$$

Аналогичное утверждение имеет место для F-редукций. Эти утверждения говорят о том, что структура частичного порядка «сохраняется» при изоморфизмах.

Определение 1.4.3

Возможен случай, когда имеется набор непересекающихся Р-редукций, дающих в совокупности (при объединении базисных множеств), весь исходный объект. Этот случай есть Р-декомпозиция исходного объекта.

Определение 1.4.4

F-декомпозиция объекта – это набор непересекающихся F-редукций, который в совокупности (при декартовом произведении базисных множеств), дает весь математический объект.

1.5. Редукции высших уровней

Предложение 1.5.1

P-редукция P-редукции есть P-редукция исходного объекта. F-редукция F-редукции есть F-редукция исходного объекта. ■

Определение 1.5.1

F-редукция P-редукции $\Sigma[(A,\xi)]$-объекта будет называться его PF-редукцией. P-редукция F-редукции $\Sigma[(A,\xi)]$-объекта будет называться его FP-редукцией. PF-редукции и FP-редукции будут называться редукциями второго уровня. Аналогично определяются редукции третьего уровня, к которым, например, относятся PFP-редукции и FPF-редукции, а также редукции более высоких уровней. Совокупность редукций объекта всех уровней будем называть множеством его редукций.

2. Математические модели. Внутренние и внешние характеристики. Гипотеза о замкнутости

> «Человек должен непреложно верить, что непостижимое на самом деле постижимо; иначе бы он прекратил исследования».
> *И.В. Гете*

Основная задача главы – дать содержательное пояснение с позиций математического и имитационного моделирования, формальной конструкции, семейству родов структур «модель-компонента», которая будет определена в следующей главе.

Прежде чем перейти к строгому определению имитационной модели сложной системы как рода структуры, остановимся на таких важных понятиях, связанных с математическим и имитационным моделированием, как замкнутость модели и ее внутренние и внешние характеристики. Поскольку процесс построения математических моделей различных явлений не формализован и в значительной мере остается искусством, обсуждение будет вестись также на не слишком формальном уровне.

2.1. Математические модели

Предположим, что мы изучаем некоторое явление, которое хотим моделировать. Шанс изучить это явление методами точных наук появляется, если удается измерить и численно выразить интересующие нас характеристики этого явления. Характеристики изучаемого явления изменяются со временем, однако в этих изменениях заключается нечто постоянное – закономерность их связи между собой, которая и является предметом выявления и изучения путем математического моделирования.

Численное измерение характеристик изучаемого явления задает некоторое их отображение во множество характеристик будущей математической модели. Математическая модель была бы идеальной, если бы это отображение сохраняло связи между характеристиками явления и модели – некий аналог рода структуры.

Однако это мечты сохранить «то – не знаю что», ведь закономерности изменения характеристик явления как раз и есть предмет изучения с помощью математического моделирования. Лучшее, что можно сделать – это придумать такие соотношения между характеристиками модели, которые бы в максимальной степени уподобляли бы их изменения изменениям их прообразов – характеристик изучаемого явления. Правда, к сожалению, такое уподобление – лишь необходимое, а не достаточное условие построения адекватной модели.

В самом деле, если бы мы располагали закономерностью связи характеристик, как инвариантом преобразования характеристик явления в характеристики модели – мы, несомненно, были бы способны давать любые прогнозы динамики этого явления. С другой стороны, располагая даже очень точной, многократно проверенной на практике математической моделью явления, нельзя быть уверенным, что однажды не появится необходимость рассмотрения некоторых не учитывавшихся ранее его характеристик. Учет новых характеристик может потребовать совсем других моделей для построения адекватного прогноза развития явления. Например, с помощью геоцентрической модели движения планет Птолемея, которой человечество успешно пользовалось более полутора тысяч лет, до сих пор можно весьма точно рассчитать взаимное расположение планет на небосводе или даты наступления лунных и солнечных затме-

ний. Однако запустить с ее помощью ракету на Луну, было бы, по-видимому, проблематично.

2.2. Внутренние и внешние характеристики

Предположим, что мы измеряем и изучаем численные характеристики $X_1, X_2, \ldots, X_n$, нашего явления. Поскольку ни одно явление невозможно полностью изъять из окружающего мира, в наше поле зрения неизбежно попадает некий набор характеристик $a_1, a_2, \ldots, a_m$ внешнего по отношению к изучаемому явлению мира, так или иначе связанных с характеристиками изучаемого явления. Характеристики внешнего мира, совсем не связанные с нашим явлением, в рамках его изучения мы вправе игнорировать. Отметим, что граница между характеристиками явления и внешнего по отношению к нему мира достаточно размыта, подвижна, условна, – обычно исследователь сам определяет, что включать, а что нет в изучаемое им явление. Остановимся подробнее на возможных типах зависимостей между характеристиками X_i и a_j.

1. Предположим, что некоторое подмножество $\widetilde{A} \subset \{a_j\}_{j=1}^m$ характеристик внешнего мира, во-первых, влияет на характеристики $\{X_i\}_{i=1}^n$ изучаемого явления и, во-вторых, зависит от этих характеристик. Предлагается включить такие характеристики $\widetilde{A}$ во множество характеристик изучаемого явления и исключить из рассматриваемых характеристик внешнего мира, тем самым расширив нашу модель, ибо такова структура взаимосвязей между ее характеристиками.

2. Предположим, что некоторое подмножество $A' \subset \{a_j\}_{j=1}^m$ характеристик внешнего мира не влияет на характеристики $\{X_i\}_{i=1}^n$ изучаемого явления, но зависит от этих характеристик. Предлагается совсем не рассматривать такие характеристики A', поскольку изучение влияния нашего явления на внешний по отношению к нему мир, хотя вполне может быть содержательной задачей, но это самостоятельная задача, отличная от провозглашенной нами программы исследования зависимостей между характеристиками изучаемого явления.
3. Наконец остается последнее предположение: что подмножество $\overline{A} \subset \{a_j\}_{j=1}^m$ характеристик внешнего мира влияет на характеристики $\{X_i\}_{i=1}^n$ изучаемого явления, но не зависит от этих характеристик. Только такие характеристики внешнего мира мы и будем рассматривать в дальнейшем, называя их внешними характеристиками явления и его моделей. Характеристики $\{X_i\}_{i=1}^n$ будем называть внутренними характеристиками.

Внутренние характеристики зависят друг от друга и от внешних характеристик. Внешние характеристики влияют на внутренние, но не зависят от них. Иногда утверждение о независимости внешних характеристик от внутренних называют гипотезой об инвариантности.

Приведенные выше рассуждения показывают, что принятие такой гипотезы не уменьшает общности рассмотрения, – просто те внешние характеристики, которые зависят от внутренних, удобно включать в модель и считать дополнительными внутренними характеристиками.

2.3. Гипотеза о замкнутости

Приступая к построению математической модели изучаемого явления, хотелось бы иметь уверенность в том, что такое построение возможно. Это центральная идея данной главы, вынесенная в эпиграф. Гипотеза о замкнутости как раз и призвана дать исследователю такую уверенность.

Гипотеза о замкнутости модели постулирует, что знания значений внутренних и внешних характеристик модели в момент t достаточно для вычисления ее внутренних характеристик на некотором интервале $(t, t+\Delta t)$. Таким образом, внутренние характеристики, с одной стороны, определяют состояние модели, а с другой стороны, могут прогнозироваться с помощью этой модели на некотором интервале времени. Внешние характеристики не моделируются, но влияют на модель. Поэтому их можно считать доступными для восприятия модели характеристиками внешнего по отношению к ней мира (в том числе, например, внешними управляющими воздействиями). Обычно предполагается, что внешние характеристики доступны для измерения в любой момент модельного времени. Иногда предположение о наблюдаемости внешних характеристик включают в гипотезу о замкнутости. В данной работе, упоминая гипотезу о замкнутости, всегда будем считать, что предположение о наблюдаемости внешних характеристик включается в нее.

Из гипотезы о замкнутости следует, что если теперь нас интересует изменение выбранных нами внутренних характеристик, например во времени, то ему просто не от чего больше зависеть, кроме как от самих этих характеристик (в виртуальном мире, образуемом нашей замкнутой моделью, больше просто ничего нет).

Теперь, например, если есть веские основания считать зависимость внутренних характеристик модели от времени абсолютно непрерывной, мы можем искать ее в виде обыкновенных дифференциальных уравнений:

$$\frac{dX_i}{dt} = F_i(X_1,\ldots,X_n,a_1,\ldots,a_m),\ i = 1,\ldots,n.$$

Правда правая часть этой системы дифференциальных уравнений неизвестна, ее еще предстоит придумать, это и будет созданием математической модели.

Если кроме изменения во времени нас интересует распределение одних внутренних характеристик относительно других, получим систему дифференциальных уравнений в частных производных.

2.4. Гипотеза о замкнутости применительно к моделям сложных многокомпонентных систем

Как и прежде, будем считать, что в любой момент времени мы умеем узнавать значения внешних характеристик модели, и предполагать, что знания значений внутренних и внешних характеристик модели в момент t достаточно для вычисления ее внутренних характеристик на некотором интервале $(t, t+\Delta t)$.

Для моделей сложных систем предположение о непрерывной зависимости характеристик от времени не является естественным. Очень часто здесь зависимости некоторых характеристик от времени явно носят дискретный характер. Кроме того, все вычисления характеристик модели осуществляются, как правило, с помощью компьютера. Попробуем обсудить возможности компьютерного построения моделей сложных многокомпонентных систем.

Отметим, что реализуя имитационные вычисления на компьютере, за конечное время мы сможем обрабо-

тать лишь конечное число разрывов первого рода. Поэтому будем искать траекторию нашей модели $\vec{X}(t)$ в классе кусочно-гладких по t функций.

Далее, поскольку шаг моделирования Δt выбирается так, что на протяжении этого шага изменения модели не слишком заметны, будем всегда относить разрыв первого рода на начало шага, а далее на протяжении Δt будем считать траекторию модели $\vec{X}(t)$ гладкой. Траекторию модели будем вычислять с некоторой заданной точностью $\varepsilon > 0$. Например, если траектория модели определяется дифференциальным уравнением $\dot{X} = F(X, a)$, то при малых $\tau > 0$ справедливо

$$X(t+\tau) \approx X(t) + F(X(t), a(t))\tau .$$

Попытаемся теперь построить траекторию модели $\vec{X}(t)$ на некотором макроскопическом отрезке модельного времени $[0,T]$, попутно выясняя, в каких предположениях может быть успешным такое построение.

Определение 2.4.1

Будем называть модель замкнутой в точке $t \in [0,T)$, если найдется число $\Delta t > 0$, $t + \Delta t \in (0,T]$, которое будем называть *отрезком прогноза модели для точки* t, такое что:

1. На основании внутренних и внешних характеристик модели $\vec{X}(t)$ и $\vec{a}(t)$ можно определить, есть ли в точке t разрыв траектории $\Delta\vec{X}(t)$, и если он есть, – вычислить его.
2. Далее, на полуинтервале $(t, t+\Delta t]$, траектория модели, выходящая из точки $\vec{X}(t) + \Delta\vec{X}(t)$, является

гладкой функцией времени, например, решением системы дифференциальных уравнений
$\dot{X} = F_t(X,a)$.

Определение 2.4.2

Будем называть модель локально замкнутой на отрезке $[0,T]$, если она замкнута в любой точке $t \in [0,T)$.

Определение 2.4.3

Будем называть модель прогнозируемой или лапласовской на отрезке $[0,T]$, если существует конечное разбиение этого отрезка точками $0 = t_0 < t_1 <,\ldots,< t_n = T$, такое что модель замкнута в каждой из точек t_{i-1}, $i = 1,\ldots,n$, и каждый из полуинтервалов $(t_{i-1}, t_i]$, $i = 1,\ldots,n$, принадлежит отрезку прогноза для своего левого конца.

Очевидно, если модель прогнозируема в смысле последнего определения и удалось найти соответствующее разбиение отрезка $[0,T]$ точками $0 = t_0 < t_1 <,\ldots,< t_n = T$, то задачу построения модели можно считать в принципе решенной. С другой стороны, если не выполнены условия определения 2.4.2., т. е. существуют точки $t \in [0,T)$, откуда невозможно сделать даже малый шаг на $\Delta t > 0$, то задача построения модели представляется безнадежной. Поэтому требование локальной замкнутости на отрезке будет одним из базовых требований, предъявляемых к модели.

Из локальной замкнутости на отрезке, прогнозируемость на этом отрезке, вообще говоря, не следует.

Контрпримером является модель, описываемая условиями задачи, встречающейся в курсе математики начальной школы. Два пешехода идут навстречу друг другу с постоянной скоростью. Между ними летает муха с постоянной по абсолютной величине скоростью, большей, чем скорость любого из пешеходов. Как только муха

долетает до одного из пешеходов, она тут же разворачивается и летит к другому. (В задаче спрашивается, сколько пролетит муха до момента встречи пешеходов.)[1]

Здесь всей истории модели недостаточно для определения скорости мухи в момент встречи пешеходов. Действительно, скорость мухи разрывна, и в момент встречи пешеходов имеет два предельных значения: $v_{мухи}$ и $-v_{мухи}$. Поэтому в точке встречи пешеходов старая история приключений мухи полностью заканчивается, а новая не может начаться, без дополнительных предположений относительно ее скорости в этот момент.

Предложение 2.4.1

Сделать локально замкнутую на отрезке $[0,T]$ модель прогнозируемой на этом отрезке может добавление требования непрерывности слева траектории модели $\Delta\vec{X}(t)$ в любой точке $t \in (0,T]$.

Действительно, начнем продвигаться по оси времени из исходной точки 0 в соответствии с определением замкнутости в точке. Если за конечное число шагов мы не доходим до точки T, это означает, что существует точка накопления $\tilde{t} \in (0,T)$. В силу непрерывности траектории модели слева существует предел слева $\tilde{X} = \lim_{t \to \tilde{t}-0} X(t)$ в точке накопления. При этом для выбранной нами точности вычислений $\varepsilon > 0$ найдется такое

[1] Иногда эту задачу связывают с именем Джона фон Неймана, – якобы он мгновенно выдал ее ответ, сказав при этом, что просуммировал в уме бесконечный ряд. Этот вариант задачи известен как «муха фон Неймана», в нем вместо пешеходов фигурируют поезда, которые в момент встречи сталкиваются, и в катастрофе гибнет все, в том числе и муха. Поэтому в модели «муха фон Неймана» никакое продолжение ее истории за момент встречи поездов невозможно.

$\tilde{t} > \delta > 0$, что для всех $t \in (\tilde{t} - \delta, \tilde{t})$ справедливо $\varepsilon > \left|\tilde{X} - X(t)\right|$. До точки $\tilde{t} - \delta$ мы дойдем за конечное число шагов, иначе она, а не $\tilde{t}$ была бы точкой накопления. На интервале $(\tilde{t} - \delta, \tilde{t})$ положим $X(t) = \tilde{X}$, и, таким образом, дойдем за конечное число шагов до точки $\tilde{t}$, после чего в соответствии с локальной замкнутостью модели на отрезке $[0, T]$ можем продолжить путь дальше, на ненулевой шаг, что противоречит нашему предположению о том, что $\tilde{t}$ – точка накопления. ■

Как мы видим, непрерывность слева является достаточным условием, чтобы локально замкнутая на отрезке модель стала прогнозируемой на этом отрезке. Из определений замкнутости в точке и прогнозируемости модели на отрезке следует также, что непрерывность слева есть и необходимое условие прогнозируемости (возможно, после переопределения траектории модели в конечном числе точек).

Отметим, что приведенное выше рассуждение не просто провозглашает возможность пройти отрезок моделирования за конечное число шагов (при условии непрерывности траектории слева), а дает конструктивный способ, как это сделать. Нужно начинать с левого конца отрезка и «перешагивать» встречающиеся точки накопления – те отрезки, где траектория системы с выбранной точностью моделирования $\varepsilon > 0$, остается постоянной.

2.5. Особенности моделирования сложных многокомпонентных систем

Особенностью многокомпонентных «атомистических» систем является то, что их атомы внутри себя связаны гораздо сильнее, чем между собою. Большую часть времени каждый атом живет сам по себе в соответствии

со своим устройством, и лишь время от времени атомы взаимодействуют между собою. Часто такое взаимодействие бывает мгновенным (относительно выбранного в модели масштаба времени) и, следовательно, вызывает разрывность характеристик модели.

Такие особенности устройства многокомпонентных систем дают шанс построения синтеза в духе приводившейся во введении цитаты из Н.П. Бусленко: «…дать компонентам возможность в полной мере проявить себя». А именно в начале шага моделирования обсчитать все межкомпонентные взаимодействия, а затем в течение некоторого Δt до следующего взаимодействия компоненты проявляют себя независимо друг от друга и, скорее всего, такое проявление описывается соответствующими системами дифференциальных уравнений. Поэтому представление траектории системы кусочно-гладкой функцией представляется вполне уместным.

Моделировать сложную систему чаще всего приходится с переменным шагом времени. Очень часто приходится уменьшать стандартный шаг моделирования до ближайшей точки синхронизации компонент. В связи с этим возникает вопрос, а не зациклится ли модель, как это происходит с упомянутой выше моделью «пешеходы и муха». Другими словами, дойдем ли мы до правого конца отрезка моделирования? Как следует из предыдущего раздела, для успеха этого предприятия кроме достаточно обычного в моделировании требования локальной замкнутости нужно потребовать непрерывности слева траектории системы в любой точке.

Содержательно это означает, что для успеха моделирования мало уметь сделать хотя бы небольшой шаг вперед из любой точки – еще обязательно нужно, чтобы

всякое значение траектории модели, кроме начального, было обусловлено некоторой предысторией.

Требование непрерывности слева как достаточного условия возможности построения модели на самом деле близко к условию необходимому. То, что мы в состоянии создать компьютерную реализацию имитационной модели всегда означает, что мы проходим весь рассматриваемый отрезок модельного времени за конечное число шагов, поэтому и число разрывов первого рода траектории модели, которые мы можем распознать и учесть, конечно. В промежутках между разрывами чаще всего используется линейное приближение траектории, вычисляемое по левому концу интервала на основании гипотезы о замкнутости. Подобное необходимое условие возможности построения модели и было зафиксировано в определении 2.4.3 прогнозируемой, или лапласовской модели. Кусочно-гладкую траекторию модели всегда можно считать непрерывной слева, относя все разрывы первого рода к началу, а не к концу интервалов непрерывности.

Прогнозируемая модель была названа в соответствующем определении также и лапласовской, потому что для нее оказывается существенным не только прогноз – взгляд в будущее из любой точки, но и обусловленность – взгляд в прошлое, где текущее состояние есть результат некоторых существовавших ранее и осуществившихся в нем причин, его определивших. На этот счет у П.С. Лапласа [19] имеется следующее высказывание: «Современные события имеют с событиями предшествующими связь, основанную на очевидном принципе, что никакой предмет не может начать быть без причины, которая его произвела...».

Кроме того, прогнозируемая на отрезке модель, по-видимому, являет собой образец лапласовского детерминизма. Во всяком случае, именно в прогнозируемой на отрезке имитационной модели в смысле определения 2.4.3 воплощается мечта Лапласа, высказанная им в работе [20]:

«Ум, которому были бы известны для какого-либо данного момента все силы, проявляющиеся в природе, и относительное положение всех ее составных частей (если бы вдобавок этот ум оказался достаточно обширным, чтобы подчинить эти данные анализу), – обнял бы в одной формуле движения величайших тел вселенной наравне с движениями легчайших атомов: не осталось бы ничего, что было бы для него недостоверно, и будущее так же, как и прошедшее, предстало бы перед его взором».

В заключение раздела скажем несколько слов о детерминированности и однозначности вычислительного процесса имитационного моделирования. Детерминированность и однозначность вычислительного процесса соотносится с гипотезой о замкнутости примерно так же, как соотносятся конструктивные и неконструктивные теоремы о существовании. Фактически постулируется способность разработчика построить функционирующую модель изучаемого явления, то есть реализовать детерминированный и однозначный процесс вычислений последующих характеристик явления в зависимости от их предыдущих значений и предыдущих значений внешних характеристик.

Заметим, что детерминированность вычислительного процесса моделирования вовсе не означает детерминированности самой модели. Модель может быть стохастической от начала и до конца, однако ее разработчик

должен четко знать, когда включить генератор случайных чисел и как однозначно вычислить величины, которые в данной модели считаются случайными.

Разработчик может заранее не знать, как поведет себя модель в тот или иной промежуток модельного времени (собственно, очень часто модель разрабатывается именно для того, чтобы наблюдать и прогнозировать с ее помощью поведение моделируемой системы в той или иной ситуации). Однако он всегда обязан знать, как об этом детерминировано и однозначно узнавать во время имитационного эксперимента, иначе никакой модели не получится.

Если детерминированность вычислительного процесса постулирует возможность вычисления любой из характеристик модели в любой из моментов модельного времени, то однозначность означает, что если в некоторый момент модельного времени почему-то появляется несколько значений одной и той же характеристики модели (например, ее вычисляют различными способами), то должен быть детерминированный механизм выбора из этого множества единственного значения. Этот выбор может осуществляться, например, посредством свертки имеющегося набора значений с некоторыми весами, а может быть и вероятностным.

Заметим также, что высказанное выше требование детерминированности и однозначности вычислительного процесса есть некий идеал, к которому следует стремиться при создании моделей. Реально в достаточно сложных системах, например в известных операционных системах, эксплуатирующихся десятилетиями, время от времени находятся уязвимости – с точки зрения моделирования, недокументированные цепочки внешних характеристик, приводящие к неожиданному для разработчи-

ка результату. Например, к получению злоумышленником доступа к ресурсам компьютера. Такого рода уязвимости приходится отлаживать – выявлять и устранять. В упомянутом выше примере – десятилетиями.

Далее, однозначность, вообще говоря, не означает единственности. То, что мы в любой ситуации знаем, как поступать дальше – несомненное благо с точки зрения организации вычислительного процесса, но это вовсе не означает, что нельзя было бы поступать каким-либо иным разумным и детерминированным образом, например, при выборе единственного значения характеристики, полученной несколькими способами.

2.6. Основные выводы из гипотезы о замкнутости

Возможно, все высказанные в данной главе соображения и предложения покажутся простыми и даже тривиальными. Тем не менее, из них вытекает ряд весьма важных для дальнейшего изложения следствий, которые будут перечислены далее:

1. Коль скоро мы признаем на каждом отрезке прогноза $[t, t+\Delta t]$ функциональную зависимость от значений характеристик модели в момент t как величины возможного скачка траектории в момент t, так и последующей ее гладкой динамики на $(t, t+\Delta t]$, – естественно признать и возможность вычисления этих функциональных зависимостей в функциональной парадигме программирования.
2. Функциональная зависимость однозначна по определению. Отсюда следует, что если по каким-то соображениям (например, проистекающим из предметной области моделирования, или из способа организации вычислительного процесса), нам представляется удобным декомпозировать эту за-

висимость на ряд параллельно выполняемых вычислений, то эта декомпозиция не должна нарушать упомянутой однозначности. Таким образом, если два параллельных процесса пытаются одновременно модифицировать одну и ту же характеристику модели – это есть просто ошибка ее проектировщика – нарушение однозначности функциональной зависимости (а отнюдь не отражение объективной сложности модели или моделируемой ею системы). Стало быть, декомпозированные на параллельные процессы вычисления функциональной зависимости, не эквивалентны исходным. Сохранение однозначности вычислений – необходимое условие такой эквивалентности.

3. Из определения 2.4.3. прогнозируемой или лапласовской модели следует, что процесс вычисления ее траектории естественным образом декомпозируется на чередующиеся вычисления «скачков» и интервалов гладкой динамики. Траекторию модели можно считать полунепрерывной слева. Разрывов траектории модели первого рода может быть лишь конечное число, они происходят мгновенно в модельном времени и зависят только от предела слева характеристик модели в момент разрыва. На интервалах гладкой динамики внутренние характеристики модели непрерывно зависят от ее характеристик на левом конце интервала и от времени прошедшего от начала интервала. Именно эта декомпозиция и определяет предлагаемые в следующей главе правила организации имитационных вычислений (аксиому поведения).

На сформулированные выше выводы мы будем существенно опираться в последующих разделах работы.

3. Модельный синтез – концепция описания и реализации имитационных моделей сложных многокомпонентных систем

«Одно во всем, и все в одном».
Сэнцань

Для облегчения понимания довольно громоздких формальных конструкций разд. 3.2, дадим сначала неформальное описание двух самых важных понятий модельного анализа: основного понятия – «модель-компонента» и вспомогательного понятия – «модель-комплекс».

3.1. Неформальное описание моделей-компонент и моделей-комплексов

Сначала опишем устройство модели-компоненты.

Характеристики

Модель-компонента, подобно объекту объектного анализа имеет характеристики. Эти характеристики мы будем разбивать на внутренние и внешние. Внутренние характеристики – это то, что модель-компонента моделирует, внешние – это то, что она знает о внешнем мире.

Процессы

Функциональность модели-компоненты удобно структурировать следующим образом. Считается, что модель-компонента реализует один или несколько параллельно выполняющихся процессов. Процесс состоит в последовательном чередовании элементов – алгоритмически элементарных методов. Если какой-либо процесс какую-то часть модельного времени не выполняется – удобно считать, что в это время он выполняет «пустой»

элемент, не меняющий никаких характеристик модели-компоненты.

Элементы

Элементарные, алгоритмически однородные методы, реализующие функциональности модели-компоненты (т.е. то, что компонента умеет делать, получение на основании значений некоторых внутренних и внешних характеристик модели-компоненты, новых значений некоторых ее внутренних характеристик).

По отношению к модельному времени некоторые элементы выполняются мгновенно, это сосредоточенные или быстрые элементы. Быстрыми элементами можно моделировать характеристики системы, имеющие разрывы первого рода, или принимающие только целочисленные значения.

Выполнение других элементов занимает определенное время. Если при этом элемент для любого промежутка времени $\Delta\tau$, не превосходящего стандартный шаг моделирования Δt выдает некий осмысленный результат, такой элемент называется распределенным или медленным. Распределенные элементы – естественное средство вычисления непрерывных характеристик модели.

Может оказаться и так, что выполнение элемента занимает определенное модельное время, но результат его действия наступает лишь в конце, после полного выполнения элемента, т. е. никаких промежуточных результатов за время меньшее полного времени выполнения нет. Такие элементы называются условно-распределенными. Вообще говоря, условно-распределенные элементы можно не рассматривать как отдельный класс, а моделировать парой: пустой распределенный элемент, за которым идет сосредоточенный, выдающий результат.

Частью предлагаемой концепции является жесткая дисциплина работы методов с внутренними характеристиками модели: каждый метод имеет право изменять только «свои» характеристики. Эта дисциплина основана на принятии предположения о детерминированности и однозначности имитационных вычислений. В рамках предлагаемой концепции конфликт доступа возникающий, когда методы A и B вычисляют одну и ту же характеристику $X = X_A$ и $X = X_B$, может быть разрешен, например, введением метода C, который, получая на входе в качестве параметров X_A и X_B, вычисляет на их основании искомую характеристику X, устраняя тем самым не только конфликт доступа к ней, но и, очевидно, имевшую место неоднозначность вычисления упомянутой характеристики. Принятая дисциплина доступа к характеристикам позволяет вызывать параллельно те методы, которые в модельном времени выполняются одновременно.

Наконец, последнее, что следует заметить относительно элементов. Как и всякий метод, каждый элемент имеет входные и возвращаемые параметры. Концепция моделирования предполагает возможность распределенного вычисления элементов, т. е. элемент, вообще говоря, может быть найден разработчиком модели-компоненты на просторах интернета, поэтому его параметры таковы, какими их сделал его автор, и, скорее всего, никак не согласованы с характеристиками модели-компоненты, которые придумал ее разработчик. Поэтому при описании компоненты обязательно должно быть уделено внимание описанию коммутаций входных параметров элементов с внутренними и внешними характеристиками компоненты и выходных параметров элементов – с ее внутренними характеристиками.

События

Содержательно, события – это то, что нельзя пропустить при моделировании динамики системы – точки синхронизации различных ее функциональностей, представляемых процессами. Точки, когда получены такие значения характеристик модели и внешних характеристик, на которые обязаны отреагировать некоторые процессы модели-компоненты.

Формально событие – функция значений внутренних и внешних характеристик в начале шага моделирования. С точки зрения организации имитационных вычислений, событие – это метод, входными параметрами которого является подмножество внутренних и внешних характеристик модели-компоненты, а выходной параметр один – прогнозируемое время до наступления этого события. Если это прогнозируемое время равно нулю, значит, событие уже наступило. События управляют чередованием элементов в процессе.

Для каждой упорядоченной пары элементов процесса $\{A,B\}$, если между ними возможен переход, то ему обязан соответствовать метод-событие $E_{\{A,B\}}$, прогнозирующий время этого перехода. Возможны также события вида $E_{\{A,A\}}$, прерывающие выполнение элемента A, например, если его еще не нужно заканчивать, но он вычислил характеристики модели-компоненты, которые могут повлечь смену элементов других процессов. Процесс перехода должен быть однозначным. Одновременное наступление событий $E_{\{A,B\}}$ и $E_{\{A,C\}}$ говорит лишь о том, что разработчик модели при ее проектировании упустил из своего рассмотрения этот случай, которому, быть может, должен соответствовать переход $\{A,D\}$.

Возможно тем, кто знаком с архитектурой компьютера или с событийно-ориентированными языками программирования такое определение событий покажется странным, и вызовет вопросы типа, а где же здесь что-нибудь подобное обработчикам событий? Дело в том, что в моделировании (если, конечно не брать моделирование полунатурное, когда к компьютерному комплексу подключают реальное изделие «в железе») чаще всего наступление события должна определить сама модель – даже если это событие неожиданное, например, случайное, – все равно, в нужный момент модель вполне детерминированно должна запустить генератор случайных чисел, чтобы выяснить, не произошло ли оно. А подобный порядок действий вполне адекватно ложится на изложенную выше концепцию событий.

Выполнение модели-компоненты

Модель-компонента элементарная, но тем не менее, полноправная модель сложной системы. Поэтому ее можно запустить на выполнение имитационного эксперимента, конечно, если имеются начальные данные и способ нахождения внешних характеристик модели-компоненты в моменты событий.

Правила запуска модели-компоненты на выполнение следующие. Во-первых, задается стандартный шаг моделирования Δt. Во-вторых, считается, что в начале шага моделирования известны текущие элементы всех процессов и все внутренние и внешние характеристики модели. Далее

1. Вычисляются события, связанные с текущими элементами процессов. **Все эти события можно вычислять параллельно.** Если есть наступившие события, проверяется, нет ли переходов к быстрым (сосредоточенным) элементам, если они

есть – **параллельно** выполняются быстрые элементы (они становятся текущими), затем возврат к началу п.1; если нет переходов к быстрым элементам – совершаются переходы к новым медленным (распределенным) элементам, затем возврат к началу п.1.

2. Если нет наступивших событий – из всех прогнозов событий выбирается ближайший $\Delta\tau$.
3. Если стандартный шаг моделирования не превосходит прогнозируемого времени до ближайшего события, $\Delta t \le \Delta\tau$ – **параллельно** моделируем текущие распределенные элементы со стандартным шагом Δt . В противном случае – **параллельно** моделируем их с шагом времени до ближайшего спрогнозированного события $\Delta\tau$.
4. Возвращаемся к началу п.1.

Все события и элементы есть функции в математическом смысле и в смысле парадигмы функционального программирования, т.е. не имеют состояний и побочных эффектов, что может дать при их вычислении дополнительные возможности распараллеливания.

В связи с приведенными правилами выполнения модели-компоненты могут возникнуть вопросы: не может ли выполнение быстрых элементов в п. 1, а также уменьшение шага времени в п. 3 привести к зацикливанию программы за счет возникновения точек накопления системных событий. Полную гарантию, как было показано в разд. 2, можно дать лишь для кусочно-гладких непрерывных слева систем.

Еще заметим, что раз мы требовали от элементов однозначности имитационных вычислений и сумели этого добиться – на выполнение все одновременно выполняющиеся в модельном времени элементы можно запус-

кать асинхронно, т.е. загружать ими имеющиеся ядра процессора или же доступные распределенные компьютеры. Если же этого почему-то добиться не удалось – у системы поддержки выполнения модели есть полная информация о том, кто что меняет (собственно, это она должна будет по выполнении элементов обновить соответствующие данные в базе) – и она обязана выдать соответствующую диагностику ошибки времени выполнения.

Модели-комплексы

Модели-компоненты могут объединяться в модели-комплексы, при этом (необязательно) может оказаться, что некоторые компоненты явно моделируют внешние характеристики некоторых других компонент. Для того чтобы полностью описать модель-комплекс, достаточно указать:

1. Какие модели-компоненты и в каком количестве экземпляров в него входят.
2. Коммутацию моделей-компонент внутри модели-комплекса, если она имеет место, т. е., какие внутренние характеристики каких моделей-компонент, являются какими внешними характеристиками и каких именно компонент модели-комплекса.

При объединении компонент в комплекс, следует иметь в виду, что однозначность вычислительного процесса может быть потеряна. Так может произойти, если по каким-то причинам, несколько компонент вычисляют одну и ту же характеристику моделируемого явления. В таком случае можно ввести в модель-комплекс новую компоненту, которая в качестве внешних характеристик получает весь многозначный набор значений упомянутой характеристики, а в качестве внутренней характери-

стики каким-то образом вычисляет единственное ее значение.

Комплекс как модель-компонента

Модель-комплекс, состоящий из многих моделей-компонент, вовне может проявляться в качестве единой модели-компоненты.

Введем следующую операцию объединения компонент комплекса:

1. Внутренними характеристиками комплекса объявляется объединение внутренних характеристик всех его компонент.
2. Процессами комплекса объявляется объединение всех процессов его компонент.
3. Методами комплекса объявляется объединение всех методов его компонент.
4. Событиями комплекса объявляется объединение всех событий его компонент.
5. Внешними характеристиками комплекса объявляется объединение всех внешних характеристик его компонент, из которого исключаются все те характеристики, которые моделируются явно какими-либо компонентами.

Операция объединения превращает модель-комплекс в модель-компоненту. Этот факт позволяет строить модель как фрактальную конструкцию, сложность которой (и соответственно подробность моделирования) ограничивается лишь желанием разработчика.

3.2. Семейство родов структур «модель-компонента»

Попробуем формально определить математический объект, представляющий элементарную имитационную модель сложной системы. Предыдущий раздел на не-

формальном уровне поясняет, почему ее стоит определять именно таким образом.

Введем однопараметрическое семейство родов структур Σ_N «модель-компонента». Параметром N семейства Σ_N является количество процессов модели-компоненты. Что такое процесс – на неформальном уровне было определено в предыдущем разделе. Формально процессы модели-компоненты будет определены ниже соотношениями типизации (10) и аксиомами R_9.

$$\Sigma_N =< X, M, E, \{M_j\}_{j=1}^N, \{E_j\}_{j=1}^N;$$

$$x \subset X,$$

$$a \subset X,$$

$$s \subset M,$$

$$f \subset M,$$

$$\{m_{j,real} \subset M_j \times M\}_{j=1}^N,$$

$$\{e_{j,real} \subset E_j \times E\}_{j=1}^N,$$

$$\{m_{j,in} \subset M_j \times \beta(X)\}_{j=1}^N,$$

$$\{m_{j,out} \subset M_j \times \beta(X)\}_{j=1}^N,$$

$$\{e_{j,in} \subset E_j \times \beta(X)\}_{j=1}^N,$$

$$\{m_j^0 \subset M_j\}_{j=1}^N,$$

$$\{sw_j \subset E_j \times M_j \times M_j\}_{j=1}^N,$$

$$\left\{p_j \subset \beta(M_j) \times \beta(E_j) \times M_j \times \beta(E_j \times M_j \times M_j)\right\}_{j=1}^{N};$$

$$R_1:\ (x \cup a = X)\,\&\,(x \cap a = \emptyset),$$

$$R_2:\ (s \cup f = M)\,\&\,(s \cap f = \emptyset),$$

$$R_3:\ \left\{\left(\forall m \in M_j\right)\left(\exists\,!\tilde{m} \in M\right)\left(\{m, \tilde{m}\} \in m_{j,real}\right)\right\}_{j=1}^{N},$$

$$R_4:\ \left\{\left(\forall e \in E_j\right)\left(\exists\,!\tilde{e} \in E\right)\left(\{e, \tilde{e}\} \in e_{j,real}\right)\right\}_{j=1}^{N},$$

$$R_5:\ \left\{\left(\forall m \in M_j\right)\left(\exists\,!r \in \beta(X)\right)\left(\{m, r\} \in m_{j,in}\right)\right\}_{j=1}^{N},$$

$$R_6:\ \left\{\left(\forall m \in M_j\right)\left(\exists\,!r \in \beta(x)\right)\left(\{m, r\} \in m_{j,out}\right)\right\}_{j=1}^{N},$$

$$R_7:\ \left\{\left(\forall e \in E_j\right)\left(\exists\,!r \in \beta(X)\right)\left(\{e, r\} \in e_{j,in}\right)\right\}_{j=1}^{N},$$

$$R_8:\ \left\{\left(\left(\forall e \in E_j\right)\left(\exists\,!r \in M_j \times M_j\right)\left(\{e, r\} \in sw_j\right)\right)\& \right.$$
$$\left.\&\left(\left(\{e, r\} \in sw_j, \{\tilde{e}, \tilde{r}\} \in sw_j, r = \tilde{r}\right) \Rightarrow \left(e = \tilde{e}\right)\right)\right\}_{j=1}^{N}$$

$$R_9:\ \left\{p_j = \left\{M_j, E_j, m_j^0, sw_j\right\}\right\}_{j=1}^{N},$$

R_{10}: аксиома однозначности вычисления характеристик модели-компоненты,

R_{11}: аксиома поведения модели-компоненты (организации имитационных вычислений)>.

Обозначение $\{\ldots_j\}_{j=1}^{N}$ используется для краткости и означает, что содержимое скобок повторяется через запятую N раз, при этом индекс j заменяется на $1,\ldots,N$. Например, $\{M_j\}_{j=1}^{N}$ есть краткий вариант записи $M_1,\ldots,M_N$.

Дадим пояснения приведенному выше формальному определению рода структуры «модель-компонента». Оно заключено в угловые скобки «<» и «>» и состоит из трех частей, которые отделяются друг от друга символом «;», а отдельные определения в каждой части разделяются символом «,». В первой части определяются базисные множества. Во второй части перечислены соотношения типизации, которые определяют родовые константы, как части множеств, построенных из базисных специальным образом – возможным последовательным применением операций декартова произведения $\times$ и взятия множества всех подмножеств $\beta(\cdot)$. В третьей части перечислены аксиомы рода структуры R_1 – R_9, которые устанавливают определенные зависимости между базисными множествами и родовыми константами.

В качестве основных базисных множеств выбраны

$$X, M, E, \{M_j\}_{j=1}^{N}, \{E_j\}_{j=1}^{N}$$

Здесь X – множество характеристик модели, причем иногда будем разделять его на два подмножества: $X = \{x, a\}$, где x — внутренние характеристики модели, то, что в соответствии с гипотезой о замкнутости полностью определяет ее состояние, а a — ее внешние характеристики, то, что в силу той же гипотезы о замкнутости полностью определяет взаимодействие модели с внешним по отношению к ней миром. Первые два из соотношений типизации

$$x \subset X\ ,\ a \subset X\ , \tag{1}$$

утверждают, что внутренние характеристики модели x и внешние характеристики a — есть части ее характеристик. Аксиома R_1: $(x \cup a = X)\,\&\,(x \cap a = \varnothing)$ утверждает,

что эти части x и a не пересекаются, и в совокупности составляют все множество характеристик модели X .

Далее идет M — множество различных реализаций методов-элементов, элементарных умений нашей модели, среди которых мы также иногда будем выделять два подмножества: $M = \{s, f\}$ — медленных s, реализующих гладкую зависимость внутренних характеристик модели от ее внутренних и внешних характеристик, если например, $\dot{x} = F(x,a)$, то $x(t+\Delta t) \approx x(t) + F(x(t),a(t))\Delta t$, и быстрых f — реализующие скачки внутренних характеристик модели: $\Delta x = G(x(t),a(t))$. Точно так же, как и в случае с характеристиками, запишем соответствующие соотношения типизации:

$$s \subset M\ ,\ f \subset M\ , \qquad (2)$$

и аксиому: R_2: $(s \cup f = M)\ \&\ (s \cap f = \emptyset)$.

Далее, E — множество различных реализаций методов-событий, связанных с моделью. События – это то, на что обязана реагировать наша модель. Метод-событие – функция, которая, получая на входе подмножество характеристик $Y \subset X$, на выходе дает неотрицательное число, означающее, что событие наступило, если число равно нулю, или же прогноз времени до наступления события, если число положительно.

Отметим, что во множествах X, M, E нет одинаковых элементов. Этот факт также утверждается аксиомой рода структуры R_{10}. Для X это следует из того, что речь идет о модели, и принцип неувеличения числа сущностей сверх необходимости (бритва Оккама) побуждает нас избегать ненужного дублирования одних и тех же характеристик как моделируемого явления, так и его связей с внешним миром. Что касается M и E, словами «множе-

ство различных реализаций» подчеркивается именно уникальность методов в этих множествах. На самом деле, язык математики хорош именно тем, что иногда позволяет одними и теми же соотношениями записать различные законы, имеющие место в совсем различных предметных областях. Излагаемый здесь подход позволяет учитывать этот факт и пользоваться им. Однако для этого необходимо знать, сколько и каких различных функциональных зависимостей имеется в нашем распоряжении. Множества M и E как раз и являются такими «хранилищами» различных функциональных зависимостей для модели-компоненты.

Далее будут упоминаться процессы модели-компоненты. Подробное объяснение того, что такое процессы будет дано далее, пока же скажем, что содержательно процессы подобны, например, системным службам операционной системы компьютера. Они работают всегда и при этом одновременно. Будем считать, что число процессов модели – N. Каждый процесс p_j, $j=1,\ldots,N$, последовательно осуществляет некоторый конечный набор возможных для него элементарных действий M_j, который будем называть множеством его методов-элементов, возможно, в зависимости от возникающих в системе ситуаций E_j, на которые процесс умеет реагировать, их будем называть множеством его методов-событий.

Этим исчерпывается описание базисных множеств.

Вспомогательных множеств нет.

Перейдем теперь к соотношениям типизации и аксиомам. Уже упоминавшееся соотношение $x \subset X$ определяет внутренние характеристики модели как подмножество всех ее характеристик.

$$\{m_{j,real} \subset M_j \times M\}_{j=1}^{N} \quad (3)$$

Для каждого процесса p_j задается отображение множества его методов-элементов M_j во множество их реализаций M. Таким образом, для любого $m \in M_j$ однозначно определяется его реализация $\tilde{m} \in M$. Формально это можно записать как аксиомы

$$R_3 : \{(\forall m \in M_j)(\exists ! \tilde{m} \in M)(\{m, \tilde{m}\} \in m_{j,real})\}_{j=1}^{N}.$$

Разница между имеющими одинаковую реализацию методами, с точки зрения процесса, может быть, например, в способе коммутации их параметров с характеристиками модели, о чем речь пойдет ниже. Не запрещаются (хотя и не рекомендуются) и просто «синонимы».

$$\{e_{j,real} \subset E_j \times E\}_{j=1}^{N} \quad (4)$$

Для каждого процесса p_j задается отображение множества его методов-событий E_j во множество их реализаций E. Для любого $e \in E_j$ однозначно определяется его реализация $\tilde{e} \in E$. Аксиомы

$$R_4 : \{(\forall e \in E_j)(\exists ! \tilde{e} \in E)(\{e, \tilde{e}\} \in e_{j,real})\}_{j=1}^{N}$$

утверждают этот факт. Так же, как и в случае методов-элементов, различные с точки зрения процесса события, могут иметь одинаковые реализации.

$$\{m_{j,in} \subset M_j \times \beta(X)\}_{j=1}^{N} \quad (5)$$

Коммутируются внутренние и внешние характеристики модели с входящими параметрами методов-

элементов j-го процесса. Включение должно определить, какое подмножество характеристик модели передается каждому из методов-элементов. Это можно выразить аксиомами

$$R_5: \{(\forall m \in M_j)(\exists ! r \in \beta(X))(\{m, r\} \in m_{j,in})\}_{j=1}^N.$$

$$\{m_{j,out} \subset M_j \times \beta(X)\}_{j=1}^N \qquad (6)$$

Коммутируются возвращаемые параметры методов-элементов с внутренними характеристиками модели. При этом должны выполняться аксиомы

$$R_6: \{(\forall m \in M_j)(\exists ! r \in \beta(x))(\{m, r\} \in m_{j,out})\}_{j=1}^N.$$

$$\{e_{j,in} \subset E_j \times \beta(X)\}_{j=1}^N \qquad (7)$$

Коммутируются внутренние и внешние характеристики модели с входящими параметрами методов-событий j-го процесса. Включение должно определить, какое подмножество характеристик модели передается каждому из методов-событий. При этом должны выполняться аксиомы

$$R_7: \{(\forall e \in E_j)(\exists ! r \in \beta(X))(\{e, r\} \in e_{j,in})\}_{j=1}^N.$$

Соотношениями

$$\{sw_j \subset E_j \times M_j \times M_j\}_{j=1}^N \qquad (8)$$

определяются переключения методов-элементов в каждом процессе. Если при определенных условиях возможно переключение процесса с выполнения метода $A \in M_j$

на выполнение метода $B \in M_j$ (а таким условием в силу гипотезы о замкнутости может быть лишь некое сочетание внутренних и внешних характеристик модели $\{x, a\} \in X$, – в виртуальном мире модели просто ничего больше нет), то должен быть создан единственный метод-событие $e_{AB}(x, a) \in E_j$, прогнозирующий и вычисляющий такое переключение. Возможны также события вида $e_{AA}(x, a)$, не переключающие метод-элемент $A \in M_j$ на другой, а лишь прерывающие его выполнение (например, для синхронизации результатов его вычислений с другими процессами модели). Таким образом, время и порядок переключений элементов каждого процесса определяется тем, что происходит внутри и вне модели. Вообще говоря, в самых простых процессах, событий может и не быть, тогда не будет и переключений. Если же множество событий j-го процесса – E_j, непусто, то соотношение типизации (8) должно удовлетворять следующим аксиомам

$$R_8: \{((\forall e \in E_j)(\exists ! r \in M_j \times M_j)(\{e, r\} \in sw_j))\&$$
$$\&((\{e, r\} \in sw_j, \{\tilde{e}, \tilde{r}\} \in sw_j, r = \tilde{r}) \Rightarrow (e = \tilde{e}))\}_{j=1}^{N},$$

утверждающим, что отображения множеств методов-событий во множества переключений биективны.

Соотношение

$$\{m_j^0 \subset M_j\}_{j=1}^{N} \qquad (9)$$

определяет начальные методы-элементы процессов.

Соотношения типизации

$$\{p_j \subset \beta(M_j) \times \beta(E_j) \times M_j \times \beta(E_j \times M_j \times M_j)\}_{j=1}^{N} \qquad (10)$$

определяют процессы модели, а соответствующий им набор аксиом

$$R_9 : \{p_j = \{M_j, E_j, m_j^0, sw_j\}\}_{j=1}^{N},$$

уточняет это понятие. Процесс – это множества элементов и событий, начальный элемент и правила переключений. Множества M_j и E_j – это множества элементов и событий j-го процесса; m_j^0 – его начальный элемент; sw_j – правила переключения элементов, которые были подробно описаны выше.

Формальное определение семейства родов структур «модель-компонента» на этом полностью заканчивается.

Тем не менее дополним набор аксиом семейства родов структур «модель-компонента» еще двумя. Эти аксиомы в некотором смысле стоят особняком, поскольку ничего не добавляют к формальному определению отношений, задаваемых на базисных множествах рассматриваемого семейства родов структур – упомянутые отношения уже полностью определены. Смысл дополнительных аксиом R_{10} и R_{11} в том, чтобы описать, что и как мы собираемся делать дальше с математическими объектами семейства родов структур «модель-компонента».

Аксиома R_{10} требует однозначности вычисления характеристик модели-компоненты. Возможность однозначности вычисления характеристик постулируется гипотезой о замкнутости и подробно обсуждалась в разд. 2.3 – 2.5. Для модели-компоненты это требование означает, например, что два метода-элемента разных процессов не должны одновременно менять одну и ту же характеристику. А гипотеза о замкнутости утверждает, что этого возможно добиться. Собственно, для модели-

компоненты этим все и ограничивается, и можно было бы не выделять это требование в аксиому. Однако, при объединении компонент в комплекс, все оказывается не так просто, и эта аксиома начинает работать. Поэтому пока можно считать ее заготовкой на будущее, которая потребуется в следующем разд. 3.3.

Аксиома поведения R_{11} задает следующие правила запуска на счет любого математического объекта семейства родов структур «модель-компонента».

Во-первых, считается, что в начале шага моделирования известны текущие элементы всех процессов и все внутренние характеристики модели (на первом шаге – это начальные значения внутренних характеристик и начальные элементы процессов).

Во-вторых, предполагается, что внешние характеристики модели возможно определить в любой момент модельного времени.

Далее

1. Вычисляются события связанные с текущими элементами процессов. Связь событий с текущими элементами процессов определяется правилами переключений (8). Вычисляться события могут параллельно, однако для продвижения вычислительного процесса далее, следует дождаться завершения вычислений всех событий. Если есть наступившие события, проверяется, нет ли переходов к быстрым элементам из множеств $\{f_j\}_{j=1}^{N}$, если они есть – выполняются соответствующие быстрые элементы (они становятся текущими). Вычисляться они могут также параллельно, однако для продвижения вычислительного процесса далее, следует дождаться завершения вычислений всех быстрых элементов, затем возврат к началу

п.1; если нет переходов к быстрым элементам – совершаются переходы к новым медленным элементам из множеств $\{s_j\}_{j=1}^{N}$, затем возврат к началу п.1.

2. Если нет наступивших событий – из всех прогнозов событий выбирается ближайший $\Delta\tau$.
3. Если стандартный шаг моделирования Δt не превосходит прогнозируемого времени до ближайшего события, $\Delta t \le \Delta\tau$ – вычисляем текущие медленные элементы со стандартным шагом Δt. В противном случае вычисляем их с шагом времени до ближайшего спрогнозированного события $\Delta\tau$. Медленные элементы из множеств $\{s_j\}_{j=1}^{N}$, также можно вычислять параллельно, с ожиданием завершения последнего.
4. Возвращаемся к началу п.1.

Еще раз проиллюстрируем соотношения типизации и аксиомы семейства родов структур «модель-компонента» следующей таблицей:

Таблица соотношений типизации и аксиом

Соотношения типизации, пояснение	Аксиома, пояснение
$x \subset X$, $a \subset X$ x и a – множества внутренних и внешних характеристик модели, X – всех ее характеристик.	$R_1: (x \cup a = X) \& (x \cap a = \emptyset)$ Подмножества внутренних и внешних характеристик не пересекаются и дают в совокупности все характеристики.
$s \subset M$, $f \subset M$ s и f – медленные и быстрые методы-элементы, M – все элементы.	$R_2: (s \cup f = M) \& (s \cap f = \emptyset)$ Подмножества медленных и быстрых элементов не пересекаются и дают в совокупности все элементы.

Соотношения типизации, пояснение	Аксиома, пояснение
$\{m_{j,real} \subset M_j \times M\}_{j=1}^{N}$ Отображение методов-элементов j-го процесса в общее хранилище методов-элементов M.	$R_3: \{(\forall m \in M_j)(\exists! \tilde{m} \in M)(\{m, \tilde{m}\} \in m_{j,real})\}_{j=1}^{N}$ У каждого элемента из M_j есть единственная реализация в M.
$\{e_{j,real} \subset E_j \times E\}_{j=1}^{N}$ Отображение методов-событий j-го процесса в общее хранилище методов-событий E.	$R_4: \{(\forall e \in E_j)(\exists! \tilde{e} \in E)(\{e, \tilde{e}\} \in e_{j,real})\}_{j=1}^{N}$ У каждого события из E_j есть единственная реализация в E.
$\{m_{j,in} \subset M_j \times \beta(X)\}_{j=1}^{N}$ Коммутация входных параметров методов-элементов.	$R_5: \{(\forall m \in M_j)(\exists! r \in \beta(X))(\{m, r\} \in m_{j,in})\}_{j=1}^{N}$ У каждого $m \in M_j$ есть единственный набор входных параметров из X.
$\{m_{j,out} \subset M_j \times \beta(X)\}_{j=1}^{N}$ Коммутация возвращаемых параметров методов-элементов.	R_6: $\{(\forall m \in M_j)(\exists! r \in \beta(x))(\{m, r\} \in m_{j,out})\}_{j=1}^{N}$ У каждого элемента из M_j есть единственный набор возвращаемых в x параметров.
$\{e_{j,in} \subset E_j \times \beta(X)\}_{j=1}^{N}$ Коммутация входных параметров методов-событий.	R_7: $\{(\forall e \in E_j)(\exists! r \in \beta(X))(\{e, r\} \in e_{j,in})\}_{j=1}^{N}$ У каждого события из E_j есть единственный набор входных параметров из X.

Соотношения типизации, пояснение	Аксиома, пояснение
$\{m_j^0 \subset M_j\}_{j=1}^N$ Начальные методы-элементы процессов.	
$\{sw_j \subset E_j \times M_j \times M_j\}_{j=1}^N$ Правила переключений методов-элементов j-го процесса.	R_8: $\{((\forall e \in E_j)(\exists ! r \in M_j \times M_j)(\{e,r\} \in sw_j))\&$ $\&((\{e,r\} \in sw_j, \{\tilde{e},\tilde{r}\} \in sw_j, r = \tilde{r}) \Rightarrow (e = \tilde{e}))\}_{j=1}^N$ Каждое событие из E_j, определяет единственное переключение между двумя (возможно, даже одинаковыми), методами-элементами из M_j; и отображения событий в переключения биективны.
$\{p_j \subset \beta(M_j) \times \beta(E_j) \times M_j \times$ $\times \beta(E_j \times M_j \times M_j)\}_{j=1}^N$ Процессы.	R_9: $\{p_j = \{M_j, E_j, m_j^0, sw_j\}\}_{j=1}^N$ Процесс – это множества элементов и событий, начальный элемент и правила переключений.
	R_{10}: аксиома однозначности вычисления характеристик модели-компоненты
	R_{11}: аксиома поведения модели-компоненты

Таким образом, род структуры «модель-компонента» полностью формально определен и даже частично содержательно объяснен.

3.3. Синтез модели-комплекса из моделей-компонент

Пусть теперь есть два математических объекта из семейства родов структур «модель-компонента» Σ_N и $\Sigma_{N'}$, с базисными множествами $X, M, E, \{M_j\}_{j=1}^{N}, \{E_j\}_{j=1}^{N}$ и $X', M', E', \{M'_j\}_{j=1}^{N'}, \{E'_j\}_{j=1}^{N'}$ соответственно. Определим модель-комплекс, составленную из этих двух моделей-компонент. Назовем базисными множествами комплекса следующий набор множеств:

$$X \cup X', M \cup M', E \cup E', \{M_j\}_{j=1}^{N+N'}, \{E_j\}_{j=1}^{N+N'};$$

где $\{M_j\}_{j=1}^{N+N'}$ и $\{E_j\}_{j=1}^{N+N'}$ обозначают $M_1,\dots,M_N,M'_1,\dots,M'_{N'}$ и $E_1,\dots,E_N,E'_1,\dots,E'_{N'}$ соответственно. Объединения множеств $M \cup M'$ и $E \cup E'$ понимаем в самом обычном смысле:

$$M \cup M' = \{m : (m \in M) \vee (m \in M')\},$$

$$E \cup E' = \{e : (e \in E) \vee (e \in E')\}.$$

Что касается множеств X и X', будем считать, что они не пересекаются, т.е. если даже $X \cap X' \neq \emptyset$, договоримся, что мы умеем различать характеристики из $X \cap X'$ по признаку их происхождения из X или X'. Таким образом, непустые элементы $X \cap X'$ входят в объединение $X \cup X'$ дважды, как $X \cap X' \subset X$ и $X \cap X' \subset X'$, и различаются. Содержательный смысл такого предположения – мы собрали вместе две модели-компоненты, но они никак не взаимодействуют между собой – каждая

живет собственной жизнью так, как если бы она была одна.

Покажем, что на базисных множествах комплекса можно задать род структуры $\Sigma_{N+N'}$ из семейства «модель-компонента». Будем проверять выполнение следующих соотношений типизации:

$$x \cup x' \subset X \cup X',\ a \cup a' \subset X \cup X', \tag{11}$$

$$s \cup s' \subset M \cup M',\ f \cup f' \subset M \cup M', \tag{12}$$

$$\left\{m_{j,real} \subset M_j \times (M \cup M')\right\}_{j=1}^{N+N'}, \tag{13}$$

$$\left\{e_{j,real} \subset E_j \times (E \cup E')\right\}_{j=1}^{N+N'}, \tag{14}$$

$$\left\{m_{j,in} \subset M_j \times \beta(X \cup X')\right\}_{j=1}^{N+N'}, \tag{15}$$

$$\left\{m_{j,out} \subset M_j \times \beta(X \cup X')\right\}_{j=1}^{N+N'}, \tag{16}$$

$$\left\{e_{j,in} \subset E_j \times \beta(X \cup X')\right\}_{j=1}^{N+N'}, \tag{17}$$

$$\left\{m_j^0 \subset M_j\right\}_{j=1}^{N+N'}, \tag{18}$$

$$\left\{sw_j \subset E_j \times M_j \times M_j\right\}_{j=1}^{N+N'}, \tag{19}$$

$$\left\{p_j \subset \beta(M_j) \times \beta(E_j) \times M_j \times \beta(E_j \times M_j \times M_j)\right\}_{j=1}^{N+N'}; \tag{20}$$

и аксиом:

R_1: $(x \cup x' \cup a \cup a' = X \cup X')\ \&\ ((x \cup x') \cap (a \cup a') = \varnothing)$,

$$R_2:\ (s \cup s' \cup f \cup f' = M \cup M')\ \&\ ((s \cup s') \cap (f \cup f') = \varnothing),$$

$$R_3:\ \left\{\left(\forall m \in M_j\right)\left(\exists !\tilde{m} \in M \cup M'\right)\left(\{m, \tilde{m}\} \in m_{j,real}\right)\right\}_{j=1}^{N+N'},$$

$$R_4:\ \left\{\left(\forall e \in E_j\right)\left(\exists !\tilde{e} \in E \cup E'\right)\left(\{e, \tilde{e}\} \in e_{j,real}\right)\right\}_{j=1}^{N+N'},$$

$$R_5:\ \left\{\left(\forall m \in M_j\right)\left(\exists !r \in \beta(X \cup X')\right)\left(\{m, r\} \in m_{j,in}\right)\right\}_{j=1}^{N+N'},$$

$$R_6:\ \left\{\left(\forall m \in M_j\right)\left(\exists !r \in \beta(x \cup x')\right)\left(\{m, r\} \in m_{j,out}\right)\right\}_{j=1}^{N+N'},$$

$$R_7:\ \left\{\left(\forall e \in E_j\right)\left(\exists !r \in \beta(X \cup X')\right)\left(\{e, r\} \in e_{j,in}\right)\right\}_{j=1}^{N+N'},$$

$$R_8:\ \Big\{\left(\left(\forall e \in E_j\right)\left(\exists !r \in M_j \times M_j\right)\left(\{e, r\} \in sw_j\right)\right)\&$$
$$\&\left(\left(\{e, r\} \in sw_j, \{\tilde{e}, \tilde{r}\} \in sw_j, r = \tilde{r}\right) \Rightarrow \left(e = \tilde{e}\right)\right)\Big\}_{j=1}^{N+N'}$$

$$R_9:\ \left\{p_j = \left\{M_j, E_j, m_j^0, sw_j\right\}\right\}_{j=1}^{N+N'},$$

R_{10}: аксиома однозначности вычисления характеристик модели-компоненты,

R_{11}: аксиома поведения модели-компоненты (организации имитационных вычислений).

Очевидно, соотношения типизации (11) – (20) и аксиомы R_1 – R_9 выполняются, поскольку как для компоненты Σ_N, так и для $\Sigma_{N'}$, выполняются соотношения (1) – (10) и соответствующие аксиомы R_1 – R_9.

Для примера рассмотрим самые громоздкие соотношения типизации (20). Пусть сначала $1 \le j \le N$, тогда в силу (10) справедливо:

$$p_j \subset \beta(M_j) \times \beta(E_j) \times M_j \times \beta(E_j \times M_j \times M_j) \subset$$
$$\subset \beta(M_j) \times \beta(E_j) \times M_j \times \beta(E_j \times M_j \times M_j).$$

Пусть теперь $N < j \le N + N'$. Тогда снова в силу (10) заключаем

$$p'_j \subset \beta(M'_j) \times \beta(E'_j) \times M'_j \times \beta(E'_j \times M'_j \times M'_j) \subset$$
$$\subset \beta(M'_j) \times \beta(E'_j) \times M'_j \times \beta(E'_j \times M'_j \times M'_j).$$

Самый громоздкий набор аксиом R_8, равно как и R_9, проверяется совсем уж просто – это всего лишь объединение соответствующих наборов аксиом для объектов Σ_N и $\Sigma_{N'}$, поэтому для примера проверим аксиомы R_6.

Пусть $1 \le j \le N$, тогда, в силу выполнения R_6 для Σ_N справедливо

$$\left(\forall m \in M_j\right)\left(\exists!\, r \in \beta(x)\right)\left(\{m, r\} \in m_{j,out}\right).$$

Если $r \in \beta(x)$, тем более $r \in \beta(x \cup x')$, поскольку

$$\beta(x) \subset \beta(x \cup x').$$

Следовательно, справедливо:

$$\left(\forall m \in M_j\right)\left(\exists!\, r \in \beta(x \cup x')\right)\left(\{m, r\} \in m_{j,out}\right).$$

Аналогично, если $N < j \le N + N'$, тогда в силу выполнения R_6 для $\Sigma_{N'}$ справедливо

$$\left(\forall m' \in M'_j\right)\left(\exists!\, r' \in \beta(x')\right)\left(\{m', r'\} \in m'_{j,out}\right).$$

Если $r' \in \beta(x')$, то тем более $r' \in \beta(x \cup x')$, поскольку

$$\beta(x') \subset \beta(x \cup x').$$

Следовательно, справедливо

$$\left(\forall m' \in M'_j\right)\left(\exists!\, r' \in \beta(x \cup x')\right)\left(\{m', r'\} \in m'_{j,out}\right),$$

что и завершает проверку выполнения аксиом R_6.

Из сказанного выше заключаем, что наш комплекс является математическим объектом семейства родов структур «модель-компонента» $\Sigma_{N+N'}$.

Аксиома поведения R_{11} по определению такова, что применима к любому объекту семейства «модель-компонента».

Обсудим теперь аксиому однозначности вычисления характеристик модели-компоненты R_{10}.

Поскольку мы «принудительно» решили, что

$$X \cap X' = \emptyset, \tag{21}$$

то однозначность вычисления характеристик следует из таковой, имевшей место для каждой из компонент. Правда, при этом наши компоненты никак не взаимодействуют друг с другом, они просто формально объединены вместе.

Обратим внимание на то, что формального критерия выполнения условия (21) быть не может. Формально можно лишь добиться выполнения (21), всегда считая разными характеристики, первоначально принадлежащие разным компонентам, как это и было сделано. Определить, какие характеристики модели одинаковы, а какие отличаются, может лишь разработчик, исходя из своих представлений о ее предметной области. Тем не менее если разработчик указал, что условия (21) нарушены, и конкретизировал, в чем состоят эти нарушения, можно указать на ряд вполне формальных действий, которые следует предпринять, чтобы исправить ситуацию.

Отметим, что если бы мы понимали объединение характеристик комплекса самым обычным образом:

$$X \cup X' = \{y : (y \in X) \vee (y \in X')\},$$

не различая «происхождение» объединенных характеристик, то, как не трудно видеть, соотношения типизации (11) – (20) и аксиомы R_2 – R_9 также были бы выполнены. При этом в случае $X \cap X' \neq \emptyset$, имело бы место взаимодействие между компонентами за счет пересечения части их характеристик. Однако гарантировать однозначность вычисления объединенных характеристик (аксиома R_{10}), вообще говоря, было бы невозможно. Действительно, в этом случае уже никакие гипотезы о замкнутости не запрещают двум методам-элементам разных процессов (первоначально принадлежащих разным компонентам), одновременно изменять одну и ту же характеристику, если эта характеристика принадлежит $X \cap X'$.

Попробуем найти примирение двух приведенных выше крайностей. Будем считать, во-первых, что попрежнему все элементы объединения $X \cup X'$ формально различимы, т.е., формально $X \cap X' = \emptyset$. Во-вторых, предположим, что с содержательной точки зрения, наоборот, некоторые характеристики компонент одинаковы (например, обе модели-компоненты пытаются дать прогноз погоды на завтра), т.е., с точки зрения семантики модели $X \cap X' \neq \emptyset$.

Начнем с самого простого. Пусть пересекаются внешние характеристики наших компонент, например, по характеристике $\tilde{a}$. Это означает, что внешняя характеристика $a_i = \tilde{a}$ первой модели и внешняя характеристика $a'_j = \tilde{a}$ второй, отражают один и тот же (иначе они бы различались) фактор внешнего по отношению к нашим моделям мира. Из общих представлений о единстве и объективности внешнего мира следует, что значения этих характеристик (конечно, правильно идентифицированные) должны совпадать, и поэтому, в моделикомплексе вполне достаточно одной из них. Исключим

для определенности a' из $X \cup X'$. Тогда во всех соотношениях коммутации (15) и (17), куда входит a', следует заменить a' на a, что позволит этим соотношениям сохранить прежний смысл в модели-комплексе. Если кому-то соображения о «единстве и объективности мира» кажутся не слишком убедительными, – ниже для устранения пересечения внутренних характеристик компонент, будет предложен иной способ, который вполне может быть применен также и в данном случае.

Пусть теперь пересекается множество внутренних характеристик одной модели-компоненты с множеством внешних характеристик другой.

Например, пусть $x \cap a' = \tilde{y} = x_i = a'_j$. Это означает, что если во второй модели-компоненте характеристика a'_j не моделировалась, а являлась фактором внешнего по отношению к этой компоненте мира, то в модели-комплексе та же характеристика моделируется явно, так как явно моделируется в первой модели-компоненте внутренней ее характеристикой x_i. Стало быть, в модели-комплексе внешняя характеристика второй компоненты $\tilde{y} = a'_j$ становится внутренней характеристикой первой модели-комплекса $\tilde{y} = x_i$. Стало быть, из объединения $X \cup X'$ следует исключить a'_j, а во все соотношения коммутации из (15) и (17), куда входит $\tilde{y} = a'_j$, вместо него включить $\tilde{y} = x_i$.

Таким образом, в полном соответствии с шутливой программистской поговоркой: «Описанная ошибка становится новой полезной возможностью», – можно не только объединять компоненты в комплекс, но и использовать в модели-комплексе то, что некоторые из компонент, быть может, моделируют нечто, что без них не уме-

ли другие, извлекая определенные преимущества из усложнения модели путем создания комплексов.

Пусть, наконец, пересекаются внутренние характеристики моделей-компонент. К сожалению, звучавшие в случае внешних характеристик соображения о единстве и объективности мира, здесь не убеждают. Наоборот, всем хорошо известно, что могут быть десятки различающихся прогнозов погоды или курсов валют на завтра. Пусть внутренние характеристики двух моделей-компонент пересекаются по характеристике $\tilde{x} = x_1 = x_2$. Предлагается дополнить множество $X \cup X'$ характеристикой $\tilde{x}$. Вопрос вычисления значения этой характеристики поручить специальной модели-компоненте, внешними характеристиками которой будут x_1 и x_2, алгоритм ее вычисления по x_1 и x_2 оставляется на усмотрение разработчика. В соотношениях коммутации входящих параметров – это (15) – (17), где присутствуют x_1 и x_2, они должны быть заменены на $\tilde{x}$.

Теперь основной результат данного раздела можно сформулировать в виде следующего предложения.

Предложение 3.3.1

Пусть имеются n математических объектов семейства родов структур $\Sigma_{N_1},\ldots,\Sigma_{N_n}$ «модель-компонента» с базисными множествами

$$X_1, M_1, E_1, \{M_{1,j}\}_{j=1}^{N_1}, \{E_{1,j}\}_{j=1}^{N_1}; \ldots; X_n, M_n, E_n, \{M_{n,j}\}_{j=1}^{N_n}, \{E_{n,j}\}_{j=1}^{N_n}.$$

Тогда возможно распространение рода структуры $\Sigma_{N_1+\ldots+N_n}$ семейства «модель-компонента» на базисные множества модели-комплекса, составленной из этих моделей-компонент:

$$X_1 \cup \ldots \cup X_n, M_1 \cup \ldots \cup M_n, E_1 \cup \ldots \cup E_n, \{M_j\}_{j=1}^{N_1+\ldots+N_n}, \{E_j\}_{j=1}^{N_1+\ldots+N_n},$$

при этом:

1. В объединение характеристик $X_1 \cup \ldots \cup X_n$ входят все характеристики компонент, т.е., формально различными считаются даже одинаковые в предметной области характеристики разных компонент. Объединения реализаций методов-элементов $M_1 \cup \ldots \cup M_n$ и методов-событий $E_1 \cup \ldots \cup E_n$ понимаются в обычном смысле.
2. Если множества внутренних характеристик моделей-компонент $x_1 \subset X_1, \ldots, x_n \subset X_n$ имеют непустые попарные пересечения, то исходный набор из n моделей-компонент придется пополнить еще некоторым количеством L объектов семейства родов структур «модель-компонента». Первоначальное множество характеристик комплекса также пополняется одной новой характеристикой на каждую дополнительную компоненту.
3. Если попарно пересекаются внутренние и внешние характеристики компонент: $x_i \subset X_i$, $a_j \subset X_j, i \neq j$ и $x_i \cap a_j \neq$ Ø, то эти характеристики коммутируются, при этом коммутируемые внешние характеристики компонент исключаются из набора внешних характеристик комплекса, так как становятся его внутренними характеристиками.
4. Если попарно пересекаются внешние характеристики моделей-компонент – они либо отождествляются, либо, как и в случае с внутренними характеристиками, неоднозначность разрешается путем добавления в комплекс некоторого количества дополнительных компонент и характеристик. ■

Таким образом, модели-компоненты допускают объединение в комплексы с возможной (но не всегда обязательной) коммутацией некоторых внутренних характеристик некоторых компонент с некоторыми внешними характеристиками других, возможным дополнением первоначального набора компонент некоторыми новыми и коррекцией части соответствий в соотношениях (5) – (7) для некоторых компонент. Как было показано, полученная таким образом модель-комплекс обладает структурой рода «модель-компонента» (быть может, после добавления в комплекс еще некоторого количества компонент, разрешающих возникшие при объединении базисных множеств неоднозначности), и, стало быть, в свою очередь может входить в новые модели-комплексы.

Этим способом можно строить сколь угодно сложные фрактальные конструкции комплексов, состоящих из компонент, которые сами затем становятся компонентами комплексов. При этом тем не менее все вычислительные процессы конструкций произвольной сложности, полностью определяются аксиомой поведения R_{11} рода структуры «модель-компонента».

4. Модельно-ориентированное программирование

> Поведение системы – это ее способность давать стандартные ответы на стандартные запросы внутренней и внешней среды.

Построение имитационных моделей сложных многокомпонентных систем невозможно без компьютера. При этом компьютерная реализация имитационной модели, скорее всего, является достаточно большой программной системой, в силу сложности ее предметной области. Сказанное поднимает проблему программной реализации имитационных моделей. Основным средством реализации больших программных систем является в последние десятилетия объектно-ориентированный подход, воплощенный в ряде популярных языков программирования высокого уровня.

4.1. Декларативное и императивное программирование

Философ времен окончательного упадка Римской империи А. Боэций сказал: «Гораздо важнее знать, что делается, чем делать то, что знаешь». Можно соглашаться или спорить с ним на предмет что же важнее. Однако самым существенным здесь нам кажется указание на два вида знания:

1. Про некоторые вещи мы точно (конечно, в пределах точности наших моделей) знаем, каковы они. Например, свое имя, паспортные данные и банковские реквизиты. Или каковы характеристики и способы взаимодействия «атомов» корпускулярной модели. Или какова должна быть страничка Web-сайта, которую мы описываем на языке HTML.

2. Про некоторые вещи мы не знаем, каковы они, а знаем (или только считаем, что знаем) лишь, как об этом узнавать. Например, каково решение достаточно сложной системы дифференциальных уравнений — существуют различные алгоритмы решения таких систем, для некоторых из них известны строгие обоснования их применимости в определенных случаях. Или, например, как работает и на что способна предложенная Р. Рейганом СОИ – можно построить ее имитационную модель и поиграть на ней в звездные войны. Здесь проблема с обоснованием применимости модели, по-видимому, будет сложнее.

Следует отметить, что граница между этими видами знаний достаточно подвижна. Например, легко можно забыть свои паспортные данные и банковские реквизиты. Однако если вместе с ними не утрачены основные навыки бытовой культуры, то известно, какие стандартные действия нужно предпринять, чтобы восстановить эти знания. С другой стороны, многие скажут, что знают, что такое π, или $\sqrt{2}$, хотя в очень строгом смысле, мы всего лишь знаем, как вычислять эти числа с некоторой (вовсе не какой угодно!) степенью точности. Аналогично для специалистов в области вычислительной математики численное решение некоторых классов систем дифференциальных уравнений, особенно если под рукой имеются соответствующие инструментальные средства, является почти такой же рутиной, как вычисление $\sqrt{2}$, и в этом смысле можно сказать, что такие специалисты «знают» решения упомянутых систем уравнений.

С приведенными выше двумя типами знания достаточно естественно связываются две известные и очень важные для нас парадигмы программирования – декларативная и императивная.

Вообще-то различных парадигм программирования гораздо больше. Например, в википедии их можно насчитать около трех десятков. В данной работе будут упомянуты лишь те из них, которые имеют отношение к исследуемой теме – построению корпускулярной индуктивной модели сложной системы.

Под парадигмой программирования, вслед за Памелой Зейв (Pamela Zave), автор склонен понимать набор представлений о некотором классе программных систем, допускающих реализацию с помощью этой парадигмы «way of thinking about computer systems» [7]. Таким образом, парадигма не есть язык или система программирования, а некий порядок в голове программиста, позволяющий ему даже на ассемблере писать структурно, или объектно, или функционально, или же в рамках какой-то еще парадигмы. Тем не менее многие современные языки программирования ориентированы на ту или иную парадигму программирования, и, следовательно, предоставляют специальные средства и удобства для ее реализации. Например, C++, Java, C# реализуют ООП (разновидность императивного программирования); LISP, F#, РЕФАЛ – функциональное программирование (разновидность декларативного); PROLOG – логическое.

При декларативном программировании описывается то, каким должен быть известный заранее желаемый результат, например статическая страничка HTML или документ в системе LaTeX, или образ DVD-диска со всей сложной системой его меню и видеоматериалов в системе авторинга Scenarist. При этом выбор последователь-

ности действий, приводящей к описанному в системе декларативного программирования результату, как правило, оставляется на усмотрение соответствующего компилятора или интерпретатора.

Под императивным программированием будем понимать самый распространенный подход к написанию программ, где программа есть последовательность инструкций-приказов, которые должен выполнить компьютер. Таким образом, применяя императивное программирование, мы описываем последовательность действий, достаточную (на наш взгляд) для получения результата проекта. Результат при этом не предполагается известным заранее, наоборот, скорее всего, целью проекта императивного программирования как раз и является получение этого результата.

Мы видим, что декларативная парадигма программирования достаточно естественна для описания наших знаний о системе первого типа, т.е. того, что мы знаем наверняка. Императивное программирование хорошо для описания знаний второго типа – алгоритмов узнавания того, что мы не знаем непосредственно, но хотим и умеем (считаем, что умеем) узнавать.

Заметим, что как декларативное, так и императивное программирование дают достаточно простора для ошибок. Об этом знает всякий, кому приходилось программировать, хотя бы даже на HTML. Однако отлаживать декларативную программу гораздо легче – очень сильно помогает знание о том, как все должно быть. Императивные программы отлаживать очень трудно. Результат работы программы не известен заранее. В случае достаточно сложной программы, даже правильно работающей, этот результат может оказаться весьма неожиданным для исследователя (например, в оптимальном

управлении вместо обывательских представлений о плавном восхождении от хорошего к еще лучшему могут возникать скользящие режимы управления, весьма далекие от подобных представлений). Так что если даже императивная программа не вылетела и не зациклилась, а благополучно что-то сосчитала – всегда остается сомнение: полученный ею результат действительно верен, или же программа где-то врет? Для решения этой проблемы разрабатываются специальные системы тестирования, иногда сопоставимые по затратам с разработкой основной программы. Заметим еще что предполагаемое «знание о том, как узнавать» в императивных программах может оказаться ложным – здесь также достаточно обширное поле для разного рода иллюзий и ошибок.

Скажем еще несколько слов о функциональном программировании – разновидности декларативного. Здесь процесс вычисления трактуется как вычисление значений функций в математическом понимании последних. Математическая функция отличается от функции-подпрограммы императивного программирования тем, что осуществляет отображение своей области определения в область допустимых значений. Функция-подпрограмма, вообще говоря, устроена сложнее. Она также может оказаться отображением множества своих входящих параметров в множество возвращаемых, однако вполне может и не оказаться. Например, у нее могут существовать различные внутренние состояния, связанные со статическими переменными или со значениями записей базы данных, с которой она работает. И в зависимости от этих состояний, функция-подпрограмма при одном и том же входе может выдавать, вообще говоря, различные выходы. Функциональную парадигму легко осуществить средствами большинства современных уни-

версальных языков программирования, ограничившись константами вместо переменных и следя за тем, чтобы функции-подпрограммы были отображениями в математическом смысле. Функциональные программы хороши тем, что по сравнению с императивными легче отлаживаются и тестируются: никаких побочных эффектов, никаких состояний – выход зависит лишь от входа, ошибка достаточно легко локализуется.

Из всего сказанного в этом разделе следует вывод: если что-то в системе допускает декларативное описание – надо это ценить и стараться всячески использовать, отладка будет гораздо легче. Аналогично, если есть возможность сократить императивное программирование, ни в коем случае нельзя ее упускать – это существенно упростит последующую жизнь. Если есть возможность написать подпрограмму в функциональной парадигме, обязательно нужно этим воспользоваться – во время отладки и тестирования окупится сторицей.

Почему так трудно программировать модель сложной системы?

Потому что она и в самом деле очень сложная!

Попробуем перечислить основные сложности:

1. Сложная структура системы и сложные связи между ее компонентами;
2. Сложная организация данных;
3. Сложное поведение компонент системы:
   а) сложная логика поведения;
   б) сложная функциональность действий.

Здесь мы выделили несколько типов сложности, а в модели сложной системы они еще все тесно переплетены между собой. Психологи говорят, что средний человек уверенно способен оперировать не более чем 4 – 6 объектами одновременно. Если объектов становится больше,

человек начинает ошибаться. Быть может, природная одаренность в данной области или же опыт и тренировка способны повысить этот порог с 4 – 6 даже до 15 – 20 объектов. Беда только в том, что в сложных системах приходится оперировать с сотнями и тысячами объектов.

Особенно сложным является императивное программирование, о чем уже говорилось выше. Кроме упомянутых сложностей выбора алгоритма, действительно приводящего к желаемому результату, отладки и тестирования – именно императивные программы обычно сочетают в себе сложную логику со сложной функциональностью и сложной организацией данных.

Возникает вопрос, что же тогда делать, как найти технологию программирования, которая поможет преодолеть перечисленные сложности? На самом деле, такие технологии существуют и достаточно давно применяются в области автоматизации проектирования.

4.2. Декомпозиция и инкапсуляция

Проблема синтеза сложных систем из заданных компонент достаточно успешно решается в различных системах автоматизации проектирования (САПР). Основная идея САПР – модульность разработки. Система проектируется не из самых мелких своих составляющих, а из модулей (блоков) самого высокого уровня, которые могут состоять в свою очередь из модулей более низкого уровня, и, наконец, модули самого низкого уровня состоят из самых мелких «атомов» системы. Как только тот или иной модуль отлажен – про его внутреннее устройство можно забыть. Для работы с ним на более высоком уровне важны лишь его интерфейс и функциональность. Такой подход позволяет создавать процессоры, содержащие десятки миллионов транзисторов, что было бы

немыслимо при проектировании их на уровне отдельных транзисторов.

Заметим, что, по крайней мере, на первый взгляд, эта задача хорошо укладывается в императивную парадигму программирования. В самом деле, отталкиваясь от имеющихся знаний об отдельных компонентах сложной системы, мы беремся построить синтез всей системы, основанный на воспроизведении известного нам поведения каждой ее компоненты, и при этом собираемся наблюдать и изучать поведение системы в целом, не известное нам заранее. В основе синтеза многокомпонентной системы лежит идея, высказанная еще в работах Н.П. Бусленко: дать каждой из компонент, про которые нам все известно, максимально проявить себя, учитывая при этом и все межкомпонентные связи [14].

Основным инструментом реализации проектов императивного программирования являются объектно-ориентированные языки программирования. В свое время эти языки возникли в значительной мере как ответ на запросы исследователей, занимавшихся имитационным моделированием. Одним из самых первых объектно-ориентированных языков можно считать Симулу-67. Структура сложной системы, состоящей из многих экземпляров различных типов компонент хорошо описывается иерархией классов объектно-ориентированного программирования. У ООП масса известных всем достоинств, которые и сделали этот подход бесспорным лидером в программировании последних 30 лет. Однако применительно к нашей задаче есть у ООП и существенные недостатки.

Главный из них (конечно, применительно к классу задач, о которых идет речь в данной работе) – отсутствие у объекта поведения. У объекта есть характеристики,

есть масса всяческих умений – это его методы – а поведения, в смысле умения давать ответы на стандартные запросы окружающей среды, у него нет. Конечно, ООП в соединении с соответствующим программным обеспечением промежуточного уровня, позволяет объектам даже в распределенных системах взаимодействовать между собою, наблюдать чужие характеристики, пользоваться чужими методами, однако все это делается в некотором смысле «вручную».

В этом смысле коллектив программистов, приступающий к созданию сложного проекта, и вооруженный набором библиотек самых полезных классов, похож на человека, которому выдали несколько компьютеров, каждый из которых оснащен большим набором полезных прикладных программ, однако из системных программ имеющий только загрузчик, и поставили задачу сделать из этого распределенную систему. Понятно, что современный программист возмутился бы страшно: «Так не бывает! Где моя сетевая операционная система? Где, наконец, ПО промежуточного уровня?». Однако при создании сложной системы средствами ООП дело обстоит именно так – к сложности содержательных вычислений предметной области добавляется еще и сложность организации поведения объектов а также их связей и взаимодействий между собою.

4.3. Поведение математических моделей сложных систем

Что же такое поведение сложной системы?

На наш взгляд, поведение сложной системы есть функциональный аналог операционной системы в области системного программного обеспечения компьютера, и функциональный аналог бытовой культуры в социаль-

ных системах – способность давать стандартные ответы на стандартные запросы внутренней и окружающей среды. В имитационном моделировании сложных систем очень важно уметь моделировать поведение компонент. Как уже говорилось выше, именно из их поведения естественно синтезировать поведение системы в целом.

В истории развития информатики известно несколько подходов к объектному программированию, где объекты обладают поведением. Подходы эти зародились в среде исследователей, занимавшихся проблемами искусственного интеллекта, и известны как модель акторов [3] и агентное программирование [5, 6]. Однако хотя автор согласен с главным посылом этих подходов – важностью моделирования поведения агентов системы, детали описания этого поведения в [3, 5, 6], на наш взгляд, слишком окрашены спецификой проблематики искусственного интеллекта, т. е. недостаточно универсальны. Так, у И. Шохема [5, 6] поведение – это поведение ментальное, о состоянии которого рассуждают в категориях «убеждения», «обязанности», «способности» и т.д. Акторная модель Хьюитта [3] интересна своей ориентированностью на параллельные вычисления, однако асинхронный обмен акторов сигналами-сообщенями, а также лавинообразное порождение новых акторов представляются нам избыточными конструкциями, затрудняющими реализацию подобных систем. Подробно этот вопрос обсуждался в [8].

Проблема здесь в том, что, согласившись на наличие поведения у компонент системы, нужно учиться описывать и реализовывать это поведение. Это поведение, с одной стороны достаточно сложно, так как это поведение элементарной сложной системы. Заранее оно не может быть известно, поскольку может зависеть от внут-

ренних и внешних по отношению к рассматриваемой компоненте условий, на которые она должна уметь реагировать должным образом. Следовательно, такие поведенческие моменты должны вычисляться и, казалось бы, алгоритмы их вычисления должны быть описаны на императивных языках программирования.

С другой стороны, кое-что о поведении компоненты мы всегда знаем заранее, так как она в силу исходных предположений работы имеет в предметной области моделирования достаточно хорошо изученный прообраз. Поэтому обычно мы заранее можем перечислить весь набор ее «умений» – элементарных действий, а также сказать, какое из этих элементарных действий может перейти в какое и под действием каких обстоятельств – просто уж так устроен прообраз этой компоненты в предметной области моделирования.

Вот это-то знание, коррелирующее с пониманием устройства в предметной области как прообраза отдельной компоненты, так и всей многокомпонентной модели, как раз наличествует заранее еще до построения модели и не зависит от хода имитационных экспериментов. Поэтому такое знание об устройстве модели сложной системы вполне может быть выражено на декларативном языке программирования.

Такой подход позволяет получить «ортогональные» (подобным образом «ортогональны», например, описания стиля и наполнения в Word- или HTML-документах) описания модели сложной системы: на декларативном языке описывается то, как устроена система, и правила поведения ее составляющих. На императивном языке описываются отдельные законченные элементарные алгоритмы, не взаимодействующие ни с чем в модели, кроме собственных входных и выходных параметров, и, ста-

ло быть, вполне укладывающиеся в парадигму функционального программирования.

Предлагаемое разделение описаний на декларативные и функциональные оказывается весьма технологичным: декларативные и функциональные части можно отлаживать независимо друг от друга. При этом самая сложная составляющая программы, которая пишется на обычных языках программирования, распадается на ряд независимых друг от друга законченных элементарных алгоритмов. При этом она редуцируется настолько, что чаще всего возможности ООП оказываются слабо востребованными: алгоритмически цельную программу с фиксированным набором входных и выходных параметров чаще всего нетрудно реализовать в функциональной парадигме. И это существенно облегчает отладку системы. Вся объектная ориентированность, идущая от предметной области моделирования, оказывается в декларативном описании модели.

Одной из первых сред программирования, воплотивших описанную выше концепцию разделения описания сложной системы на декларативную и функциональную составляющие, была разработанная в конце 80-х в ВЦ АН СССР инструментальная система имитационного моделирования MISS (Multilingual Instrumental Simulation System). Концепция программирования, лежащая в ее основе была полностью описана на уровне руководства пользователя в работе [10]. В системе MISS была реализована в среде MS-DOS система программирования на специальном декларативном языке описания сложных систем, интегрированная с базой данных, системой поддержки выполнения модели и системой презентации результатов моделирования. В качестве императивных языков программирования, на которых писались вычис-

лительные алгоритмы, были разрешены две версии языка MODULA-2, а также языки C и C++ в Борландовской реализации. Следует заметить, что до этого моделирующее сообщество создавало исключительно языки моделирования императивного типа.

Впоследствии идея разделения описания сложной системы на декларативную и императивную части применялась неоднократно. Так в спецификации архитектуры верхнего уровня (High Level Architecture – HLA) [4] – средстве создания сложных распределенных моделей – устройство системы, ее компонент и связей между ними описывается специальными шаблонами. В коммерчески успешной отечественной инструментальной системе имитационного моделирования AnyLogic [22] в качестве декларативного языка описания сложной системы применяется интерактивная графическая система, где на экране размещаются пиктограммы компонент будущей системы и тут же проводятся связи между ними. В качестве императивного языка программирования используется родной для системы AnyLogic язык Java. Наконец, появился и даже стал международным стандартом язык моделирования объектных систем UML [16], однако автору он не симпатичен в силу своей громоздкости и порой за счет этого неоднозначности. В качестве результатов компиляции транслятор UML может выдавать заготовки модулей для императивных языков.

В настоящее время в ВЦ РАН продолжается развитие концепции разделения описания сложной системы на декларативную и функциональную составляющие. Функционирует макет инструментальной системы распределенного моделирования [8], основанный на этой концепции.

4.4. Модельно-ориентированная парадигма

Исторически смена парадигм программирования сопровождалась укрупнением, агрегированием основного инструмента деятельности программиста.

Начиналось все с машинной команды, затем, с появлением языков программирования высокого уровня – таким инструментом стал оператор языка, реализующий некое законченное действие, возможно, с помощью нескольких машинных команд.

С победой идей структурного программирования – на смену отдельным операторам и переменным пришли стандартные конструкции типа «цикл», «ветвление», подпрограммы-функции и структуры данных.

С появлением объектного анализа основной единицей конструирования стал объект, объединяющий некую структуру данных с набором необходимых для их обработки методов. Кроме того, с помощью механизма наследования можно строить иерархии классов объектов, развивающих, конкретизирующих и воплощающих некоторый набор базовых идей, заложенных в корневых классах такой иерархии. Данная парадигма программирования в настоящее время является господствующей и ее базовые понятия, такие как класс, объект, типизация, наследование, инкапсуляция, полиморфизм реализованы с некоторыми нюансами в большинстве современных императивных языков программирования, таких как C++, Java, C#, Delphi и многих других.

Модельно-ориентированная парадигма программирования предлагает снова увеличить степень агрегирования основной единицы программирования. Предлагается конструировать программную систему из моделей-компонент – сущностей, помимо характеристик и отдельных умений, обладающих собственным поведением,

т.е. способных во всех ситуациях, которые могут им встретиться, давать стандартные для них ответы на стандартные запросы внутренней и внешней среды. Формально этот факт следует из уже упоминавшейся выше гипотезы о замкнутости модели.

Воспользовавшись описанной выше инвариантностью организации вычислительного процесса имитационной модели (аксиома поведения R_{11}) относительно операции объединения моделей-компонент в модель-комплекс, можно предложить новый подход к программированию, во-первых, имитационных моделей сложных систем, и, во-вторых, сложных программных систем, быть может, не имеющих отношения к имитационному моделированию, но имеющих выраженную многокомпонентную организацию, где уместен синтез целого из составляющих его частей, природа которых, а также способы взаимодействия между собой, известны заранее настолько, что для них выполнены требования второй главы этой работы.

В отличие от объекта объектного анализа, методы модели-компоненты не нужно (да и невозможно) вызывать извне. Не нужно заботиться и об организации функционирования модели-компоненты. Модель-компонента функционирует всегда (как всегда функционирует, например, операционная система компьютера), в соответствии с аксиомой R_{11} рода структуры «модель-компонента», и всегда готова отреагировать заложенным в ее конструкцию способом на происходящие внутри и вне ее изменения, на которые способны реагировать ее методы-события. Поэтому, про внутреннее устройство однажды отлаженной модели-компоненты можно полностью забыть, используя ее в дальнейших конструкциях как готовый функциональный блок – просто нужно пра-

вильно коммутировать входы и выходы – и все будет работать. Все происходит примерно так же, как при включении готовой микросхемы в микросборку и размещении затем этой микросборки на печатной плате. Если при этом на каждом уровне проекта коммутация разумна – микросхема будет правильно функционировать в составе гораздо более сложного электронного устройства.

Описания моделей-компонент, как математических объектов соответствующего рода структуры, декларативны. Также декларативны описания объединения моделей-компонент в модели-комплексы. Для таких описаний предлагается специальный декларативный язык [12, 8] ЯОКК (язык описания компонент и комплексов). Значительный удельный вес в этом языке имеют операторы коммутации.

Декларативные описания модели-компоненты на ЯОКК, эквивалентные соотношениям типизации (1)-(10), вполне определяют ее устройство и логику функционирования, при этом полностью абстрагируясь от содержательной стороны, как действий модели-компоненты, осуществляемых ее методами-элементами, так и причин этих действий, выявляемых методами-событиями.

В силу все той же гипотезы о замкнутости, все методы модели-компоненты, как быстрые и медленные методы-элементы, так и методы-события, вычисляют некоторые функции (в математическом, а не программистском понимании термина «функция») от некоторых подмножеств внутренних и внешних характеристик модели-компоненты. Следовательно, эти методы вполне могут быть реализованы в функциональной парадигме программирования (что, между прочим, может еще сильнее увеличить степень параллельности получаемого кода). Последнее вовсе не означает призыва к всеобщему пере-

ходу на лямбда-исчисление – функциональная парадигма вполне может быть реализуема и на всеми любимых C, C++, C# и Java.

В результате оказывается, что императивное программирование – камень, о который больше всего спотыкаются как при разработке, так и при отладке сложных программных систем – в модельно-ориентированном программировании не применяется вовсе. Кроме того, реализация даже весьма сложной программной системы методом модельно-ориентированного программирования декомпозируется на ряд вполне обозримых декларативных описаний моделей-компонент и составляемых из них моделей-комплексов и не зависящих от этих описаний и друг от друга функциональных программ, допускающих независимое написание и отладку. Подобная декомпозиция весьма технологична при коллективной разработке большой программной системы.

Из аксиомы поведения R_{11} рода структуры «модель-компонента», определяющей вычислительный процесс функционирования любой модели, следует, что чем больше у модели-компоненты процессов, тем большее число методов можно запускать на выполнение параллельно. Поскольку при объединении моделей-компонент в модель-комплекс число процессов полученного комплекса есть сумма процессов его компонент, то можно заключить, что при усложнении модели количество методов, допускающих параллельное выполнение, также увеличивается. Еще более увеличиться оно может от реализации методов в функциональной парадигме программирования. Все это позволяет надеяться на плодотворное применение методов модельно-ориентированного программирования на высокопроизводительных вычислительных системах.

4.5. Модельно-ориентированный декларативный язык описания компонент и комплексов (ЯОКК)

В данном разделе разобран специализированный декларативный язык, на котором должны составляться описания классов компонент и комплексов модели. По поводу языка описаний стоит сделать несколько замечаний.

Основная задача рассматриваемого языка ЯОКК – описать род структуры «модель-компонента», введенный в разд. 3.2, и некоторые связанные с ним понятия, например, организацию внутренних и внешних характеристик модели и их коммутацию. Также с помощью этого языка описываются модели-комплексы. Их описания затем можно откомпилировать в соответствующие описания моделей-компонент.

Рассматриваемый язык в полной мере выражает предлагаемую в данной работе концепцию имитационного моделирования сложных многокомпонентных систем. В концепции моделирования существуют определенные понятия, например комплекс, компонента – в языке описаний им соответствуют одноименные описатели. В концепции моделирования имеются понятия коммутации компонент в комплексе или коммутации элементов в компоненте – в языке ЯОКК им соответствуют описатели коммутации и т. д.

Возможно, в дальнейшем над языком ЯОКК будет создана надстройка в виде графического интерфейса пользователя (GUI), где мышкой на рабочее поле вытаскивались бы нужные сущности в виде соответствующих пиктограмм и мышкой проводились бы необходимые связи между ними. Такая надстройка переводила бы графические манипуляции с пиктограммами в соответству-

ющие конструкции ЯОКК, – можно считать, что это задача следующего этапа реализации системы.

Потребность в непроцедурных языках, на которых описывается не что нужно выполнить, а то, кто и как устроен и как и с кем связан внутри и снаружи, возникла достаточно давно. Ниже перечислены несколько языков подобной направленности с указанием их разработчиков и иногда систем, где они применялись:

- SQL (IBM, ANSI) — 1986 г.
- Язык MISS (ВЦ АН СССР) — 1986-1990 гг.
- Язык IDL (CORBA, OMG) — 1991 г.
- Язык Slice (ICE, ZeroC) — 2003 г.
- XML (W3C, SOAP, Microsoft) — 1996-2005 гг.
- Язык UML (OMG, UML Partners) — 1997-2005 гг.

По функциональности выразительных средств, для наших целей подошли бы кроме второго, пожалуй, два последних. Однако из них UML, несмотря на статус международного стандарта, слишком избыточен для заявленных в данной работе целей, а потому был бы неоправданно сложен в реализации. Язык XML достаточно гибок, чтобы можно было организовать компиляцию необходимого для данного проекта его подмножества. Однако в силу специфики его синтаксиса, описания на нем, на взгляд автора, были бы недостаточно наглядны для целей обучения. Поэтому была выбрана модификация проверенного, хотя и не получившего широкого распространения решения, реализованного ранее автором совместно с В.Ю. Лебедевым [10], – измененный в сторону упрощения (так как с тех пор заметно упростилась концепция моделирования) язык MISS, который в данной системе моделирования получил и новое название – язык описания комплексов и компонент (ЯОКК).

4.5.1. Пример использования ЯОКК – пешеходы и муха

Определения конструкций ЯОКК мы будем параллельно иллюстрировать примерами описания на этом языке одной учебно-тестовой модели – уже упоминавшейся выше модели про пешеходов и муху. В связи с этим сначала опишем на неформальном уровне реализованный вариант этой модели.

Модель «Пешеходы и муха» или вариант этой модели, известный как «муха фон Неймана», – простейшая нелапласовская модель. Она уже упоминалась во второй главе. В ее основе лежит задача, встречающаяся в курсе математики начальной школы. Два пешехода идут навстречу друг другу с постоянной скоростью. Между ними летает муха с постоянной по абсолютной величине скоростью, большей, чем скорость любого из пешеходов. Как только муха долетает до одного из пешеходов, она тут же разворачивается и летит к другому. И в школьной задаче и в легенде про то, как фон Нейман ее решал, спрашивается, какое расстояние пролетит муха за время от начала движения до момента встречи пешеходов.

С точки зрения имитационного моделирования сложных систем, модель интересна тем, что при крайней простоте алгоритмов компонент она предъявляет достаточно серьезные требования к организации вычислительного процесса, например приходится моделировать с переменным шагом модельного времени. Кроме того, эта задача не является лапласовской: момент встречи пешеходов является точкой накопления системных событий, а скорость мухи в этот момент невозможно обусловить всей предшествовавшей историей модели. Если имеется желание продолжить моделирование на время после встречи пешеходов – необходимо задать мухе в точке встречи совершенно новую скорость, никак не вытекаю-

щую из всего того, что было до встречи. Связано это с тем, что скорость мухи разрывна и в точке встречи пешеходов имеет два равноправных предельных значения, – по непрерывности в точку встречи ее продолжить невозможно.

Модель была реализована как тестовый пример, необходимый для отладки программного обеспечения рабочей станции распределенного моделирования, а также как учебный пример, иллюстрирующий правила создания моделей и их компонент в предлагаемой среде распределенной имитации.

Модель «пешеходы и муха» представляет собой модель-комплекс, состоящий из двух экземпляров модели-компоненты «пешеход» и одного экземпляра модели-компоненты «муха». Компонента «пешеход» устроена совсем просто: она реализует единственный процесс, состоящий из единственного метода-элемента «движение». «Движение» – медленный элемент, по текущим значениям внутренней характеристики X – координаты пешехода и внешней характеристики V – его скорости, а также по величине текущего шага модельного времени Δt, этот метод вычисляет значение внутренней характеристики X в конце шага модельного времени по формуле $X(t+\Delta t) = X(t) + V \cdot \Delta t$. Никаких других методов, а следовательно, и переходов, и связанных с ними событий единственный процесс модели-компоненты «пешеход» не имеет.

Модель-компонента «муха» устроена сложнее. Она также участвует в единственном процессе, но этот процесс состоит из двух элементов:

- «Движение» – тот же самый алгоритм, что и у пешехода, поэтому может быть реализован тем же самым методом.

- «Разворот» – у компоненты "муха" скорость является внутренней характеристикой, а не параметром, как у "пешехода". «Разворот» – быстрый метод, он не занимает модельного времени, а действие его состоит в том, что скорость мухи меняет знак: из V становится $-V$.

Элемент «разворот» всегда переходит в элемент «движение». Элемент «движение» переходит в элемент «разворот», по наступлению события «долет до пешехода». Таким образом, с компонентой «муха» связано событие «долет до пешехода». Алгоритм его вычисления – наиболее сложный в данной модели. На входе он получает координаты и скорости всех компонент модели, на выходе же дает время до ближайшей встречи с пешеходом. Для этого сначала определяется, к какому из пешеходов летит муха (знаки скоростей у мухи и пешехода должны быть различны), затем расстояние между ними делится на сумму абсолютных величин скоростей. Если расстояние нулевое – муха уже долетела и соответственно событие уже наступило.

Предметом распределения вычислений в этой модели являются методы и событие. Каждый из них может находиться на отдельном компьютере в сети при наличии соответствующего сервиса, обеспечиваемого рабочей станцией распределенного моделирования. (Например, они предоставляются станцией, расположенной на хосте simul.ccas.ru.) Модель спроектирована так, что все методы и события одного шага моделирования вызываются асинхронно.

В дальнейшем именно эту модель в описанном выше виде мы будем использовать для иллюстрации тех или иных конструкций ЯОКК.

4.5.2. Общий синтаксис ЯОКК

Отметим, что хотя аккуратнее было бы употреблять термины «описание класса моделей-компонент» или «описание класса моделей-комплексов», мы ради краткости будем пользоваться словосочетанием «описание компоненты» и «описание комплекса». Лингвистические формулы записываются ниже в стандартной нотации, согласно которой служебные слова выделяются жирным шрифтом, разделители – кавычками, а необязательные включения – квадратными или фигурными скобками, причем фигурные скобки указывают на возможность повторения.

Собирая модель, инструментальная система автоматически генерирует ее базу данных, руководствуясь при этом содержимым специальных описателей ЯОКК. Таких описателей бывает четыре типа:

1. Описатель типа данных.
2. Описатель комплекса.
3. Описатель метода.
4. Описатель компоненты.

Все описатели, кроме описателя типа данных, состоят из заголовка и нескольких параграфов. Описатель типа данных представляет собой единственный параграф. Описатель типа данных – необязательная конструкция, в некоторых случаях она удобна, но, вообще говоря, можно обойтись и без нее.

Параграфы начинаются заголовком, после которого через точку с запятой идут операторы. Если параграф не последний – он заканчивается началом заголовка следующего параграфа, если последний – ключевым словом **END;** которое заканчивает и весь описатель.

Порядок параграфов в описателе свободный, если между ними нет зависимости. Если такая зависимость

есть – зависящие параграфы должны появляться в тексте описателя позже тех, от которых зависят.

В описателях допустимы комментарии C++ подобного синтаксиса, т.е. игнорируются строки, начинающиеся с «//», и любой текст, находящийся между символами «/*» и «*/».

Попробуем формально определить основные синтаксические единицы ЯОКК.

Начнем с метасимволов, которые сами не входят в ЯОКК, но используются в синтаксических формулах, его описывающих.

| – «или» – знак альтернативы. Не входит в число символов ЯОКК. Через этот знак будут перечисляться возможные альтернативы той или иной синтаксической конструкции.

::= – «это есть» – определение синтаксической единицы, стоящей слева от знака через синтаксическую конструкцию, стоящую справа от него. Используется только в метаформулах.

«» – кавычки. Сами кавычки не входят в число символов ЯОКК. В метаформулах в кавычки будут заключены разделители ЯОКК, в основном с целью привлечения к ним внимания. Например, «» или « » означают пробел, «,» – запятая.

[] – квадратные скобки. Не входят в число символов ЯОКК. В метаформулах в квадратные скобки заключается конструкция возможная, но не обязательная в данном месте.

{} – фигурные скобки. Не входят в число символов ЯОКК. В метаформулах в фигурные скобки заключается конструкция, которая может быть повторена в данном месте любое конечное число раз, в том числе и ни одного раза.

<> – угловые скобки. Не входят в число символов ЯОКК. В метаформулах в угловые скобки заключается название той или иной синтаксической единицы.

Определим теперь символы и общие синтаксические конструкции ЯОКК.

<символ ЯОКК> ::= <буква> | <цифра> |
| <разделитель>

<буква> ::= a | … | z | A | … | Z | а | … | я | А | … | Я | _

<цифра> ::= 0 | <значащая цифра>

<значащая цифра> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<разделитель> ::= « » | «.» | «,» | «;» | «:» | «(» | «)» |
| «=» | <ключевое слово>

<ключевое слово> ::= **END;** | **TYPE** | **int** | **double** |
long	**char**	**byte**	**bool**	**uint**	**short**	**ulong**	**decimal**
float	**address**	**sbyte**	**ushort**	**COMPLEX**
COMPONENTS	**COMMUTATION**	**SAMENESS**
METHOD	**FAST**	**SLOW**	**EVENT**	**ADDRESS**	**local**
INPUT	**OUTPUT**	**COMPONENT**	**PHASE**
PARAMETERS	**phase**	**param**	**ELEMENTS**
EVENTS	**SWITCHES**

<идентификатор> ::= <буква>
[{<буква> | <цифра>}]

При этом идентификатор не может совпадать ни с одним ключевым словом.

< целое без знака > ::=
::= 0 | <значащая цифра>[{<цифра>}]

Синтаксические единицы, названия которых начинаются с <имя …> – всегда идентификаторы.

Перейдем теперь к синтаксису описателей.

4.5.3. Описатель типа данных

Заголовок описателя типа данных начинается ключевым словом **TYPE**, за которым следует идентификатор – имя типа. Заканчивается заголовок – «;» – точкой с запятой. После заголовка разделяемые символами «;» – точкой с запятой идут операторы описания данных.

Описания данных в ЯОКК базируются на встроенных типах данных, совпадающих с аналогами в языках семейства С. Набор встроенных типов может меняться, в зависимости от платформы.

Синтаксис оператора описания данных следующий:

<идентификатор типа> « » <описание данных>
{«,»<описание данных >}«;»

Идентификатор типа отделяется пробелом от одного или нескольких разделенных запятой описаний данных, заканчивается оператор точкой с запятой. Здесь идентификатор типа – либо одно из встроенных имен типов, приведенных в таблице ниже, либо идентификатор типа, чей описатель типа данных расположен выше данного в охватывающем описателе, либо это каталогизированный тип данных. Каталогизированным тип данных становится после успешной компиляции его описателя.

Описание данных – это либо идентификатор, либо массив. Массив – это идентификатор, в квадратных скобках справа от которого положительными целыми числами через запятую указаны размерности массива.

Встроенные типы определяются приведенной ниже таблицей:

Встроенные типы ЯОКК

Тип в ЯОКК	Тип в языке C#	Тип в .Net	Длина в байтах
int	int	System.Int32	4
double	double	System.Double	8
long	long	System.Int64	8
char	char	System.Char	2
byte	byte	System.Byte	1
bool	bool	System.Boolean	1
uint	uint	System.UInt32	4
short	short	System.Int16	2
ulong	ulong	System.UInt64	8
address	void*	System.Void*	4[1]
decimal	decimal	System.Decimal	16
float	float	System.Single	4
sbyte	sbyte	System.SByte	1
ushort	ushort	System.UInt16	2

Пример описателя типа данных:

```
TYPE test;
    double X, Y(2,3,4), Z;
    bool a, b(4);
    int i;
END;
```

Тогда если где-то описано поле vrbl:

test vrbl;

то имеет смысл поле vrbl.X(0,1,3) типа **double**, а также булевские поля vrbl.a и vrbl.b(2).

[1] Для 32-разрядных систем. Для 64-разрядных – 8.

4.5.4. Описатель комплекса

Описатель комплекса открывается заголовком, который состоит из ключевого слова **COMPLEX**,

За заголовком идут параграфы:

1. Параграф компонент.
2. [Параграф коммутации компонент.]
3. [Параграф отождествления внешних характеристик.]

Обязательный параграф компонент открывается ключевым словом **COMPONENTS**, после которого через запятую идут имена компонент, за которыми в круглых скобках стоит целое число – количество экземпляров компоненты, определяемого ее именем типа, входящих в комплекс. Если число экземпляров – 1, описатель количества экземпляров (1) разрешается опустить. Заканчивается параграф символом «**;**» – точкой с запятой. Для успешной компиляции комплекса необходимо, чтобы описатели входящих в него компонент были откомпилированы ранее.

Необязательный параграф коммутации компонент описывает, какие внутренние характеристики каких компонент вычисляют какие внешние характеристики каких компонент. Параграф открывается ключевым словом **COMMUTATION**, за которым следуют операторы коммутации. Операторы коммутации имеют следующий вид:

<идентификатор компоненты> «(»
<экземпляр компоненты>«)»«.»
[{<имя типа данных> «.»}]
<поле параметра компоненты> «**=**»
<идентификатор компоненты> «(»
<экземпляр компоненты >«)»«.»

[{<имя типа данных> «.»}]
<поле внутренней характеристики
компоненты>«;»

До нужного элемента данных, возможно, придется добираться через несколько квалификаторов, поэтому поля параметра компоненты и поля внутренней характеристики компоненты в операторах коммутации могут иметь вид

{<идентификатор>«.»}<поле>

где поле – либо идентификатор, либо элемент массива.

Необязательный параграф отождествления внешних характеристик компонент открывается ключевым словом **SAMENESS**, за которым следуют операторы отождествления. Операторы отождествления имеют следующий вид:

<идентификатор компоненты> «(»
<экземпляр компоненты>«)»«.»
[{<имя типа данных> «.»}]
<поле параметра компоненты> {«=»
<идентификатор компоненты> «(»
<экземпляр компоненты >«).»
[{<имя типа данных> «.»}]
< поле параметра компоненты >}«;»

Результат компиляции комплекса – текстовый описатель этого комплекса как модели-компоненты.

Пример описания модели-комплекса

Модель-комплекс «пешеходы и муха»:

```
COMPLEX menANDfly;
COMPONENTS
 Fly(1), Man(2);
```

```
COMMUTATION
Fly(0).man0Phase.x=Man(0).x;
Fly(0).man0Phase.v=Man(0).v;
Fly(0).man1Phase.x=Man(1).x;
Fly(0).man1Phase.v=Man(1).v;
END;
```

4.5.5. Описатель метода

Методы – это или элементы процессов, или методы, вычисляющие наступление событий. Это те части модели, которые могут вызываться как локально, так и распределенно. Считается, что методу передается некий набор параметров, тип которого должен быть описан, и некий набор параметров возвращается методом. Так как тип этих наборов параметров, в особенности для удаленных методов, определяется разработчиком метода, а не разработчиком модели, возникает вопрос о коммутации полей параметров метода с полями фазовых характеристик и/или параметров компоненты. Кроме того, методы различаются на события и методы, реализующие элементы процессов, а последние, еще и по отношению к модельному времени на:

- сосредоточенные (быстрые) – происходящие чены для вычисления возможных разрывов первого рода характеристик модели-компоненты в начале шага моделирования;
- распределенные (медленные) – занимающие не менее одного модельного такта и дающие определенный результат своего выполнения в виде изменений внутренних характеристик модели в конце каждого такта. Они предназначены для вычисления характеристик модели-компоненты на интервалах их непрерывности.

Описатель метода начинается заголовком – ключевым словом **METHOD,** за которым следует идентификатор – имя метода, за которым следуют необязательные [«:» <тип метода>], наконец, завершает заголовок «;» – точка с запятой.

Тип метода – одно из ключевых слов **FAST**, **SLOW** или **EVENT,** соответствующее сосредоточенным или распределенным элементам, или событиям. По умолчанию тип метода, реализующего элемент процесса, считается **SLOW,** и в этом случае явное его указание разрешается опустить.

Далее следует необязательный параграф адреса метода. Он имеет вид

ADDRESS «:» <адрес хоста> «;»

Адрес хоста это URL (Uniform Resource Locator) – единый указатель ресурсов или ключевое слово **local.** По умолчанию адрес метода – **local**, в этом случае параграф адреса можно опустить.

Далее следуют параграфы описаний входящих и возвращаемых параметров метода. По синтаксису они отличаются от описателя типа лишь другими ключевыми словам заголовка – **INPUT** и **OUTPUT** соответственно – и тем, что идентификатор типа после ключевых слов необязателен. Если тем не менее он присутствует – каталогизированный тип данных с таким именем появится в базе данных рабочей станции в результате успешной компиляции описателя метода.

Если возвращаемые параметры полностью совпадают с входными – параграф возвращаемых параметров можно опустить. У событий также нет параграфа возвращаемых параметров, так как известно, что они всегда возвращают значение типа **double** – прогнозируемое время до наступления соответствующего события.

Примеры описаний методов-элементов моделей-компонент «пешеход» и «муха» и метода-события компоненты «муха» –

Медленный метод-элемент «движение»:

```
METHOD move;
 // по умолчанию подразумевается SLOW
 ADDRESS: 192.168.137.1;
// где находится реализация
 INPUT
 double x,v;
 // всем медленным по умолчанию всегда передается dt
 // после объявленных, и всем всегда t - в самом конце
OUTPUT
 double x;
 END;
```

Быстрый метод-элемент «разворот»:

```
METHOD Uturn: FAST;
 // по умолчанию ADDRESS: local;
 INPUT
 double v;

/**************
Поскольку про OUTPUT ничего не сказано – по умолчанию он такой же, как и INPUT, т.е.
OUTPUT
 double v;
 ***************/
 END;
```

Метод-событие «долет до пешехода»:

```
METHOD reaching: EVENT;
 ADDRESS: simul.ccas.ru;
 INPUT
 double x,v,m0x,m0v,m1x,m1v;
 // У всех методов-событий OUTPUT всегда – double dt;
 // – прогноз времени его наступления
END;
```

4.5.6. Описатель компоненты

Описатель компоненты начинается заголовком – ключевым словом **COMPONENT,** за которым следует идентификатор – имя компоненты, за которым следует «;» – точка с запятой, завершающая заголовок.

За заголовком следуют параграфы описателя:

1. [Параграфы типов.]
2. Параграф внутренних характеристик.
3. [Параграф параметров.]
4. Параграф элементов.
5. [Параграф событий]
6. Параграф коммутации.
7. [Параграф переключателей.]

Порядок параграфов в описателе несущественен, кроме параграфов типов, которым естественно быть в начале, так как всякий тип должен быть описан до его использования. Каждый параграф начинается соответствующим ему ключевым словом и заканчивается ключевым словом следующего параграфа. Последний параграф заканчивается ключевым словом **END** и «;» – точкой с запятой.

Параграфы типов необязательны, они нужны лишь если определенные в них типы используются в параграфах внутренних характеристик и параметров. Синтаксис их был описан выше в описателе типов.

Синтаксис параграфов внутренних характеристик и параметров (внешних характеристик) такой же, как и параграфов типов – разница лишь в ключевых словах параграфов: **PHASE** и **PARAMETERS** соответственно. Кроме того, идентификатор типа после ключевого слова необязателен. Если он присутствует – тип данных с таким именем появится в базе данных рабочей станции в результате успешной компиляции описателя. Если нет, то чтобы идентифицировать поля характеристик компоненты достаточно либо имени компоненты, либо используются специальные ключевые слова **phase** и **param** (см. далее).

Разбиение характеристик компоненты на внутренние и внешние в описателях компонент сделано исключительно ради более строгого воплощения концепции моделирования. Компиляция переводит описания внутренних и внешних характеристик в единую базу данных характеристик.

Приведем формальное описание параграфа элементов:

< параграф элементов> ::= **ELEMENTS**
{<описатель процесса> «;»}
[**END;**]

<описатель процесса> ::= [<метка процесса> «:»]
<элемент> [{«,» <элемент>}]

<элемент> ::= < имя элемента>
[«:» <имя реализации>]

< имя элемента> ::= <идентификатор>

< имя реализации> ::= <идентификатор>

< метка процесса > ::= <идентификатор> |
| <целое без знака>

Параграф методов-элементов открывается ключевым словом **ELEMENTS** и заканчивается ключевым словом **END;**, если он последний.

На самом деле, этот параграф определяет не только методы-элементы, но и процессы модели-компоненты. В этом параграфе через «**;**» – точку с запятой перечислены процессы компоненты. Каждый процесс в этом перечислении – это метка процесса, не обязательная в простейших случаях, например, когда процесс один. Метка процесса – целое без знака или идентификатор, после которого следует двоеточие. После метки процесса идут перечисленные через запятую имена образующих его элементов с возможным указанием через двоеточие имени реализации метода. Имя реализации должно совпадать с именем, появлявшемся в одном из ранее откомпилированных описателей методов. Если имя реализации явно не указано – реализация будет искаться в базе данных под именем элемента в процессе.

Порядок элементов в их перечислении в составе процесса произволен, за исключением того, что первым в списке идет так называемый начальный элемент, который будет считаться текущим при первом запуске модели на выполнение. Имена элементов должны быть уникальными в пределах описания процесса: нельзя одинаково назвать два разнотипных элемента одного процесса, но использование одного имени элемента в описаниях разных процессов не возбраняется. Приведем пример параграфа элементов:

ELEMENTS
1: Man_0_move: move;
2: Man_1_move: move;
3: Fly_0_move: move, Fly_0_Uturn: Uturn;
END;

Приведем формальное описание параграфа событий:

< параграф событий > ::= **EVENTS**
{<описатель событий процесса> «;»}
[**END;**]

<описатель событий процесса> ::=
[<метка процесса> «:»]
<событие> [{«,» <событие>}]

<событие> ::= <имя события>
[«:» <имя реализации>]

<имя события> ::= <идентификатор>

<имя реализации> ::= <идентификатор>

<метка процесса> ::= <идентификатор> |
| <целое без знака>

Синтаксис параграфа событий очень похож на синтаксис параграфа элементов. В этом параграфе через «;» – точку с запятой перечислены наборы событий процессов компоненты. Каждый набор событий процесса в этом перечислении – это метка процесса, не обязательная в простейших случаях, например, когда процесс один. Метка процесса – целое без знака или идентификатор, после которого следует двоеточие. После метки процесса идут перечисленные через запятую имена событий процесса с возможным указанием через двоеточие имени реализации метода-события. Имя реализации должно совпадать

с именем, появлявшемся в одном из ранее откомпилированных описателей методов. Если имя реализации явно не указано – реализация будет искаться в базе данных под именем события в процессе.

Порядок событий в их перечислении в составе процесса произволен. Имена событий должны быть уникальными в пределах описания процесса: нельзя одинаково назвать два разнотипных события одного процесса, но использование одного имени события в описаниях разных процессов не возбраняется.

Параграф коммутации начинается ключевым словом **COMMUTATION**. За ключевым словом следуют операторы коммутации, они бывают двух видов:

1. Коммутация входящих параметров методов, ее синтаксис:

   <имя метода>«**.**»<имя входящего параметра> «**=**»
   [**phase** | **param** «**.**»]
   [{<имя типа данных> «**.**»}]<имя поля>«**;**»

   Входящий параметр метода связывается с внутренней (ключевое слово **phase**) или внешней (ключевое слово **param**) характеристикой компоненты. Если в описателе компоненты нет параграфа параметров, квалификатор «**phase.**» можно опустить, оставив в операторе коммутации лишь имя поля.

2. Коммутация возвращаемых параметров методов, ее синтаксис:

   [{<имя типа данных> «**.**»}]
   <имя поля внутренней характеристики> «**=**»
   <имя метода>«**.**»
   <имя возвращаемого параметра>«**;**»

Возвращаемый параметр метода связывается с полем внутренней характеристики компоненты. До нужного элемента данных, возможно, придется добираться через несколько квалификаторов, поэтому все имена в операторах коммутации вполне могут иметь вид

{<идентификатор>**.**}<поле>

где поле – либо идентификатор, либо элемент массива.

Для успешной компиляции параграфа коммутации необходимо, чтобы описатели методов, присутствующих в операторах коммутации, были откомпилированы до этого. При компиляции операторов коммутации проверяется соответствие типов коммутируемых полей.

Параграф переключателей начинается ключевым словом **SWITCHES**. За ним следуют операторы переключений, которые имеют вид:

[<метка процесса> «:»]
<имя текущего элемента> «,»
<имя следующего элемента >
[«:»<имя события>] «**;**»

Если переход безусловный, т. е. происходит всегда, что достаточно типично для быстрых элементов, – имя события не указывается.

Примеры описаний моделей-компонент –

Модель-компонента «пешеход»:

```
COMPONENT Man;
 PHASE
 ManPhase:
 double x;
 PARAMETERS
 ManParam:
 double v;
```

```
 ELEMENTS
 move;
 COMMUTATION
 move.x = x;
 move.v = v;
 x = move.x;
 END;
```

Модель-компонента «муха»:

```
COMPONENT Fly;
 PHASE
 FlyPhase:
 double x,v;
 PARAMETERS
 FlyParam:
 FlyPhase man0Phase, man1Phase;
 ELEMENTS
 move, Uturn;
 EVENTS
 reaching;
 SWITCHES
 Move, Uturn: reaching;
 Uturn, move;
 COMMUTATION
 move.x=FlyPhase.x;
 move.v=FlyPhase.v;
 FlyPhase.x=move.x;
 Uturn.v=FlyPhase.v;
 FlyPhase.v=Uturn.v;
 reaching.x=FlyPhase.x;
 reaching.v=FlyPhase.v;
 reaching.m0x=FlyParam.man0Phase.x;
 reaching.m0v=FlyParam.man0Phase.v;
 reaching.m1x=FlyParam.man1Phase.x;
```

```
reaching.m1v=FlyParam.man1Phase.v;
END;
```

Модель-компонента «пешеходы и муха».

Модель-комплекс «пешеходы и муха», рассматриваемая как компонента, в результате синтеза из своих компонент (распространения рода структуры «модель-компонента») – описание генерируется автоматически, в результате компиляции описателя модели-комплекса «пешеходы и муха», при условии, что все входящие в модель-комплекс модели-компоненты были успешно откомпилированы:

```
COMPONENT menANDflyAScomp;
PHASE
menANDflyPhase:
double FlyPhase_0_x, FlyPhase_0_v, ManPhase_0_x,
ManPhase_1_x;
PARAMETERS
menANDflyParam:
double ManParam_0_v, ManParam_1_v;
ELEMENTS
1: Man_0_move: move;
2: Man_1_move: move;
3: Fly_0_move: move, Fly_0_Uturn: Uturn;
EVENTS
3: Fly_0_reaching: reaching;
SWITCHES
3: Fly_0_move, Fly_0_Uturn: Fly_0_reaching;
3: Fly_0_Uturn, Fly_0_move;
COMMUTATION
Man_0_move.x=ManPhase_0_x;
Man_0_move.v=ManParam_0_v;
ManPhase_0_x=Man_0_move.x;
```

```
Man_1_move.x=ManPhase_1_x;
Man_1_move.v=ManParam_1_v;
ManPhase_1_x=Man_1_move.x;
Fly_0_move.x=FlyPhase_0_x;
Fly_0_move.v=FlyPhase_0_v;
FlyPhase_0_x=Fly_0_move.x;
Fly_0_Uturn.v=FlyPhase_0_v;
FlyPhase_0_v=Fly_0_Uturn.v;
Fly_0_reaching.x=FlyPhase_0_x;
Fly_0_reaching.v=FlyPhase_0_v;
Fly_0_reaching.m0x=ManPhase_0_x;
Fly_0_reaching.m0v=ManParam_0_v;
Fly_0_reaching.m1x=ManPhase_1_x;
Fly_0_reaching.m1v=ManParam_1_v;
END;
```

4.5.7. Компиляция описателей ЯОКК

Основой компиляции описателей ЯОКК можно назвать компиляцию типов данных. Для каждого проекта имитационной модели сложной системы, состоящего из набора комплексов и компонент, в базе данных заводится таблица типов данных, в которую автоматически заносятся встроенные в ЯОКК типы данных.

Таблица типов данных отличается от приведенной в пункте 4.5.3 наличием еще одного столбца, где для каждого типа данных хранится в текстовом виде результат его полного синтаксического анализа – схема типа данных. Для встроенных типов данных схема совпадает с именем встроенного типа. Для более сложных типов данных схема содержит все имена промежуточных типов данных и имена их полей, начиная от имени типа в целом, которое может быть сгенерировано автоматически,

если не было задано явно в соответствующем описателе, и кончая терминальными полями встроенных типов.

По схеме типа можно однозначно определить, имеются ли у рассматриваемого типа данных поля с заданным именем, сколько таких полей, а также вычислить величину смещения в байтах любого поля типа от начала данных.

Основной задачей компиляции явного описания типа данных (описатель типа данных), или неявного их описания (описания параметров методов, внешних и внутренних характеристик компонент) как раз и является построение схемы типа данных и определение размера данных этого типа в байтах.

Поскольку новый тип данных определяется как набор полей и/или массивов существующих типов - процесс компиляции состоит в поиске в уже упоминавшейся таблице базы данных имен этих типов. Если тип найден в таблице, его схема включается в схему вновь создаваемого типа, а размер учитывается при определении размера нового типа данных. Если же какой-либо из типов данных, используемых в описании компилируемого типа, отсутствует в таблице, – компиляция описателя заканчивается аварийно с соответствующей диагностикой.

Схемы типов данных оказываются очень важными при компиляции операторов коммутации во всех описателях ЯОКК, где такие операторы встречаются – в описателях комплексов и компонент. Прежде всего, поскольку в операторах коммутации требуется не полная квалификация имен коммутируемых полей (хотя она, конечно же, не возбраняется), а минимальная, не приводящая к неоднозначности, то по соответствующим схемам типов проверяется, сколько полей с запрашиваемыми именами существует в схеме рассматриваемого типа данных. Если их

не оказывается или оказывается больше одного – компиляция завершается аварийно с соответствующей диагностикой. Если и слева и справа в операторе коммутации находится ровно одно поле, поля проверяются на соответствие типов. Здесь может быть несколько уровней строгости контроля их соответствия:

1. Самый строгий – проверяется соответствие ключей в таблице типов базы данных.
2. Проверяется соответствие схем данных, т.е. разрешается коммутировать синонимы типов данных.
3. Проверяется соответствие коммутируемых полей, выраженных через встроенные типы данных, т.е. разрешается коммутировать поля, возможно, существенно различных типов – не синонимов, но одинаково представляемые встроенными типами данных (например, массив **double**, и столько же отдельных полей этого типа).
4. Проверяется соответствие размеров коммутируемых полей.
5. Иногда могут иметь смысл коммутации даже при несоответствии размеров коммутируемых полей.

В случае несоответствия полей при выбранном уровне проверки строгости, происходит аварийное завершение компиляции с соответствующей диагностикой. В случае соответствия заполняется соответствующая таблица базы данных. Например, в случае коммутации параметров метода с характеристиками модели-компоненты можно хранить в строке таблицы коммутации, соответствующей оператору коммутации, номера метода и модели-компоненты в проекте, смещения от начала данных для типов параметров метода и характеристик компоненты, и, наконец, номер в проекте типа коммутируе-

мого поля. Ссылку на тип, а не длину коммутируемого поля имеет смысл хранить потому, что она может пригодиться, например, при дальнейшей коммутации моделей-компонент в модель-комплекс.

При компилировании операторов коммутации описателя компоненты, также должно проверяться:

1. Все ли характеристики модели задействованы, т.е. если какая-либо внутренняя характеристика не изменяется ни одним из методов-элементов – об этом должно быть диагностическое сообщение (возможно, это внешняя характеристика модели). Если внешняя характеристика компоненты не передается ни одному из методов-элементов или методов-событий – об этом также должно быть диагностическое сообщения.
2. Нет ли попыток изменять внешние характеристики модели. Если такие попытки имеются, должна быть соответствующая диагностика (возможно, эти характеристики внутренние).
3. Должна быть предупреждающая диагностика, если методы-элементы разных процессов изменяют одну и ту же внутреннюю характеристику модели-компоненты. В этом может и не быть криминала, если поведение модели таково, что эти методы никогда не вызываются одновременно в модельном времени. Однако если такие методы могут попасть в набор методов-элементов, вызываемых параллельно, должна быть соответствующая диагностика времени выполнения. Вообще говоря, такая ситуация говорит о том, что модель спроектирована неверно: если одну и ту же характеристику пытаются одновременно получить несколькими различными способами, то это свидетельствует о

том, что нарушена однозначность вычислительного процесса моделирования. О способах преодоления подобных неоднозначностей подробно говорилось в разд. 3.3.

Что касается компиляции описателей моделей-компонент, нерассмотренными остались параграфы элементов, событий и переключений. Вместе эти три параграфа полностью описывают процессы модели-компоненты, т.е. ее поведение. На стадии компиляции здесь необходимы следующие проверки:

1. У процессов не должно быть изолированных методов-элементов, т.е. таких, в которые принципиально невозможно попасть из начального элемента за конечное число шагов – не описаны соответствующие переключения.
2. Для быстрых элементов наряду с переходами по событиям, желателен безусловный переход. Отсутствие такового не будет криминалом, если всегда реализуется одно из событий, обеспечивающих переход из быстрого элемента. Если же нет, возникает диагностика времени выполнения: элемент закончился, а к какому переходить далее, не указано.
3. Не должно быть неиспользуемых событий, т.е. событие, появившееся в параграфе событий, должно встретиться и в параграфе переключений. И наоборот, встречающиеся в параграфе переключений события должны быть описаны в параграфе событий, при этом события должны относиться к одному и тому же процессу.

Наконец, несколько слов о компиляции описателя модели-комплекса. Одно из важных отличий его компиляции от компиляции всех остальных описателей ЯОКК

состоит в том, что ее результат – не записи в тех или иных таблицах связанной с проектом базы данных, а текстовый файл описателя модели-компоненты, в которую превращается модель-комплекс. В дальнейшем этот описатель модели-компоненты, полученный автоматически, подлежит обычной компиляции, как и любой описатель модели-компоненты.

Неоднозначности внешних характеристик комплекса можно разрешать с помощью операторов коммутации параграфа **SAMENESS**. К этим операторам, а также к операторам обычной коммутации относятся все обычные проверки, относящиеся к коммутации, о которых уже было сказано выше.

Для разрешения неоднозначностей, связанных с методами комплекса (см. разд. 3.3) возможно, придется скорректировать указатели реализаций этих методов в соответствующих описателях, а еще проще – в базе данных проекта.

Неоднозначности внутренних характеристик комплекса нужно разрешать вручную, так, как это было описано в разд. 3.3, т.е. вводя в комплекс новые компоненты, чья задача – по правилам, определенным разработчиком модели, на основании набора характеристик, имеющих одинаковый смысл в данной модели, выдавать единственную характеристику, несущую этот смысл.

Для компиляции описателя модели-комплекса необходимо, чтобы все входящие в него модели-компоненты были до этого успешно откомпилированы.

5. Модельный синтез и объектный анализ

> «UML подчиняется правилу 80/20, т.е., 80% большинства проблем можно смоделировать, используя 20% средств UML».
> *Г. Буч, Д. Рамбо, И. Якобсон*

> В модельном синтезе имеется единственное основное понятие – модель-компонента и одно вспомогательное понятие – модель-комплекс, который, в конце концов, также превращается в модель-компоненту.

Прежде всего следует отметить, что объектный анализ последние тридцать лет является основным средством проектирования и реализации любых программных систем. В то же время модельный синтез не претендует на освоение всего поля разработки сложных программных систем. Изложенная выше концепция предлагается в первую очередь для синтеза имитационных моделей сложных многокомпонентных систем – систем, состоящих из множества агентов, каждый из которых обладает собственным поведением – способностью определенным образом откликаться на происходящее внутри и снаружи этого агента. При этом поведение модели всей сложной системы в целом интересует нас в первую очередь как изменения во времени ее характеристик.

Далеко не все сложные программные системы таковы. Например, при разработке транслятора, если действующих агентов еще иногда можно придумать, то разворачивание изменения характеристик во времени не интересно совсем – в идеальном случае хорошо бы все транслировать мгновенно – здесь может происходить лишь чередование различных стадий обработки исходных данных.

Однако можно надеяться, что предлагаемый подход применим для разработки некоторого не очень малого класса сложных программных систем, организация которых атомистична, допускает выделение агентов, обладающих поведением, и разворачивается индуктивно снизу вверх.

5.1. Элементы модельного анализа и проектирования имитационных моделей сложных систем

До сих пор мы все время говорили о синтезе более сложных моделей-комплексов из более простых моделей-компонент. На самом деле, на основе методов модельного синтеза вполне возможно проектировать имитационные модели сложных многокомпонентных систем сверху вниз, начиная с самых агрегированных представлений о модели и постепенно переходя к детальной проработке состава и организации внутренних структуры и связей каждого из таких агрегатов.

Пусть модель-компонента будет терминальным символом некоторого алфавита, а модель-комплекс – нетерминальным. Определим следующее правило вывода, применимое к нетерминальному символу: модель-комплекс может состоять из моделей-комплексов и/или моделей-компонент. Правило вывода применяется, пока имеется хотя бы один нетерминальный символ.

Указанное выше правило вывода порождает древовидные структуры, подобные файловой системе компьютера, где модель-комплекс есть аналог папки, а модель-компонента – файла. Применяя это правило вывода и выводя в конечном итоге с его помощью модели-компоненты, тем не менее запоминаем и все вложенности компонент и комплексов в те комплексы более высокого уровня, в которые они входят.

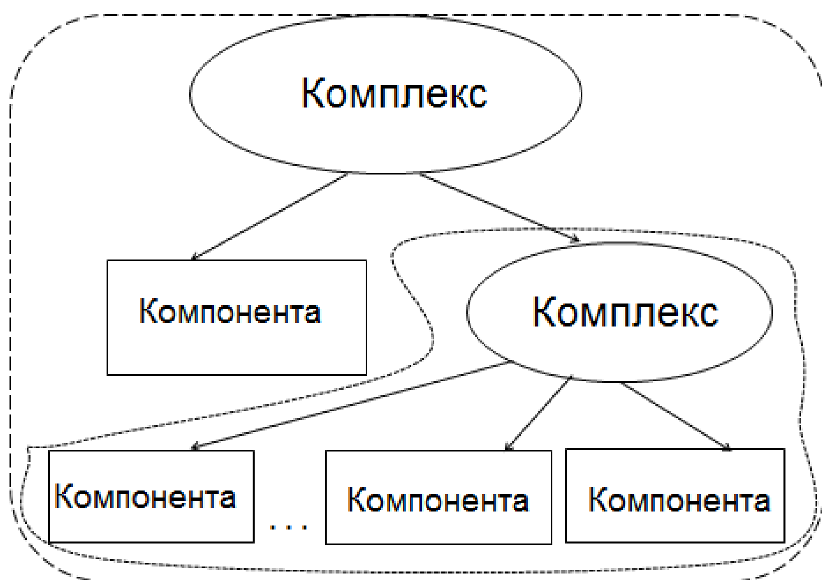

Рисунок 1. Иерархическое проектирование имитационной модели сложной системы.

На рис. 1 пунктиром очерчено то, что входит в различные комплексы.

Таким образом, мы проектируем сложную систему, каждый раз конкретизируя, из чего состоит комплекс и как взаимосвязаны его компоненты. Когда мы дойдем до комплексов, состоящих из одних только моделей-компонент, можно по выработанным в разд. 3.3 правилам синтеза начинать обратное движение снизу вверх, представляя модели-комплексы моделями-компонентами, и, таким образом, строить снизу вверх синтез нашей модели сложной многокомпонентной системы.

5.2. Модельный синтез vs объектный анализ

Теперь остановимся на сравнении основных понятий модельно-ориентированного и объектно-ориентированного программирования. Прежде всего отметим, что объектно-ориентированный подход в настоящее время представлен в двух видах: один из них можно назвать «базовым», в его основе лежат такие понятия, как класс, объект, типизация, наследование, инкапсуляция, полиморфизм[1], которые реализованы с некоторыми нюансами в большинстве современных императивных языков программирования, таких как C++, Java, C#, Delphi и многих других.

Второй, который можно назвать «расширенным», представлен унифицированным языком моделирования UML [16, 27]. Язык UML включает все перечисленные выше базовые понятия, кроме того, его создатели пошли по пути резкого увеличения числа исходных понятий и представлений. Например, кроме «вертикального» отношения наследования, которое в UML чаще называется отношением обобщения (при этом отношение обобщения направлено от потомка к предку), имеются отношения ассоциации, композиции, агрегации и зависимости. Появилась возможность описывать поведение систем, причем даже несколькими способами: диаграммы взаи-

---

[1] Термин «полиморфизм», хотя и является общепринятым, не представляется нам удачным, поскольку его название неверно передает суть обозначаемого им явления – переопределения методов. Действительно, то, что сохраняет объект после переопределения методов, – это именно форма: унаследованные от предка характеристики, а также сигнатуры методов. А то, что при этом изменяется – не форма, а содержание вычислений – то, что переопределенные методы выполняют. Поэтому в дальнейшем, вместо полиморфизма мы будем говорить о переопределении методов.

модействия, диаграммы состояний и диаграммы деятельности.

Вообще в UML очень многое можно делать несколькими способами, что делает его удобным инструментом знатока, но весьма затрудняет жизнь новичку. Сами авторы языка говорят, что: «UML подчиняется правилу 80/20, т.е. 80% большинства проблем можно смоделировать, используя 20% средств UML» [16].

В отличие от объектного анализа UML модельный синтез минималистичен в наборе базовых понятий: в нем имеется единственное основное понятие – модель-компонента и одно вспомогательное понятие – модель-комплекс, состоящий из моделей-компонент, который, в конце концов, в результате устранения возникших неоднозначностей также может стать моделью-компонентой. Правда, на широту охвата «всего на свете» модельный синтез не претендует, область его применения – сложные многокомпонентные «атомистические» системы, которые естественно строить снизу вверх.

Продолжим сравнение базовых понятий. Такие понятия, как класс, объект, типизация в двух рассматриваемых подходах понимаются примерно одинаково. Следует лишь заметить, что корни этих понятий следует искать конечно не в С++ и даже не в Симуле-67, а, скорее, в работах Н. Бурбаки [13], структуралистов XX века [17], вплоть до Эрлангенской программы Ф. Клейна [21].

Примерно одинаково понимаются также характеристики и методы. Что же касается использования методов, здесь имеется существенная разница. Она состоит в том, что в объектном программировании объект затем и нужен, чтобы вызывать его методы в различных программах, причем в качестве параметров методу можно передать, а также принять от него любые переменные, удо-

влетворяющие его сигнатуре – не обязательно характеристики его объекта. В модельно-ориентированном программировании метод модели-компоненты может работать лишь с ее характеристиками и вызывать его «вручную» нет ни возможности, ни необходимости; он будет вызван автоматически тогда, когда этого потребует логика поведения модели-компоненты. Инкапсуляция в модельно-ориентированном программировании такова, что не позволяет напрямую добраться до методов. Оперировать можно лишь моделями-компонентами.

Это вовсе не означает, что методы совсем недоступны. Наоборот, например, в реализованном макете распределенной системы имитационного моделирования [8, 12], библиотеки методов публикуются в интернете для общего использования, и возможны модели-компоненты, не имеющие ни одного локального метода, притом все реализации методов могут физически находиться в разных местах сети. Тем не менее воспользоваться методом можно, лишь включив его в состав некоторой модели-компоненты, и работать он в ней будет не сам по себе, а в соответствии с ее логикой поведения – таков уровень инкапсуляции. В некоторых случаях это может показаться усложнением, но оно оплачено отсутствием забот об организации поведения модели-компоненты – она всегда ведет себя так, как умеет, и поэтому включение ее в любой комплекс всегда всего лишь вопрос правильной коммутации.

Переопределение методов в модельно-ориентированном программировании может быть легко достигнуто путем указания иной реализации в соотношении типизации (3) из разд. 3.2, если речь идет об элементе, или (4), если речь идет о событии.

Скажем несколько слов о наследовании. Определим базисное множество рода структуры «объект» как совокупность его характеристик X и методов M. Вводится отношение наследования, которое по базисному подмножеству характеристик есть включение множества характеристик предка во множество характеристик потомка, а по базисному подмножеству методов есть инъекция множества методов предка во множество методов потомка, при этом сохраняющая так называемую сигнатуру, т.е. названия методов и типы их параметров. Сами же реализации методов потомка могут отличаться в силу возможности переопределения потомком методов предка. Отношение наследования, рассматриваемое в направлении от потомка к предку, в объектном анализе называют также отношением обобщения. В объектно-ориентированных языках программирования по своему типу потомок является предком, но предок не является потомком (аксиома антисимметрии отношения частичного порядка), поэтому переменные типа потомка могут быть использованы как значения переменных или параметров типа предка, но не наоборот.

Таким образом, отношение наследования для множества классов объектно-ориентированного языка программирования есть отношение частичного порядка. Классы, не имеющие предков, но обладающие потомками, называются по отношению к ним корневыми или базовыми. Классы, не имеющие потомков, называются листовыми.

Проектирование в объектно-ориентированной парадигме большой программной системы состоит в воплощении базовых понятий и представлений этой системы в базовые классы объектов и построении затем иерархии классов, развивающих, конкретизирующих и

воплощающих эти идеи во множестве листовых классов, с помощью которых и будет строиться целевая программная система.

С точки зрения геометрической теории декомпозиции, набор базовых классов представляет собой F-редукции будущих наборов листовых классов, которые разворачивается из набора базовых сверху вниз. Наследование в объектно-ориентированном программировании носит характер постепенного выявления различий в первоначально однородном фактор-множестве, поэтому такой тип наследования можно назвать F-наследованием. Такой дедуктивный способ проектирования большой программной системы хорош, когда она творится «с чистого листа», подобно миру у Платона.

В имитационном моделировании, однако, гораздо чаще возникает задача не создания новых миров, а воспроизведения фрагментов уже существующего. В подобном фрагменте запросто могут быть собраны вместе «…диван, чемодан, саквояж, картина, корзина, картонка и маленькая собачонка». Друг из друга они не выводятся, а восходить вверх до их общих предков – достаточно бессмысленно. Для подобных задач «базовому» варианту объектного подхода недостает наследования снизу вверх (в UML такое наследование имеется, что впрочем, все равно не делает его удобным инструментом проектирования имитационных моделей).

Возникает вопрос, возможно ли в модельно-ориентированном программировании дедуктивное F-наследование сверху вниз? Включая компоненту в комплекс, мы тем самым всегда получаем в распоряжение ее характеристики и методы. Как уже говорилось выше, методы переопределяются легко, например путем указания других URL их реализаций. Однако просто так добавить характе-

ристик мы не можем, их всегда должен кто-то обрабатывать. Но мы можем добавить новую компоненту, в состав которой входят дополнительные характеристики, при этом возможно коммутировать эти характеристики со старыми, унаследованными.

Таким образом, в модельно-ориентированном программировании объединение моделей-компонент в модель-комплекс можно считать множественным наследованием снизу вверх. Точно так же, как в языках программирования, где множественное наследование разрешено, в модельно-ориентированном программировании оно может приводить к некоторой неоднозначности, которую тем не менее можно разрешить способом, показанным в разд. 3.3. При этом модели-компоненты, хотя могут и не являться точными P-подобъектами содержащей их модели-комплекса из-за возможных межкомпонентных связей, тем не менее очень на такие подобъекты похожи. Поэтому объединение компонент в комплекс с позиций геометрической теории декомпозиции можно назвать P-наследованием.

Геометрическая теория декомпозиции [23] утверждает, что F- и P-декомпозициями и их сочетаниями исчерпываются всевозможные декомпозиции математических объектов, при этом F- и P- конструкции двойственны. Поэтому отсутствие одной из них (например, в механизмах наследования), является признаком функциональной неполноты любого метода, в особенности претендующего на универсальность.

Даже если в объектно-ориентированном проекте с помощью наследования построена самая замечательная иерархия классов, все равно все сложности организации вычислительного процесса, состоящего в использовании разработанных и отлаженных методов листовых классов,

лежит на разработчике системы: чтобы система что-то делала, необходимо организовать вызов нужных методов в нужной последовательности. Между прочим, на этом этапе и теряется модульность объектно-ориентированной программы: обычно все используемые методы так или иначе связаны друг с другом. Для настоящей модульности (например, такой как в САПР микроэлектроники), не хватает уровня инкапсуляции: из объектов приходится вручную в нужной последовательности вызывать их методы. Самый сложный этап построения большой программной системы остается неформализованным – это искусство.

Попытки формализовать процесс проектирования сложных программных систем и породили UML. По-видимому, и в самом деле на UML можно описать любую систему и даже с нескольких точек зрения. Вопрос в том, что делать дальше с таким описанием, здесь нет единства мнений.

Некоторые специалисты (например, [27]) считают, что основная ценность UML как раз в применении как средства документирования и обмена формализованными описаниями на стадиях эскиза и проектирования сложных систем. Судя по некоторым изменениям работы [16] (2-е изд.), при переизданиях создается впечатление, что и сами авторы языка UML начинают склоняться к такому мнению.

Тем не менее имеется и ряд средств, позволяющих компилировать UML-описания в заготовки классов универсальных языков программирования, и в этом случае можно говорить о режиме использования UML в качестве языка программирования. Однако здесь мы снова остаемся в рамках объектной парадигмы, получаем иерархию классов и заготовок классов, но не избавляемся от необ-

ходимости писать императивные программы, вызывающие в нужном порядке методы этих классов.

В рамках объектно-ориентированной императивной парадигмы с F-наследованием сверху вниз остается и возникшая на основе UML концепция разработки программных систем, управляемая моделями, известная в различных вариантах под названиями MDA (Model Driven Architecture), MDE (Model Driven Engineering) или MDD (Model Driven Development).

Эта концепция предполагает, вооружившись всей мощью UML, начинать разработку с построения абстрактной модели программной системы, которая затем воплощается в платформенно-независимую модель, которая преобразуется (возможно, автоматически) в платформенно-зависимую, из которой (возможно, автоматически) генерируется программный код. Здесь снова разворачивание проекта происходит сверху вниз, и все его дальнейшее развитие остается в парадигме объектно-ориентированного программирования. Следовательно, неизбежно должно завершиться написанием императивных программ, вызывающих методы как полученных автоматически, так и созданных вручную классов.

При компиляции языка UML и при автоматической трансформации в код моделей MDA/MDD/MDE неизбежно возникает вопрос о качестве этой компиляции и эффективности получаемого кода – UML достаточно сложный язык, располагающий множеством различных не слишком простых понятий и представлений. Собственно, именно сложности такого рода и заставляют многих специалистов считать UML в первую очередь, именно средством документирования и эскизного проектирования сложных программных комплексов. Программную реализацию комплекса при этом создают вручную на стан-

дартных объектно-ориентированных языках, пользуясь описанием эскиза программного комплекса, выполненного на UML.

Наконец, следует заметить, что в последнее время все более востребованными становятся высокопроизводительные многопроцессорные и многоядерные вычислительные системы. Связано это в том числе и с тем, что на горизонте уже начали вырисовываться физические ограничения для увеличения производительности однопроцессорных компьютеров. Объектный подход ни в «базовой», ни в «расширенной» версии здесь ничего особенного не предлагает – задача параллелизации вычислений остается одной из самых сложных и трудно формализуемых.

Модельный синтез и модельно-ориентированное программирование предполагают более высокий уровень инкапсуляции, нежели принят в объектном анализе. Основная единица программы – модель-компонента – наделена самостоятельным поведением, поэтому организовывать вычислительный процесс путем вызова каких-либо методов не нужно и невозможно, он всегда организуется автоматически в соответствии с описанным поведением компонент (правилами переключения методов-элементов в процессах моделей-компонент, – соотношения типизации (8) в разд.3.2).

Наличие у модели-компоненты собственного поведения позволяет последовательно провести идею модульности (до сравнимой, например, с САПР микроэлектроники степени) разрабатываемой с помощью модельного синтеза программной системы. Как в микроэлектронике однажды разработанную микросхему можно включить в любую микросборку или печатную плату, зная ее функцию и правила соединения и не заботясь о ее

внутреннем устройстве на уровне транзисторов – просто правильно припаять, так и в модельном синтезе однажды разработанную и отлаженную модель-компоненту можно включать в любой комплекс, зная правила коммутации и выполняемую функцию, позабыв полностью подробности ее реализации. При этом работать она всегда будет сама, как и микросхема, в полном соответствии с описанными где-то ранее правилами поведения, дополнительно заботиться об этом не нужно. Играющие роль «паяльника» операторы коммутации, например языка ЯОКК, полностью декларативны, т.е. программировать в привычном смысле императивного программирования, для построения сложных программных комплексов из готовых моделей-компонент, не приходится.

Вообще, модельный синтез полностью исключают императивное программирование. Это следует из того, что гипотеза о замкнутости (разд. 2.4.) гарантирует нам функциональную зависимость на интервале модельного времени $(t,t+\Delta t]$ внутренних характеристик модели от ее внутренних и внешних характеристик на левом конце этого интервала – в точке t.

Если зависимость возвращаемых параметров методов-элементов от их входящих параметров функциональная, ее вычислительный процесс вполне может быть осуществлен в функциональной парадигме программирования (пусть и на обычных универсальных языках), что существенно упрощает отладку программы, а в некоторых случаях может к тому же увеличить степень параллельности получаемого кода.

В рамках концепции модельного синтеза, устройство модели последовательно описывается на декларативном языке ЯОКК. Описатели ЯОКК компилируются в таблицы базы данных совершенно определенной струк-

туры. Вопрос о качестве такой компиляции, как это часто бывает с декларативными описаниями, не стоит: компиляция либо верна, если компилятор работает правильно, а если неверна, значит, компилятор нужно отлаживать. Вопрос об эффективности получаемого кода здесь не стоит, потому что этого кода просто нет – генерируются и заполняются таблицы базы данных известной формы.

Наконец, последнее преимущество модельного синтеза – ориентированность программы организующей имитационные вычисления (аксиома поведения модели-компоненты R_{11} из разд. 3.2) на параллельное выполнение множества методов-элементов и методов-событий. При этом при усложнении модели, количество методов, которые допускают параллельное вычисление, также растет, пропорционально росту количества всех процессов модели.

Заключение

Не только ***как***, но и ***почему так.***

В настоящее время область построения моделей сложных многокомпонентных систем относится в значительной мере к области искусства, поскольку все решения основаны на эвристических представлениях их авторов о моделировании сложных систем и эвристических способах, пусть даже и весьма успешных, решения различных проблем, возникающих при таком моделировании.

В данной работе, оттолкнувшись от некоторых общепризнанных в кругах разработчиков имитационных моделей положений, таких как требование замкнутости модели, однозначности и детерминированности имитационных вычислений, а также необходимости реализации имитационных вычислений на компьютере за конечное время, мы попробовали вывести некоторые необходимые свойства получаемых моделей. Одно из таких свойств – кусочно-непрерывный, с не более чем конечным числом разрывов первого рода, характер траектории модели. При этом, переопределяя траекторию модели не более чем в конечном количестве точек, можно считать, что траектория модели полунепрерывна слева.

Выделен класс моделей, (а именно удовлетворяющих в каждой точке гипотезе о замкнутости и имеющих кусочно-гладкую, непрерывную слева траекторию), для которых из локальной гипотезы о замкнутости модели будет следовать возможность успешного синтеза модели на конечном отрезке моделирования (разд. 2.4, предложение 2.4.1).

Предложена организация имитационных вычислений, ориентированная на модели с кусочно-гладкими траекториями, как инварианта относительно объединения моделей-компонент в модели-комплексы, позволяющая полностью решить задачу описания и синтеза имитационных моделей сложных многокомпонентных систем.

Все это позволяет формально определить класс имитационных моделей сложных многокомпонентных систем, как род структуры «модель-компонента» в смысле Н. Бурбаки, и на основе такого определения построить новые концепции формального описания, синтеза и реализации имитационных моделей – модельный синтез и модельно-ориентированное программирование, являющиеся альтернативой повсеместно используемых объектного анализа и объектно-ориентированного программирования, прежде всего для задач имитационного моделирования сложных многокомпонентных систем.

Концепция модельного синтеза минималистична в отношении базовых понятий и представлений: в ней всего одно основное понятие – модель-компонента, и одно вспомогательное – модель-комплекс, который путем несложных операций синтеза, описанных в разд. 3.3, в конечном итоге также превращается в модель-компоненту.

Концепции модельного синтеза и модельно-ориентированного программирования полностью исключают из проекта создания имитационной модели наиболее трудоемкое в проектировании, реализации и отладке императивное программирование, ограничиваясь лишь программированием декларативным и функциональным. Кроме того, модельно-ориентированное программирование производит код высокой степени параллельности.

Описание модели сложной системы состоит из набора декларативных описаний моделей-компонент и составленных из них моделей-комплексов на специальном декларативном языке ЯОКК. При этом многие из этих описаний (например, компоненты одного комплекса) не зависят друг от друга и поэтому могут создаваться и отлаживаться независимыми группами разработчиков. Синтез имитационной модели сложной многокомпонентной системы строится путем компиляции этих описаний. Результатом компиляции любой модели-компоненты (а таковой, как было показано, является, в том числе и вся модель сложной системы), является некий набор таблиц постоянной структуры в базе данных. Для того чтобы запустить модель-компоненту на счет, остается внести в базу данных начальные значения ее характеристик, а также написать на универсальных языках программирования (можно в функциональной парадигме) и отладить ее методы.

То, что конкретно делает модель (методы-элементы, проявляющие ее поведение), полностью отделено от того, почему она что-то делает (методы-события, реагирующие на изменения внутренней и внешней среды), и от того, каким образом она что-то делает (описания логики поведения модели, соотношениями типизации (8) из разд. 3.2).

Таким образом, построение большой программной системы, реализующей имитационную модель, разбивается на вполне обозримые, слабо зависящие друг от друга фрагменты декларативного описания моделей-компонент и моделей-комплексов, а также написание на универсальных языках программирования, но в функциональной парадигме, методов-элементов и методов-событий моделей-компонент.

Отладка декларативных описаний моделей-компонент, моделей-комплексов, а также функциональных программ, реализующих методы, происходит по принципу – отладил и забыл. Так же как, например, при проектировании изделий микроэлектроники, про модель-компоненту достаточно знать, какие функции она выполняет, какие у нее входы и выходы, после чего ее смело можно включать в любой комплекс с полной уверенностью, что она будет функционировать правильно, если ее входы и выходы правильно скоммутированы внутри этого комплекса. Императивное программирование при этом полностью исключается из проекта, что облегчает его отладку.

Вычислительный процесс имитационного моделирования (аксиома поведения R_{11} рода структуры «модель-компонента»), организуется так, что чем сложнее фрактальная конструкция модели, тем более высокая степень параллельности кода, который производит программа, реализующая аксиому R_{11}, тем большее количество методов может быть вызвано параллельно.

Изложенная концепция модельного синтеза применима в первую очередь для синтеза моделей сложных многокомпонентных систем. Однако можно надеяться, что подобный подход применим и для разработки сложных программных систем, организация которых атомистична и укладывается в описанные во второй главе требования гипотезы о замкнутости, а также требования однозначности и детерминированности вычислений.

Также есть надежда применения предложенных методов модельно-ориентированного программирования для программных систем, ориентированных на высокопроизводительные многопроцессорные вычислительные системы.

Разработанная концепция моделирования сложных систем была реализована на практике в виде ряда имитационных систем, реализованных под влиянием модельно-ориентированной парадигмы программирования, инструментальной системы имитационного моделирования MISS [10], а также в виде программного обеспечения макета рабочей станции пиринговой системы распределенного имитационного моделирования [8].

На рабочей станции системы распределенного имитационного моделирования, во-первых, можно полностью создавать модели сложных систем из элементов собственной разработки и элементов, опубликованных другими участниками пиринговой сети. Во-вторых, можно публиковать в сети реализованные на этой рабочей станции элементы, которые после этого становятся доступными для включения их в модели, разрабатываемые на других станциях.

В настоящее время в отделе имитационных систем и исследования операций ВЦ РАН ведутся работы по реализации системы модельно-ориентированного программирования для высокопроизводительных вычислительных систем.

Литература

1. *Brodsky Yury I.* Simulation Software //System Analysis and Modeling of Integrated World Systems – V 1, Oxford: EOLSS Publishers Co. Ltd., 2009. P. 287-298.
2. *Brodsky Yury I., Tokarev Vladislav V.* Fundamentals of simulation for complex systems. //System Analysis and Modeling of Integrated World Systems – V 1, Oxford: EOLSS Publishers Co. Ltd., 2009. P. 235-250.
3. Hewitt Carl Viewing Control Structures as Patterns of Passing Messages Journal of Artificial Intelligence. June 1977.
4. *Kuhl F., Weatherly R., Dahmann J.* Creating Computer Simulation Systems: An Introduction to the High Level Architecture NY: Prentice Hall PTR, 1999. 212 p.
5. *Shoham Y.* Agent-oriented programming //Artificial Intelligence, vol. 60, 1993. P. 51-92.
6. *Shoham Y.* MULTIAGENT SYSTEMS: Algorithmic, Game-Theoretic, and Logical Foundations Cambridge: Cambridge University Press, 2010, 532 p.
7. Zave, P. A compositional approach to multiparadigm programming. IEEE Software, 6(5): 15—25, September 1989.
8. *Бродский Ю.И.* Распределенное имитационное моделирование сложных систем М.: ВЦ РАН, 2010, 156 с.
9. *Бродский Ю.И.* О модельном анализе как альтернативе объектному, в задаче описания и синтеза имитационных моделей сложных многокомпонентных систем //Инновации на основе информационных и коммуникационных технологий: Материалы международной научно-практической конференции. М.: МИЭМ НИУ ВШЭ, 2013. С. 181-183.

10. *Бродский Ю.И., Лебедев В.Ю.* Инструментальная система имитации MISS М.: ВЦ АН СССР, 1991, 180 с.
11. *Бродский Ю.И., Мягков А.Н.* Декларативное и императивное программирование в имитационном моделировании сложных многокомпонентных систем //Вестник МГТУ им. Н.Э. Баумана. Сер. Естественные науки. Спец. выпуск № 4 «Математическое моделирование». 2012. С.178-187.
12. *Бродский Ю.И., Павловский Ю.Н.* Разработка инструментальной системы распределенного имитационного моделирования. //Информационные технологии и вычислительные системы, №4, 2009. С. 9-21.
13. *Бурбаки Н.* Теория множеств. М.: Мир. 1965. 456 с.
14. *Бусленко Н.П.* Моделирование сложных систем М.: Наука, 1978, 400 с.
15. *Бусленко Н.П.* Сложная система //Статья в Большой Советской Энциклопедии, 3-е изд., М.: Советская энциклопедия, 1969-1978.
16. *Буч Г., Рамбо Д., Якобсон И.* Введение в UML от создателей языка. 2-е изд.: Пер. с англ. Н. Мухин М.: ДМК Пресс, 2012. 494 с.
17. *Грецкий М.Н.* Структурализм (философ.) //Статья в Большой Советской Энциклопедии, 3-е изд., М.: Советская энциклопедия, 1969-1978.
18. *Елкин В.И.* Редукция нелинейных управляемых систем. Декомпозиция и инвариантность по возмущениям. М.: ФАЗИС, 2003. 207 с.
19. *Лаплас П.С.* Изложение системы мира М.: Наука, 1982. 676 с.
20. *Лаплас П.С.* Опыт философии теории вероятностей Пер. с фр., Изд.2, М.: URSS, 2011. 208 с.

21. Об основаниях геометрии //Сборник классических работ по геометрии Лобачевского и развитию ее идей /Ред. и вступ. статья *А.П. Нордена*, М.: Гос. изд. технико-теоретической литературы, 1956. 533 с.
22. *Осоргин А.Е.* AnyLogic 6. Лабораторный практикум Самара: ПГК, 2011. 100 с.
23. *Павловский Ю.Н.* Геометрическая теория декомпозиции и некоторые ее приложения. М.: ВЦ РАН, 2011. 93 с.
24. *Павловский Ю.Н., Смирнова Т.Г.* Проблема декомпозиции в математическом моделировании. М.: ФАЗИС, 1998. 272 с.
25. *Павловский Ю.Н., Смирнова Т.Г.* Введение в геометрическую теорию декомпозиции. М.: ФАЗИС, ВЦ РАН, 2006. 169 с.
26. *Пономарев И.Н.* Введение в математическую логику и роды структур: Учебное пособие. М.: МФТИ, 2007. 244 с.
27. *Фаулер М.* UML. Основы., 3-е издание. Пер. с англ. СПб: Символ-Плюс, 2004. 192 с.

Бродский Юрий Игоревич

Модельный синтез и модельно-ориентированное программирование

Подписано в печать 26.11.2013
Формат бумаги 60×84 1/16
Уч.-изд. л. 7. Усл.-печ. л. 9
Тираж 120 экз. Заказ 25

Отпечатано на ротапринтах
в Федеральном государственном бюджетном учреждении науки
Вычислительный центр им. А.А.Дородницына Российской академии наук
119333, Москва, ул. Вавилова, 40