Математические основы проектирования сложных систем в эпоху ИИ

Сергей Салищев, Центр ИИ СПбГУ

Аннотация

В работе рассматривается влияние стремительно растущей сложности компьютерных систем (в том числе систем, содержащих элементы искусственного интеллекта) на методы их проектирования. Показывается, что любой инженер, проверяющий свойства системы, фактически занимается математикой, пусть зачастую и неявно. Обосновывается невозможность эффективной разработки программы путём неформальных запросов ("программирования на промптах" для ИИ) без строгой математической базы. По аналогии с математическим программированием предлагается формальная модель математического проектирования, выражающая процесс проектирования через декомпозицию и абстракцию функции требований на языке теории категорий, сочетающая классические методы анализа и синтеза с методами абстрагирования и конкретизации, опирающаяся на теоретические границы вычислимости и надёжности. Вводятся формальные понятия эндофункторов абстракции и конкретизации на категории типов системы. Проводится сравнение с теорией абстрактной интерпретации на основе соединений Галуа. Дополнительным результатом работы является доказательство невозможности проектирования одновременно сложных и абсолютно надежных систем, демонстрация эквивалентности фундаментальных понятий ООП (инкапсуляция, наследование, полиморфизм) и аппликативностью функтора абстракции, а так же фундаментального преимущества ООП над ФП как модели проектирования с точки зрения вычислимости при разработке "достаточно хорошего" ПО. Рассматриваются сжимающие свойства и эффективность математических абстракций, подчёркивается роль фундаментальной математики (логики, теории алгоритмов, теории вероятностей, математического анализа и абстрактной алгебры) для обеспечения надёжности сложных дискретных систем на этапе проектирования.

1 Введение

Развитие искусственного интеллекта (ИИ) и связанных технологий приводит к автоматизации многих рутинных задач программирования. В результате, ценность инженера всё больше определяется его способностью решать задачи проектирования и архитектуры сложных систем. Однако включение ИИ в сами проектируемые системы и в процесс разработки приводит к взрывному росту их сложности. Из-за этого возрастает и сложность обеспечения их корректности и надёжности. Чтобы новые системы хотя бы сохраняли уровень надёжности предыдущего поколения, инженеру-проектировщику требуется глубокое понимание теоретических границ применяемых методов проектирования, формальной верификации и тестирования. В частности,

необходимо знать пределы вычислимости, связанные с комбинаторным взрывом, и понимать, какие проекты принципиально невыполнимы без новых подходов.

Один из ключевых тезисов работы – любой инженер, проверяя свойства системы или рассуждая о её поведении, явно или неявно занимается математикой. Проектирование сложных систем не может быть выполнено **ad hoc** методами без опоры на строгий математический аппарат. В эпоху ИИ возрастает соблазн пытаться "программировать на естественном языке", формулируя задачи для ИИ-систем. Но, как будет показано, это в принципе несостоятельно: даже если кажется, что ИИ сам догадался, что нужно сделать, без формального обоснования надёжность и предсказуемость такого проектирования не могут быть гарантированы.

Существует множество парадигм и неформальных подходов к проектированию систем. Одной из наиболее развитых методологий высокоуровневого проектирования сложных распределенных гетерогенных систем является методология Объектно-Ориентированного Проектирования Коада-Йордона [1], предшественница многих других подходов, например, UML [2]. Однако все они не опираются на строгие математические модели и не могут непосредственно рассматриваться как предшественники предлагаемой математической постановки задачи проектирования. Кратко опишем существующие подходы к **математизации проектирования систем**: К таким подходам можно отнести формальные методы спецификации и верификации программ [3], строгие методологии проектирования (например, "разработка через доказательство"). Но эти методы остаются в рамках дискретной математики и не могут компенсировать эффект комбинаторного взрыва состояний. Особняком стоит теория абстрактной интерпретации [4] использующая алгебру решеток и соединений Галуа на типах для описания архитектуры, однако она в большей степени ориентирована на проверку свойств систем и теорию компиляции. По сравнению с ними предлагаемый подход делает акцент на контролируемом переходе при проектировании систем от дискретных объектов к непрерывным и синтезе известных элементов математики, включая матанализ, в единую методологию, заполняющую пробелы в подготовке инженера-проектировщика. Математический аппарат работы наиболее близок к языку абстрактной интерпретации, при этом предположения корректности абстрактной интерпретации включаются как нижняя оценка, в то время как новая теория дает конструктивную верхнюю оценку потери качества в результате проектирования за счет введения инженерных калибровок анализа и абстракции. В этом заключается математическая новизна работы. При этом основная цель работы – не столько предложить радикально новый формализм, сколько интегрировать существующие результаты (из теории алгоритмов, логики, теории типов, вероятностных методов и т. д.) в целостную концепцию, которая позволяет заранее учитывать фундаментальные ограничения при проектировании сложных систем и доказывать результаты, эмпирически известные инженерам в виде строгих математических утверждений.

Перечислим основное содержание работы:

- Математическое определение сложной системы через понятие массовой задачи и рассмотрение проектирования как массовой задачи построения вычислимой аппроксимации требований.

- Доказательство неразрешимости общей задачи проектирования корректной сложной системы только методами анализа-синтеза из-за проблемы невычислимости в композиции.

- Анализ модели акторов (асинхронных агентов) и демонстрация того, что требование равномерной вычислимости на всех уровнях приводит к необходимости неограниченного недетерминизма (fairness), что эквивалентно возможности существования неконструктивной (сверхтьюринговой) машины.

- Введение функторов абстракции и конкретизации как формальной модели чередования приближений и уточнений при проектировании. Демонстрация того, что функтор абстракции сжимает необозримое множество состояний (возможно, невычислимую функцию) до вычислимой абстракции, а функтор конкретизации добавляет реализуемые детали с контролируемой погрешностью.

- Демонстрация того, что комбинация анализа-синтеза с абстракцией-конкретизацией позволяет обходить ограничения невычислимости ценой введения приемлемых погрешностей или вероятностных допущений.

- Сравнение с формализмом соединений Галуа в абстрактной интерпретации и демонстрация включения свойства включения абстрактной интерпретации как оценки снизу для качества системы, в то время как новая теория дает оценку сверху.

- Обсуждение эквивалентности математического доказательства и программы (изоморфизм Карри-Ховарда) и роли автоматизации доказательств. Указание на ограниченность автоматического доказательства теорем (барьер теоремы Гёделя) и рассмотрение подходов к его преодолению (трансфинитная индукция, наращивание системы аксиом и др.).

- Применение изложенного подхода к проектированию ПО и математике: сравнение ООП и ФП, параллельность, типизация, промпты ИИ, дискретная математика, матанализ, переход от дискретного к непрерывному и наоборот, анализ ограничений абстракции (корреляции, память, толстые хвосты распределений, человеческий фактор).

- Примеры абстракции и анализа с нетривиальной калибровкой.

2 Определение сложной системы и задача проектирования

Можно рассматривать проектирование как прикладное применение философии познания (гносеологии) к еще не созданной системе. Цель проектирования заключается во всестороннем познании системы для ее реализации. Основными методами познания являются анализ-синтез и абстракция-конкретизация. В философии основоположниками применения метода анализа-синтеза можно считать Аристотеля (IV в. до н. э.) и в новое время Рене Декарта «Рассуждении о методе» (1637). Метод абстракции-конкретизации, видимо, впервые был предложен и обоснован Платоном (V–IV в. до н. э.) и является сутью его идеалистической философии. В дальнейшем этот подход развивался философами идеалистами Иммануилом Кантом (XVIII в.), Георгом Гегелем (конец XVIII – начало XIX в.) и другими.

В дальнейшем обсуждается строгая математизация этих методов для целей проектирования сложных систем.

2.1 Массовые задачи и алгоритмы

Начнём с формализации понятий **сложная система** и **проектирование**. Вспомним концепцию **массовой задачи** из математической теории алгоритмов. Алгоритм, по определению, решает не один единичный экземпляр, а некоторый класс задач, различающихся входными данными. Другими словами, алгоритм предназначен для решения **массовой задачи** – совокупности всех задач данного типа с разными исходными данными. В классической советской школе алгоритмики (Колмогоров, Марков и др.) подчёркивалось свойство **массовости** алгоритма: «Алгоритм – это общий, единообразный, точно установленный способ решения любой задачи из данной массовой проблемы». Иначе говоря, алгоритм задаёт конечную процедуру, которая должна работать для любого входа из некоторого бесконечного множества.

Таким образом, систему можно рассматривать как решение некоторой массовой задачи, реализующее отображение $s : X \to Y$ для всех допустимых входных данных $x \in X$ и удовлетворяющее требованиям, заданным функцией $q(s) = 0$. **Программирование** (математическое программирование) при этом есть процесс конструктивного построения алгоритма (системы) s, который корректно решает поставленную массовую задачу. Отличие **сложной системы** от простой прежде всего количественное: система называется сложной, если множество её состояний или сценариев настолько велико, что не может быть полностью проверено любыми методами, то есть функция q не вычислима. С практической точки зрения, сложной обычно считают систему, состоящую из многих взаимодействующих компонентов, причём описание состояния системы экспоненциально велико по отношению к описанию самой системы.

Обратим внимание, что поскольку сложная система по определению содержит много вычислений, то ее эффективная практическая реализация будет **параллельной или распределенной**. Это соответствует наблюдениям о массовом переходе от последовательных систем к параллельным и распределенным по мере развития вычислительной техники и параллельного программирования.

Также обратим внимание, что если функции s или q заведомо не вычислимы, то алгоритма не существует, и задача программирования сформулирована некорректно (неразрешима). При этом, согласно теореме Райса [5], такая ситуация является более частой чем обратная. Поскольку конечная цель – решение произвольных практических задач, то доказательство невозможности точного решения не является приемлемым исходом. Нам требуются механизмы эффективной работы с невычислимыми системами и невычислимыми требованиями. Для этого введем два дополнительных этапа разработки систем: анализ требований и проектирование.

Определение 2.1 (Анализ требований)**.** *Задача перехода от невычислимой системы к вычислимой за счет аппроксимации функции требований q с точностью ε.*

Определение 2.2 (Проектирование)**.** *Задача нахождения вычислимой аппроксимации функции требований q с точностью ε.*

Этап	Система вычислима	Требования вычислимы
Анализ требований	−	−
Проектирование	+	−
Программирование	+	+

Таблица 1: Сравнение этапов по вычислимости системы и требований

Вычислимая аппроксимация функции требований q является **тестовым покрытием** системы. Такие определения этапов соответствуют методологии проектирования Коада-Йордана [1], в которой выделяются три основных вида деятельности при разработке:

1. Анализ требований
2. Проектирование как декомпозиция и анализ требований к компонентам
3. Программирование как нахождение алгоритма по спецификации компонент

Мы также можем рассматривать задачу проектирования как массовую. Для ее решения используется метод (алгоритм) анализа-синтеза, что приводит к разбиению системы в композицию модулей, каждый из которых решает свою подзадачу. Однако как мы увидим далее, композиция корректных компонентов не гарантирует автоматически корректность всей системы. Более того, могут возникать системные свойства, отсутствующие на уровне компонентов (так называемое **эмерджентное поведение системы**). Именно анализ и предотвращение неожиданных глобальных эффектов – ключевая задача математического проектирования систем.

Цель дальнейших рассуждений – формулировка и обсуждение общего алгоритма проектирования сложных систем как алгоритма решения массовой задачи.

2.2 Начальные предположения

Предположение 1 (Мировая симуляция)**.** *Пусть вся «мировая» динамика вычисляется на некоторой универсальной по Тьюрингу машине $\mathcal{M}$, обладающей неограниченной памятью и способностью исполнять любые алгоритмы, задаваемые конечной программой.*

Это предположение превращает тезис Чёрча-Тьюринга в теорему. Поскольку мощность симуляции не может быть выше мощности машины, на которой она запущена.

Если бы мировая вычислительная машина $\mathcal{M}$ обладала сверх-Тьюринговостью (гипервычислительной мощностью), она бы могла решать общие проблемы остановки и предсказывать с абсолютной точностью эволюцию любой термодинамической системы, что дало бы способ извлечь работу из теплового резервуара бесконечно много раз без вложения энергии, что эквивалентно построению *вечного двигателя второго рода* на макроуровне, существование которого противоречит второму началу термодинамики в формулировке Клаузиуса [6]. Таким образом, наше предположение не противоречит наблюдаемой физической реальности.

Предположение 2 (Абстракция бесконечности)**.** *Под бесконечностью понимается предельный процесс или формальный объект (например, $\aleph_0$), задающийся как предел монотонно возрастающих конечных величин. На практике любые физические измерения оперируют лишь конечными числами, и потребность в бесконечности возникает лишь как математическая удобная абстракция.*

Наблюдения в физике всегда имеют конечную точность: при дискретизации непрерывных сигналов сохраняется лишь конечный набор бит (теорема Котельникова–Шеннона [7]). Любая «непрерывная» функция фиксируется с шумом, а фактические данные представляют собой дискретный и конечный по объёму массив.

Невычислимость (неразрешимость) и неполнота формальных систем (теорема Гёделя) возникают лишь при допущении идеальных бесконечных ресурсов (лент, формул, доказательств). Поскольку в наших предположениях система с бесконечностью является абстракцией дискретной системы, то для этой дискретной системы неполнота и невычислимость её абстрактной модели выступают как признак неверного уровня абстракции и практической неприменимости к конечным вычислениям.

Поскольку в инженерной практике отсутствие решения интерпретируется как дефект постановки задачи, то с практической точки зрения неразрешимость и неполнота сигнализируют о том, что выбранная модель слишком слабая или имеет слишком низкий уровень абстракции. Для решения задачи следует ввести дополнительные предположения или подняться на более высокий уровень абстракции, представив ее в более простом виде.

2.3 Математическая постановка задач анализа и проектирования

Ограничим все рассматриваемые системы чистыми функциями $s : X \to Y$, где X – бесконечное множество. Такое рассмотрение не умаляет общности, поскольку для систем с памятью мы можем рассмотреть функцию перехода системы $s : X \to X$. Если система имеет скрытое состояние, то для нас функция перехода невычислима. Пример: детерминированная вычислимая функция перехода в системе помощи водителю (ADAS) должна включать модель автомобиля, дороги и самого водителя, что технически невозможно.

Но поскольку цель проектирования и заключается в работе с невычислимыми функциями, то это не должно представлять для нас концептуальной сложности, в отличие от подходов на основе дискретной математики и логики.

Пусть у нас есть описание желательного поведения $q : (X \to Y) \to \mathbb{R}, q \geq 0$ (спецификация), такая, что для корректной системы $s, q(s) = 0$.

Можно интерпретировать $q(s)$ как вероятность ошибки, то есть при $q(s) = 0$ система полностью удовлетворяет спецификации. Если функция q вычислима, то для нахождения s достаточно решить задачу оптимизации (математического программирования) $s = \arg\min q$, и никакое проектирование не требуется. Однако, для сложных систем по определению q не вычислима, и также для s может не быть вычислимых решений.

Это теоретическое обоснование того эмпирического факта, что **анализ-синтез сам по себе не решает проблему надёжности**. Подход, при котором систему пытаются построить целиком из корректных компонентов “снизу вверх” (анализируя требования, синтезируя компонентные решения и интегрируя их), наталкивается на **противоречие**: глобальное свойство (работоспособность всей композиции) может быть невыводимо из локальных, а попытка исследовать систему целиком приводит к комбинаторному взрыву. Если хоть одна из функций в композиции $f = g \circ h$ невычислима или не поддаётся полному анализу, то и вся f становится невычислимой или неопределённой в строгом смысле. В реальности даже при вычислимости всех компонент задача проверки сложной системы на все случаи эквивалентна перебору экспоненциально большого числа состояний, то есть неразрешима за обозримое время.

Заметим, что сама постановка задачи доказательства корректности весьма амбициозна – она подразумевает **полную** корректность. Поскольку для сложной системы

добиться этого невозможно, на практике ограничиваются частичной корректностью (если система завершает работу, то результат правильный) или более слабыми свойствами, и вместо строгого доказательства используют тестирование на некоторых примерах. Сформулируем задачи приближенного анализа и проектирования системы, что соответствует практике.

Определение 2.3 (Задача приближенного анализа требований сложной системы)**.** *Для задачи разработки системы $s : X \to Y$, требований $q(s) = 0$ и заданного ε построить модель окружения g и требования $q'(s') \leq \delta$ к вычислимой части системы s', такие что*

1. *Функция $g : X \times Y' \to Y$ – скрытая функция перехода (влияние внешних сил)*
2. *Существует вычислимая функция $s' : X \to Y'$, такая что $q'(s') \leq \delta$*
3. *Из $q'(s') \leq \delta$ следует, что $q(g(\cdot, s'(\cdot)) \leq \varepsilon$.*

Обратим внимание, что для получения оценок сверху вычислимость $q'(s')$ и $q(g(\cdot, s'(\cdot))$ не требуется.

Определение 2.4 (Задача приближенного проектирования сложной системы)**.** *Для задачи разработки системы $s : X \to Y$, требований $q(s) \leq \varepsilon$ построить требования к системе $q'(s) = 0$, такие что*

1. *Функция $q' : (X \to Y) \to \mathbb{R}, q' \geq 0$ вычислима*
2. *Существует вычислимая функция $s : X \to Y$, такая что $q'(s) = 0$*
3. *Из $q'(s) = 0$ следует, что $q(s) \leq \varepsilon$.*

То есть для заданной вероятности отказа или значения отклонения ε требуется найти вычислимую абстракцию спецификации с учетом неопределенности скрытого состояния системы. Введение параметра ε превращает саму задачу проектирования в массовую, таким образом, мы можем говорить об алгоритме проектирования в строгом смысле. При этом q' и является искомым **тестовым покрытием** системы.

Для полноты соответствия стандартной триаде этапов разработки рассмотрим в тех же терминах задачу математического программирования.

Определение 2.5 (Задача математического программирования сложной системы)**.** *Для задачи разработки системы $s : X \to Y$, требований $q(s) \leq \varepsilon$, тестового покрытия $q'(s) = 0$, найти такую оптимальную реализацию s^*, что*

$$\Omega = \{q'(s) = 0, s \in \mathrm{Hom}(X, Y)\},$$

$$s^* = \arg\min_{s \in \Omega} q(s).$$

Заметим, что в условиях ограниченных ресурсов, Ω – всегда конечное множество, поэтому если $\Omega \neq \varnothing$, то решение s^* существует.

Следует подчеркнуть, что эти рассуждения **не означают практической бесполезности анализа и синтеза**. Но нужно осознавать их **теоретические границы**. Никакая декомпозиция не позволяет превратить бесконечное множество случаев в

конечное. Дейкстра писал: «Тестирование может показать наличие ошибок, но никогда их отсутствие». Полное тестирование всех состояний и сценариев, что эквивалентно доказательству корректности, для сложной системы невозможно из-за экспоненциального или более быстрого роста числа случаев. Таким образом, необходим переход к **другим методам обеспечения надёжности** помимо анализа-синтеза. Далее мы увидим, что таким методом является введение абстракций, сжимающих пространство состояний до проверяемого, с последующей контролируемой конкретизацией.

2.4 Этика, эстетика и свобода творчества

В математике строгая формализация является необходимым шагом для автоматизации вычислений. Возникает вопрос: не приводит ли формализация проектирования к возможности исключения человека из процесса разработки сложных систем.

Однако мы можем строго указать неалгоритмизируемый шаг в нашем процессе, и это – анализ требований. Переформулировка задачи проектирования в виде массовой задачи обеспечивает существование как минимум переборного алгоритма ее решения. Однако наличие алгоритма не гарантирует ни его сходимости, ни существования непустого множества решений.

Обратим внимание, что в инженерной практике целью разработки является получение работоспособной системы. То есть, теорема несуществования, в отличие от математики, не является приемлемым результатом, даже если при данных требованиях система не может быть спроектирована. Аппроксимация требований происходит на этапе их анализа и заключается в поиске морально, эстетически и экономически допустимых условий, в которых решение существует.

При этом процесс разработки обычно включает многократные итерации по уточнению требований при столкновении с невозможностью реализации системы в процессе проектирования. Поскольку уточнение требований требует вычисления скрытой функции целеполагания человека, то мы можем заключить, что данная задача является ИИ-полной. Её алгоритмическое решение должно включать компонент, не отличимый по своим моральным, эстетическим и экономическим критериям принятия решений от человека. В этом и заключается роль творчества при разработке.

2.5 Модель акторов и вычислимость параллелизма

Одной из популярных моделей сложных систем является модель акторов (агентов), взаимодействующих посредством обмена сообщениями (асинхронно). Если мы потребуем, чтобы на всех уровнях системы (акторы, группы акторов, подсистемы) действовала однородность композиции, и вся система в целом была вычислимой и корректной, то мы придем к противоречию. **Однородность – возможность включить систему как актора в большую систему.** Для этого небходимо обеспечить **справедливость планирования** (fair scheduling), что приводит к необходимости **неограниченного недетерминизма**. **Справедливость планирования** предполагает, что если актор бесконечно долго посылает запросы другому, то рано или поздно каждый запрос будет обслужен (не произойдёт бесконечного голодания). Формально для этого в модель вводят недетерминистский выбор с потенциалом бесконечного постпостроения (например, недетерминированный генератор натуральных чисел). Такой недетерминизм выводит модель за границы машин Тьюринга. Спан с

соавторами показали, что машина с неограниченным недетерминизмом теоретически способна решать проблему остановки. Идея их алгоритма: один недетерминированный процесс выбирает некоторое число шагов N и симулирует машину Тьюринга на столько шагов, другой процесс постоянно увеличивает N; при справедливом чередовании рано или поздно будет протестировано достаточное число шагов, чтобы выявить остановку, если она случается.

Следствие: **любая реальная система, имитирующая модель акторов, не может абсолютно удовлетворять справедливости планирования**. В аппаратуре и программном планировщике всегда накладываются ограничения (приоритеты, квоты, таймауты и т.п.), иначе реализовать планирование за конечное время невозможно. Это означает, что реальная система должна либо виснуть, либо искусственно ограничивать допущения. Например, системные планировщики обычно предполагают, что задача завершит свою работу в разумные сроки, иначе её принудительно прерывают – это **ограниченный недетерминизм**. **Модель неограниченного недетерминизма служит как бы асимптотической абстракцией** идеального параллелизма, но недостижима в реальности. Инженер должен понимать: если при проектировании он пользуется моделью акторов со справедливым планированием, то полученные свойства (например, отсутствие взаимного блокирования) на практике выполняются в лучшем случае в статистическом смысле, то есть всё равно могут быть нарушены.

Таким образом, **на уровне архитектуры** выявляется принципиальный барьер: **сверхтьюринговые требования не реализуются на тьюринговых машинах**. В литературе по теории вычислений это иногда называют **machine augmented with fairness** – формально такая машина мощнее стандартной [8]. Из этого следует важный инженерный вывод: **при проектировании систем с параллелизмом и взаимодействием нужно закладывать отказоустойчивость к возможному нарушению справедливости планирования** (например, таймауты, механизмы восстановления), поскольку абсолютная справедливость недостижима. Мы видим здесь пример, как знание теоретического предела (неразрешимость проблемы остановки на детерминированной машине) на уровне абстракции оборачивается аналогичным пределом (абсолютная гарантия ответа от процесса не обеспечивается конечным планировщиком).

2.6 Эквивалентность моделей и пределы моделирования

Ещё одной распространённой инженерной практикой является моделирование системы на некотором упрощённом уровне (например, имитационное моделирование поведения, прототипирование) с целью проверки корректности. Основным постулатом математической теории моделей является **принцип эквивалентности моделей**: если модель ведёт себя “правильно”, то и реальная система, будучи моделью для той же аксиоматики, будет вести себя так же. Однако чем сложнее система, тем труднее построить её модель. Комбинаторный взрыв состояний резко ограничивает масштаб систем, доступных для полного моделирования. Формально число состояний растёт экспоненциально от числа переменных состояния, и быстро становится астрономическим. Это известная проблема при применении методик **верификации моделей** (model checking) – **state explosion problem**: малое увеличение числа компонентов может сделать исчерпывающий перебор состояний недостижимым ни вычислительно, ни по памяти.

Таким образом, **традиционная разработка, скрывающая проектирование за моделированием** (когда сначала строят модель, проверяют, затем "воплощают" её в коде) сталкивается с тем, что **модель сложной системы построить не проще, чем саму систему**. Если модель достаточно детальна, чтобы выявить все проблемы, то она почти столь же сложна, как оригинал, и проверка её корректности столь же трудна. Если же модель упрощена, то нет гарантии, что реальная система не поведёт себя иначе. Например, модель может не учитывать редких, но критичных событий (сбоев, перегрузок и т.д.), которые в реальности случатся.

Это не означает, что моделирование бесполезно – напротив, локальные модели (модели отдельных протоколов, компонентов) крайне важны. Но **общую модель всей сложной системы невозможно проверить с помощью локальных моделей**. Наш подход (см. далее) предполагает использование **абстракций и конкретизаций** вместо тотального моделирования: модель должна абстрагировать заведомо несущественные детали, чтобы уменьшить пространство состояний, но не потерять критических свойств. Это требует математического подхода к тому, что можно абстрагировать, а что нужно сохранить в модели.

Например, при анализе сетевого протокола можно абстрагироваться от конкретного размера буферов, но сохранить порядок сообщений; или можно абстрагироваться от порядка (считать, что пакеты приходят в среднем своевременно), но ввести вероятность потери. Выбор абстракции – искусство, но мы стремимся формализовать его как **функтор абстракции** в рамках некой категории моделей. Важно помнить: **любая абстракция – это компромисс**. Если абстракция нарушает условия применимости теорем (например, независимость событий для ЦПТ, линейность для суперпозиции), то прогноз модели может оказаться неверным на практике. Мы ещё вернёмся к примеру с нарушением условий Центральной предельной теоремы.

3 Методы абстракции и конкретизации в проектировании

3.1 Функторы абстракции и конкретизации: сжатие состояния и возвращение к реализации

Для преодоления описанных выше проблем (неразрешимость глобальной проверки, комбинаторный взрыв, необходимость сверхтюринговых машин) мы вводим в процесс проектирования циклы **абстракции** и **конкретизации**. Интуитивно, **абстракция** – это намеренное упрощение системы (отказ от некоторых требований или деталей, затрудняющих реализацию) для получения более общей модели. **Конкретизация** – обратный процесс добавления деталей, требуемых для реализации на практике.

Математически абстракцию можно понимать как некоторое отображение между реализуемой и нереализуемой моделями, которое сжимает множество состояний. На языке теории категорий абстракция может рассматриваться как **эндофунктор** $A : \mathcal{T} \to \mathcal{T}$ категории типов. Этот функтор должен быть **аппликативным**, то есть сохранять композицию: если g_1, g_2 – подсистемы (морфизмы), то $A(g_1 \circ g_2) = A(g_1) \circ A(g_2)$. Тогда абстрагирование можно проводить модульно. В объектно-ориентированном программировании, например, абстракция реализуется через интерфейсы и классы: класс задаёт интерфейс (абстракцию) для множества возможных реализаций.

Обратный процесс – **конкретизация** – соответствует переходу от абстрактной модели обратно к конкретной реализуемой системе. Конкретизация $E : \mathcal{T} \to \mathcal{T}$ правый сопряженный эндофунктор к функтору абстракции: $A \dashv E$, сопоставляющий конкретную реализацию абстрактным классам. В общем случае конкретизация не задаётся однозначно, поскольку абстракция теряет информацию. Конкретизация – это поиск какой-то реализации, согласующейся с абстрактной моделью.

Пример: числовые типы $3 \xrightarrow{A} \mathbb{R} \xrightarrow{E}$ FP32, абстрактные типы Dog $\xrightarrow{A} Animal \xrightarrow{E} (Dog + Cat + Cow + \ldots)$

Главное свойство абстракции – **бесконечное сжатие пространства состояний**. Абстракция отображает бесконечное множество S возможных состояний системы, возникающее в массовой задаче, в меньшее множество $X' = A(X)$, факторизуя бесконечные классы эквивалентных относительно абстракции состояний. Например, если в абстракции мы игнорируем адреса конкретных серверов, то различные состояния, различающиеся только этим адресом, схлопываются в одно абстрактное состояние. В идеале хорошая абстракция убирает "лишние" различия, сохраняя при этом ключевые свойства.

Можно провести аналогии с **теорией абстрактной интерпретации**: там вводятся отображения α (абстракция) и γ (конкретизация), образующие соединение Галуа между решётками конкретных и абстрактных свойств. При этом $\alpha \circ \gamma$ приближённо равна тождественному, и $\gamma(\alpha(S)) \subseteq S$. По сути, α сжимает множество свойств, а γ расширяет абстрактный результат до конкретного (с некоторой потерей точности). Другой аналогией является квантование вычислений, в котором выделяют функции квантования значений $Q : \mathbb{R} \to \mathbb{N}$ и деквантования $Q : \mathbb{N} \to \mathbb{R}$. Абстракция является аналогом квантования, конкретизация – деквантования.

В ходе абстракции, если полная спецификация $f(x)$ невычислима, мы ищем вычислимую функцию $g(x)$, которая аппроксимирует f. Функтор абстракции действует здесь на уровне функций: $f \mapsto g$. Требуется, чтобы g была вычислима на машине Тьюринга, но достаточно близка к f по значению (например, отличается с допустимой погрешностью или вероятностью ошибки). Таким образом, абстрагирование невычислимой функции предоставляет теоретически вычислимую функцию-абстракцию, которую уже можно практически реализовывать с помощью конкретизации. Этот принцип используется, например, в численных методах: вместо символического точного решения (которое может не выражаться в элементарных функциях) берут вычислимую аппроксимацию с нужной точностью промежуточных вычислений.

Еще одним примером является введение модели угроз и гарантий в том случае, если не может быть абсолютно надёжной распределённой системы: допускается, скажем, что не более 2 узлов выйдут из строя (абстрагируемся от худших случаев), а при этих условиях алгоритм обеспечивает согласованность данных. Эта абстракция (допущение о 2 сбоях) облегчает построение решения, но инженер должен понимать, что случится при 3 сбоях – система нарушит свойства. Поэтому на следующем шаге надо либо физически гарантировать крайне малую вероятность 3 одновременных сбоев (например, пространственным разнесением, резервированием – т.е. предпринять меры на уровне конкретизации), либо признать этот сценарий вне области требований.

3.2 Формализация задачи математического проектирования, комбинация анализа-синтеза с абстракцией-конкретизацией

Обратим внимание, что мы можем считать функции s, q морфизмами категории типов системы $\mathcal{T}$. Таким образом, мы можем непосредственно применить к ним функторы абстракции и конкретизации.

Предположение 3 (Категориальный контекст)**.** *Пусть категория типов системы – бикартезиански замкнутая категория (CCC) $\mathcal{T}$ с начальный объект 0, терминалом 1, суммой $(+, 0)$, произведением $(\times, 1)$, экспонентами Y^X (обозначаем $X \Rrightarrow Y$) и дистрибутивностью.*

Все неравенства для требований формулируются только на $\mathbb{R}_{\geq 0}$ со стандартным порядком (в интерпретации – "меньше-лучше"). Будем считать, что категория типов содержит все используемые типы, поэтому расширение $\mathcal{T}$ в процессе проектирования не требуется.

Интуиция: можно говорить, что $X \Rrightarrow Y$ – это тип морфизмов $X \to Y$, произведение и экспоненты требуются для аппликативности абстракции, сумма требуется для определения конкретизации как надтипа суммы конкретных типов, склеенных абстракцией.

Определение 3.1 (Аппликативный функтор)**.** *Эндофунктор $A : \mathcal{T} \to \mathcal{T}$ вместе с естественными преобразованиями*

$$\mathsf{pure}_X : X \to AX, \qquad \mathsf{ap}_{X,Y} : A(X \Rrightarrow Y) \times AX \to AY,$$

удовлетворяющими следующим законам:

$$\begin{aligned}
\textbf{\textit{(тождественность)}} \quad & \mathsf{ap}_{X,X}\big(\mathsf{pure}(\mathrm{id}_X),\, v\big) \;=\; v, \\
\textbf{\textit{(композиция)}} \quad & \mathsf{ap}\Big(\mathsf{ap}\big(\mathsf{ap}\big(\mathsf{pure}(\circ),\, u\big),\, v\big),\, w\Big) \;=\; \mathsf{ap}\big(u,\, \mathsf{ap}(v, w)\big), \\
& \qquad\qquad u \in A(Y \Rrightarrow Z),\; v \in A(X \Rrightarrow Y),\; w \in AX, \\
\textbf{\textit{(гомоморфизм)}} \quad & \mathsf{ap}\big(\mathsf{pure}(f),\, \mathsf{pure}(x)\big) \;=\; \mathsf{pure}\big(f(x)\big), \qquad f \in X \Rrightarrow Y,\; x \in X, \\
\textbf{\textit{(перенос)}} \quad & \mathsf{ap}\big(u,\, \mathsf{pure}(x)\big) \;=\; \mathsf{ap}\big(\mathsf{pure}(\lambda f.\, f\, x),\, u\big), \qquad u \in A(X \Rrightarrow Y),\; x \in X.
\end{aligned}$$

Здесь $\circ : (Y \Rrightarrow Z) \Rrightarrow ((X \Rrightarrow Y) \Rrightarrow (X \Rrightarrow Z))$ – внутренняя композиция.

Определение 3.2 (Абстракция как аппликативный и типонезависимый идемпотентный подъём морфизмов)**.** *Эндофунктор абстракции $A : \mathcal{T} \to \mathcal{T}$ снабжён структурой аппликативного функтора 3.1. Это задаёт* типонезависимый подъём алгоритмов

$$\alpha_{X,Y}(f) \;:=\; \big(x \mapsto \mathsf{ap}(\mathsf{pure}\, f,\; x)\big) :\; AX \to AY,$$

естественный по X, Y и функториальный: $\alpha(\mathrm{id}) = \mathrm{id},\;\; \alpha(g \circ f) = \alpha(g) \circ \alpha(f)$.

Абстракция идемпотентна

$$A \circ A = A.$$

Определение 3.3 (Конкретизация)**.** *$E : \mathcal{T} \to \mathcal{T}$ – функтор конкретизации. Даны естественные трансформации*

$$\zeta_X : AX \to EX, \qquad \theta_X : EX \to X.$$

Для $\sigma : AX \to AY$ определим «понижение» (конкретизацию морфизмов)

$$\eta_X \;:=\; E(\mathsf{pure}_X) \circ \zeta_X \circ \mathsf{pure}_X :\; X \to EAX,$$

$$\gamma_{X,Y}(\sigma) \;:=\; \theta_Y \;\circ\; E(\sigma) \;\circ\; \eta_X \;:\; X \to Y.$$

Считаем, что конкретизация "без потерь" на образе A:

$$\theta_{AX} \circ \zeta_{AX} : A^2X \to X,$$

$$\theta_{AX} \circ \zeta_{AX} = \mathrm{id}_{AX} \qquad (\forall X).$$

Типосогласованность выполняется по идемпотентности абстракции $A^2 = A$.

Теорема 1 (Каноническая стрелка из суммы конкретных типов в конкретизацию абстрактного типа)**.** *Пусть заданы семейство $(X_i)_{i\in I}$ в $\mathcal{T}$ и $\bar{X}$ такие, что $AX_i = \bar{X}$ для всех i. Тогда для каждого i стрелка*

$$\iota_i \;:=\; \eta_{X_i} \;=\; E(\mathsf{pure}_{X_i}) \circ \zeta_{X_i} \circ \mathsf{pure}_{X_i} \;:\; X_i \longrightarrow E(AX_i) = E\bar{X}$$

определена корректно, и по универсальному свойству копроизведения существует единственная

$$\iota \;:\; \coprod_{i\in I} X_i \;\longrightarrow\; E\bar{X}$$

такая, что $\iota \circ \mathrm{in}_i = \iota_i$ для всех $i \in I$, где in_i – это каноническое вложение копроизведения

$$\mathrm{in}_i :\; X_i \longrightarrow \coprod_{i\in I} X_i \qquad (i \in I).$$

Интуиция: конкретизация абстракции множества конкретных типов – это надтип суммы этих типов.

Определение 3.4 (Анализ и синтез)**.** *$D, S : \mathcal{T} \to \mathcal{T}$ – функторы, моделирующие структурное разбиение/сборку (последовательная/параллельная композиция через $(\times, 1)$).*

Даны естественные преобразования

$$\psi^D :\; \mathrm{Id}_{\mathcal{T}} \Rightarrow S \circ D \qquad и \qquad \phi^D :\; D \circ S \Rightarrow \mathrm{Id}_{\mathcal{T}}.$$

интерпретируемые как «включение после анализа» и «свёртка после синтеза».

Считаем, что анализ и синтез "без потерь" на образе D:

$$\phi^D_{DX} \;\circ\; D(\psi^D_X) \;=\; \mathrm{id}_{DX} \qquad (\forall X).$$

$$S(\phi^D_Y) \;\circ\; \psi^D_{SY} \;=\; \mathrm{id}_{SY} \qquad (\forall Y).$$

Предположение 4 (Связи и совместимости)**.** *Предполагаем естественные связи*

$$D \circ S \circ D = D, \qquad A \circ E \circ A = A,$$

и естественную совместимость синтеза и конкретизации

$$S \circ E \;\cong\; E \circ S \quad (\textit{естественный изоморфизм}).$$

Интуиция: анализ обращается синтезом без потерь на своем образе; абстракция обращается конкретизацией без потерь на своем образе;синтез и конкретизация коммутируют без потерь.

Определение 3.5 (Анализ на уровне A)**.** *Существуют естественные преобразования*

$$\sigma_A : \; A \Longrightarrow (D \circ A), \qquad \pi_A : \; (D \circ A) \Longrightarrow A,$$

то есть для каждого X: $\sigma_{A,X} : AX \to (DA)X$, $\quad \pi_{A,X} : (DA)X \to AX$, естественные по X.

Определение 3.6 (Синтез на уровне A)**.** *Для любого морфизма компонент $\tau : (D \circ A)X \to (D \circ A)Y$ определим*

$$S_A[\tau] \; := \; \pi_{A,Y} \; \circ \; \tau \; \circ \; \sigma_{A,X} \quad : \quad AX \longrightarrow AY.$$

Лемма 1 (Естественность и функториальность S_A)**.** *Для $\tau_1 : (DA)X \to (DA)Y$ и $\tau_2 : (DA)Y \to (DA)Z$ имеем $S_A[\tau_2 \circ \tau_1] = S_A[\tau_2] \circ S_A[\tau_1]$. Если $\pi_A \circ \sigma_A = \mathrm{Id}_A$, то $S_A[\mathrm{Id}_{(DA)X}] = \mathrm{Id}_{AX}$.*

Определение 3.7 (Требования)**.** *Базовые требования – это (в общем случае невычислимый) функционал*

$$q : \; \mathrm{Hom}(X, Y) \; \to \; \mathbb{R}_{\geq 0}.$$

На промежуточных уровнях (A, $D \circ A$, ...) определены соответствующие (в общем случае невычислимые) подъёмы $q_\bullet$ требований. Выполнение требований является условием корректности системы.

Предположение 5 (Неулучшаемость качества при $E \circ A$)**.** *Для любого $f : X \to Y$ верно $q(f) \; \leq \; q\big(\gamma(\alpha_{X,Y}(f))\big)$.*

Это соответствует предположению корректности Кюсо в абстрактной интерпретации [4] $id \leq \gamma\alpha$.

Предположение 6 (Калибровки шага абстракции и анализа)**.** *Потери учитываются только в абстракции A и анализе D: существуют монотонные функции*

$$\rho_A : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, \qquad \kappa_D : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0},$$

такие, что для всех допустимых $\sigma : AX \to AY$ и $\tau : (D \circ A)X \to (D \circ A)Y$

$$\textbf{\textit{(A)}} \qquad q\big(\gamma(\sigma)\big) \; \leq \; \rho_A\big(q_A(\sigma)\big),$$

$$\textbf{\textit{(D)}} \qquad q_A\big(S_A[\tau]\big) \; \leq \; \kappa_D\big(q_{D \circ A}(\tau)\big).$$

Синтез S и конкретизация E выполняются без потерь (дополнительных калибровок нет).

Замечание (смысл калибровок). Калибровки – это инженерная экстраполяция качества модели на качество реальной системы. При $\rho_A = \kappa_D = \mathrm{Id}$ мы закладываемся, что наши модели соответствуют наихудшим оценкам качества. Это может приводить к крайне консервативным оценкам на практике. При нетривиальных калибровках мы можем использовать ослабленные модели, например, использовать при оптимизации L_2 норму вместо L_∞. Несмотря на то, что калибровки можно скрыть в q, с точки зрения наглядности это не кажется хорошей идеей. Калибровки сами могут быть эмпирически измеренными функциями и уточняться в процессе проектирования. Также наличие явных калибровок позволяет идентифицировать, является ли причиной потери точности слишком консервативная модель или несоответствие модели и системы, и сократить усилия при проектировании.

Определение 3.8 (Калибровка блока и цепочки)**.** *Чередующаяся цепочка итераций проектирования имеет вид*

$$\Pi \;=\; (D_m \circ A_m) \;\circ\; \cdots \;\circ\; (D_1 \circ A_1).$$

$$\Sigma \;=\; (S_m \circ E_m) \;\circ\; \cdots \;\circ\; (S_1 \circ E_1).$$

Для блока «абстракция+анализ» $(D_i \circ A_i)$ *положим калибровку блока*

$$T_i \;:=\; \rho_{A_i} \;\circ\; \kappa_{D_i} \;:\; \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}.$$

Её единая калибровка – упорядоченная композиция

$$\Lambda_\Pi \;:=\; T_m \;\circ\; T_{m-1} \;\circ\; \cdots \;\circ\; T_1,$$

(здесь и далее Π*- уровень абстракции, на котором задача проектирования разрешима), калибровки не коммутируют.*

Интуиция: цепочка итераций моделирует итеративный процесс проектирования, когда сначала происходит проектирование на уровне всей системы, потом проектирование отдельных узлов и т. д.

Определение 3.9 (Семейство $S_A^{[R]}$)**.** *Для любого эндофунктора* $R : \mathcal{T} \to \mathcal{T}$ *определим*

$$\sigma_A^{[R]} :=\; \sigma_A \circ \mathrm{Id}_R :\; A \circ R \Rightarrow (D \circ A) \circ R, \qquad \pi_A^{[R]} :=\; \pi_A \circ \mathrm{Id}_R :\; (D \circ A) \circ R \Rightarrow A \circ R,$$

и для $\tau : (DA \circ R)X \to (DA \circ R)Y$ *положим*

$$S_A^{[R]}[\tau] \;:=\; \pi_{A,Y}^{[R]} \;\circ\; \tau \;\circ\; \sigma_{A,X}^{[R]} \;:\; (A \circ R)X \to (A \circ R)Y.$$

Определение 3.10 (Глобальный синтез S_Π)**.** *Обозначим «DA-хвосты»*

$$R_0^{DA} := \mathrm{Id}, \qquad R_i^{DA} := (D_i \circ A_i) \circ \cdots \circ (D_1 \circ A_1) \quad (i \geq 1),$$

и $A^{\mathrm{tot}} := A_m \circ \cdots \circ A_1$*. Определим*

$$S_\Pi \;:=\; S_{A_m}^{[R_{m-1}^{DA}]} \;\circ\; S_{A_{m-1}}^{[R_{m-2}^{DA}]} \;\circ\; \cdots \;\circ\; S_{A_1}^{[R_0^{DA}]} \;:\; \mathrm{Hom}(\Pi X, \Pi Y) \to \mathrm{Hom}(A^{\mathrm{tot}} X, A^{\mathrm{tot}} Y).$$

Определение 3.11 (Частичный понижающий оператор)**.** *Пусть* $R : \mathcal{T} \to \mathcal{T}$ *– эндофунктор,* A *– префикс композиции* $A \circ R$*, даны* pure*,* $\zeta : A \Rightarrow E$ *и* $\theta : E \Rightarrow \mathrm{Id}$*. Положим для любого* X

$$\eta_{RX} := E(\mathsf{pure}_{RX}) \circ \zeta_{RX} \circ \mathsf{pure}_{RX} :\; RX \to E\,A(RX),$$

и

$$\tau_{RY} := \theta_{RY} \circ \zeta_{RY} :\; A(RY) \to RY.$$

Тогда

$$\Gamma^{[R]}(\sigma) \;:=\; \tau_{RY} \;\circ\; \theta_{A(RY)} \;\circ\; E(\sigma) \;\circ\; \eta_{RX} \quad :\quad RX \longrightarrow RY,$$

где $\sigma : A(RX) \to A_i(RY)$*.*

Определение 3.12 (Глобальная конкретизация γ_Π)**.** *Обозначим «A-хвосты»*

$$A^{(0)} := \mathrm{Id}, \qquad A^{(k)} := A_k \circ A_{k-1} \circ \cdots \circ A_1 \quad (k \geq 1),$$

так что $A^{\mathrm{tot}} = A^{(m)}$*. Для* $\sigma : A^{\mathrm{tot}}X \to A^{\mathrm{tot}}Y$ *определим последовательность*

$$\sigma_m := \sigma, \qquad \sigma_{i-1} := \Gamma_i^{[A^{(i-1)}]}(\sigma_i) \quad : \quad A^{(i-1)}X \to A^{(i-1)}Y \quad (i = m, m-1, \ldots, 1).$$

Тогда $\gamma_\Pi(\sigma) := \sigma_0 : X \to Y$*, т.е.*

$$\gamma_\Pi \;=\; \Gamma_1^{[A^{(0)}]} \;\circ\; \Gamma_2^{[A^{(1)}]} \;\circ\; \cdots \;\circ\; \Gamma_m^{[A^{(m-1)}]}.$$

Определение 3.13 (Понижение всей цепочки)**.**

$$\mathcal{L}_\Pi \;:=\; \gamma_\Pi \;\circ\; S_\Pi \;:\; \mathrm{Hom}(\Pi X, \Pi Y) \;\longrightarrow\; \mathrm{Hom}(X, Y).$$

Замечание (сепарабельность). После фиксации архитектуры системы итерации проектирования можно «схлопнуть» в единые функторы S_Π и E_Π (неявно задан γ_Π). Итеративная структура остаётся в калибровке Λ_Π, что делает оценки качества консервативными на этапе проектирования; консервативность может быть снята при решении задачи математического программирования.

Определение 3.14 (Вычислимая оценка на уровне П)**.** *Для порога* $r \in \mathbb{R}_{\geq 0}$ *и функции*

$$Q_\Pi :\; \mathrm{Hom}(\Pi X, \Pi Y) \;\to\; \mathbb{R}_{\geq 0}$$

будем говорить, что оценка $Q_\Pi(\tau) \;\leq\; r$ *вычислима, если* ***индикаторная функция***

$$I_Q(\tau;\, r) \;:=\; \begin{cases} 1, & Q_\Pi(\tau) \;\leq\; r, \\ 0, & Q_\Pi(\tau) \;>\; r, \end{cases}$$

вычислима. Интуиция: Функция $1 - I_Q$ *является подъемом тестового покрытия на уровень* П*.*

Определение 3.15 (Подъём свидетеля (тестовый адаптер))**.** *Задан* вычислимый частичный *оператор*

$$\mathsf{lift}_\Pi \;:\; \mathrm{Hom}(X, Y) \;\rightharpoonup\; \mathrm{Hom}(\Pi X, \Pi Y)$$

такой что, если $\mathsf{lift}_\Pi(s)$ *определён, то*

$$\mathcal{L}_\Pi\big(\mathsf{lift}_\Pi(s)\big) \;=\; s.$$

Интуитивно: тестовый адаптер пытается «поднять» реализацию s *до абстрактного представления; если требуемые интерфейсы не реализованы, подъём* не определён*.*

Определение 3.16 (Универсальная функция тестового покрытия)**.** *Для* $s : X \to Y$ *и порога* $r \geq 0$ *положим*

$$I_Q^{\mathrm{sys}}(s;\, r) \;:=\; \begin{cases} I_Q\big(\mathsf{lift}_\Pi(s);\; r\big), & \text{если } \mathsf{lift}_\Pi(s) \text{ определён}, \\ 0, & \text{иначе (требуемые интерфейсы отсутствуют)}, \end{cases}$$

и определим ***функцию тестового покрытия***

$$q'(s) \;:=\; 1 - I_Q^{\mathrm{sys}}(s;\, r) \;\in\; \{0, 1\}.$$

Тогда тесты всегда *определены для любой* s*: если абстрактные интерфейсы отсутствуют, тесты* падают *(значение* $q'(s) = 1$*).*

Задача 1 (категориальное ε-проектирование)**.** *Дан целевой допуск* $\varepsilon \in \mathbb{R}_{\geq 0}$ *и требования* $q(s) \leq \varepsilon$*. Выбрать цепочку* Π*, построить мажоранту требований* Q_Π *и найти* свидетеля $\tau^* : \Pi X \to \Pi Y$ *и* $r = \varepsilon_\Pi$ *такие, что*

$$Q_\Pi(\tau^*) \;\leq\; \varepsilon_\Pi, \qquad \varepsilon_\Pi \;:=\; \sup\{\, u \geq 0 \;\mid\; \Lambda_\Pi(u) \leq \varepsilon \,\}.$$

Тогда $q'(s) := 1 - I^{\mathrm{sys}}_Q(s;\, \varepsilon_\Pi)$ *– искомые вычислимые требования к системе.*

Теорема 2 (Корректность решения)**.** *Если существует* s*, такой что* $q'(s) = 0$ *(т.е.* $Q_\Pi(\tau^*) \leq \varepsilon_\Pi$*), то*

$$q(s) \;\leq\; \Lambda_\Pi\big(Q_\Pi(\tau^*)\big) \;\leq\; \Lambda_\Pi(\varepsilon_\Pi) \;\leq\; \varepsilon.$$

Таким образом, выполнение тестов $q'(s) = 0$ *обеспечивает требуемый допуск* $q(s) \leq \varepsilon$ *и* q' *является решением задачи проектирования из определения 2.4.*

Доказательство. По построению $q'(s) = 0$. □

Замечание (прототипирование). Тестовое покрытие $q'(s)$ содержит внутри себя информацию об архитектуре системы в виде функции lift_Π. То есть невозможно протестировать систему как черный ящик: несмотря на то, что задача проектирования формулируется в терминах спецификаций и требований, архитектура самой системы является необходимой частью её решения. Также для того чтобы конструктивно проверить, что решение задачи проектирования не пусто, требуется свидетель τ^*, являющийся прототипом системы на уровне абстракции Π. Это обосновывает необходимость прототипирования (имитационного/математического моделирования) в процессе проектирования сложных надежных систем.

3.3 Алгоритм математического ε проектирования

Далее рассмотрим комбинированный итерационный алгоритм проектирования, сочетающий шаги абстракции и декомпозиции системы.

Алгоритм 1 (категориальное ε-проектирование через цепочку чередующихся абстракций и декомпозиций)**.**

Вход: *допуск* $\varepsilon \in \mathbb{R}_{\geq 0}$*, невычислимые требования* $q(s) \leq \varepsilon$*.*
Выход: *вычислимая функция требований* $q'(s) = 0$*.*
Инициализация: $k := 0$, $\Pi_0 := \mathrm{Id}$, $\Lambda_{\Pi_0} := \mathrm{Id}$.

1. ***Завершение по несовместности требований.*** *Если* $k < 0$ *завершить с ошибкой.*

2. ***Расширение цепочки.*** *Выбрать новый блок* (D_{k+1}, A_{k+1})*, продлить*

$$\Pi_{k+1} := (D_{k+1} \circ A_{k+1}) \circ \Pi_k, \qquad \Lambda_{\Pi_{k+1}} := (\rho_{A_{k+1}} \circ \kappa_{D_{k+1}}) \circ \Lambda_{\Pi_k}.$$

3. ***Откат на уровень.*** *Если невозможно выбрать блок* (D_{k+1}, A_{k+1})*, положить* $k \leftarrow k-1$*, перейти к шагу 1.*

4. ***Проверка вычислимости.*** *Построить* $r = \varepsilon_{\Pi_{k+1}}$*, свидетеля (прототип)* τ^* *и вычислимый индикатор* $I^{\mathrm{sys}}_Q(s;\, r)$ *(вычислимый предикат* $Q_{\Pi_{k+1}}(\tau^*) \leq r$*).*

5. ***Продолжение текущей итерации.*** *Если* $\{I_Q^{\text{sys}}(\cdot;\, r) = 1\} = \varnothing$, *вернуться к шагу 1.*

6. ***Успех.*** *Вернуть требования* $q'(s) = 1 - I_Q^{\text{sys}}(s;\, r)$

7. ***Иначе – углубление.*** *Положить* $k \leftarrow k+1$, *перейти к шагу 1.*

Этот алгоритм может рассматриваться как математическая формализация алгоритма проектирования в методологии Коада-Йордона [1].

Обратим внимание, что сходимость алгоритма для произвольных задач не гарантирована, что подтверждается практическим опытом проектирования, которое иногда заканчивается неудачей. С другой стороны, применение континуальных абстракций (матанализа) в большинстве случаев позволяет сокращать размерность задачи достаточно быстро. Также обратим внимание на глубокий **синергетический эффект между абстракцией и анализом-синтезом**. Например, переход к численным моделям на верхнем уровне системы упрощает её дальнейшую декомпозицию.

Этот подход не даёт абсолютных гарантий корректности, но в сложных системах их вообще невозможно получить ни одним методом (мы рассматривали теоретические преграды). Однако предлагаемая комбинация методов позволяет заранее предусмотреть и структурировать зоны неопределённости. Вместо того, чтобы хаотично оставлять пробелы (“тут потом подумаем, тут протестируем”), явное введение абстракций заставляет фиксировать все упрощения конкретно. Это стимулирует закладывать компенсации на этапе проектирования: например, если допущена погрешность, закладываем запас прочности; если допущено падение узла, добавляем механизм восстановления.

3.4 Связь программ и доказательств

Интересно осмыслить описанный подход с точки зрения **математической логики**. В формальном смысле, построение программы, корректной по спецификации, равносильно доказательству теоремы (что для любого входа x верно $S(x) = f(x)$). В соответствии с изоморфизмом Карри–Ховарда [9], любой программе (функции) можно сопоставить логическое утверждение (тип), а её выполнению – доказательство этого утверждения. Если представить гипотетическую “мировую вычислительную машину”, на которой выполняются все возможные программы (что эквивалентно допущению, что физический мир Тьюринг-полон), то утверждения математики можно трактовать как спецификации, а доказательства – как программы, которые на этой машине устанавливают истинность утверждений. Формальная математика сама является решением массовой задачи: вывода утверждений из аксиом. Из этого философского взгляда следует, что **вся математика накоплена как решение задач проектирования доказательств** (правил вывода одних фактов из других). Таким образом, **все математики являются программистами, а все программисты математиками**. Этот взгляд подтверждается современным доказательством теоремы Гёделя о неполноте путем сведения логического вывода в аксиоматике Пеано к запуску программы на машине Тьюринга, при этом теорема сводится к проблеме остановки, которая неразрешима.

Важный методологический вывод: при проектировании сложных систем не следует пренебрегать строгим математическим образованием инженеров. Те области

математики, которые преподаются стандартно (логика, дискретная математика, теория вероятностей, математический анализ, алгебра), вовсе не являются абстрактно ненужными, наоборот, они представляют собой проверенные временем обобщения массовых инженерных задач и методов их решения. При этом математические абстракции, обладающие сжимающими свойствами, работают как функтор абстракции в смысле проектирования.

Таким образом, **анализ-синтез** и **абстракцию-конкретизацию** в совокупности можно рассматривать как две оси проектирования. Анализ-синтез отвечает за структурную, композиционную часть (разделение задачи на этапы, модули и их сборка). Абстракция-конкретизация отвечает за содержательную, семантическую часть (адаптация решаемых задач к методам решения и обратное уточнение). Вместе они образуют **механизм решения неразрешимых в точности задач за счёт переформулировки их в решаемом виде с погрешностью**. По сути, это единственный способ "решать неразрешимое": отказаться от части требований, превратив проблему в разрешимую, и получить приближённое решение. Мы делаем это постоянно (например, вместо поиска точного решения NP-трудной задачи используем жадный алгоритм, получающий субоптимальный результат). Предлагаемый подход – лишь систематизация данного процесса, выведение его на уровень осознанного метода проектирования. Именно это соображение отражено в двух проекциях (иерархия наследования и владения) ООП.

При этом именно механизм абстракции отвечает за мощность математики, поскольку обеспечивает асимптотическое сжатие типов. Таким образом, суть математики заключается во введении более мощных и эффективных абстракций (за что многие ругали Александра Гротендика), в то время как построение доказательств путем комбинации существующих методов является более механистическим и сравнительно легко автоматизируемым.

3.5 Автоматизация доказательств и проектирования

В финале раздела отметим перспективы и ограничения **автоматизации** описанного процесса. Казалось бы, если программы и доказательства эквивалентны, то можно поручить системе автоматизации доказательств (theorem prover) проверить корректность системы или даже спроектировать её (поставив задачу как теорему о существовании нужной функции). В некоторой степени это реализуется: например, системы Coq, Isabelle умеют генерировать программный код из доказательств, а авто-теоремы могут доказывать отдельные свойства. Но теорема Гёделя о неполноте гласит, что никакой одной формальной системы не хватает, чтобы доказать все истинные утверждения о естественных числах. Значит, и универсального алгоритмического проектировщика с фиксированной формальной системой быть не может. Всегда останутся системы сложнее, чем может анализировать данный авто-проектировщик.

Тем не менее, **барьер Гёделя можно обходить, усиливая систему рассуждений**. Например, для вывода некоторых неразрешимых в арифметике утверждений достаточно добавить **трансфинитную индукцию** до определённого ординала. В терминах проектирования, если базовая система аксиом (предположений о системе) недостаточна для доказательства надёжности, можно **расширить аксиоматику** дополнительными абстракциями, т.е. принять дополнительные постулаты, сделав модель более вычислительно мощной и абстрактной. В математике это делает, например, теория множеств: добавляя аксиомы существования больших кардиналов,

получаем возможность доказать утверждения, независимые от исходной системы. Аналогично, Тарский предложил подход к определению истины: нельзя определить истину для языка L внутри L (теорема Тарского), но можно перейти на метаязык L' и там ввести предикат $True(x)$, выражающий "x истина языка L". Этот переход к метаязыку – тоже **абстракция с последующей конкретизацией**: мы признаём, что внутри системы нам утверждение не выразить, потому рассматриваем систему извне.

Для инженера это означает: **сложные системы требуют более абстрактных моделей и инструментов**. Невозможно ограничиться, скажем, только типизацией или только тестами. Нужно сочетать средства, поднимаясь на уровень мета-моделей, в том числе привлекая средства матанализа и геометрии, когда исходные модели исчерпаны. **Метод абстракции-конкретизации, рассмотренный в этой работе, – это и есть способ оперировать на мета-уровне**. Мы признаём, что проектирование корректной сложной системы внутри фиксированной парадигмы всегда упрётся в ограничение, и разрешаем себе выйти из нее, и переформулировать задачу в новой. Это как прибавить новую аксиому, не противоречащую требованиям, но делающую задачу решаемой. При этом необходимо контролировать допущения, а также возможность их нарушения в реальной системе. Задачу перехода на мета-уровень также можно решить в рамках средств авто-проектирования. Это означает, что такой авто-проектировщик должен уметь варьировать существующие и, возможно, создавать в процессе работы новые абстракции.

Однако, даже при переходе на мета-уровень при фиксированных требованиях, как это обычно бывает на практике, ответ системы авто-проектирования почти всегда будет отрицательным. Поскольку процесс разработки – это поиск морально, эстетически и экономически приемлемого компромисса о требованиях к системе, что является ИИ-полной задачей.

3.6 Эффективность математических абстракций

Наиболее эффективными на практике являются математические абстракции, то есть применение математических, в том числе числовых и функциональных типов в качестве абстрактных типов значений функтора абстракции. Числовые абстракции обладают очевидным сжимающим действием, связанным с размерностью множества чисел. Мы можем выделить абстракции типа $\aleph_0$ связанные с бесконечной продолжимостью последовательности шагов и пределом числовой последовательности. Также можно выделить абстракции типа $\aleph_1$ связанные с бесконечной непрерывной делимостью пространства и времени. Также теория множеств дает понимание того, что абстракции типа $\aleph_1$ являются более мощными, чем абстракции типа $\aleph_0$. Видимо, впервые необходимость и сила таких абстракций для решения задач познания были замечены Зеноном (V век до н. э.) в его апориях "Ахилес и черепаха" и "Стрела".

Развитие математики идет в направлении разработки все более мощных абстракций, обладающих более сильными сжимающими свойствами. Так в матанализе были разработаны абстракции гладкости и бесконечной гладкости, которые позволяют абстрагировать не отдельные числа, а целые функции, заменяя их на линейные приближенные модели. Еще одной мощной абстракцией является представление о выпуклой функции, позволяющее заменять ее, на ее глобальный экстремум. Разделы математики: алгебраическая геометрия и топологическая алгебра вводят абстракции, позволяющие сжимать целые пространства до дискретных алгебраических структур.

Особого упоминания заслуживает «Необъяснимая эффективность математики в естественных науках» (Юджин Вигнер [10]). Это означает, что математические абстракции, кроме сжатия, обеспечивают еще и эффективное сохранение важных для проектирования свойств. Основная заслуга в этом принадлежит эргодичности, Центральной Предельной Теореме и принципу наименьшего действия. Но эти свойства наблюдаемой реальности сами являются следствиями ограниченной вычислительной мощности мировой вычислительной машины.

4 Связь математического проектирования и абстрактной интерпретации

Сравним новый формализм математического проектирования с результатами Кюсо по абстрактной интерпретации [4].

4.1 Абстрактная интерпретация и соединения Галуа

Сначала напомним основные положения теории абстрактной интерпретации.

Определение 4.1 (Абстрактная интерпретация.)**.** *Пусть* $(C, \leq_C)$ *– конкретный домен свойств/состояний с монотонной семантикой* $f : C \to C$ *(например, «собирающей»/операционной), а* $(A, \leq_A)$ *– абстрактный домен (упрощённые свойства).* Абстрактная интерпретация *– это выбор пары монотонных отображений*

$$\alpha : C \to A \quad \textit{(абстракция)}, \qquad \gamma : A \to C \quad \textit{(конкретизация)},$$

и абстрактного трансформера $g : A \to A$*, который корректно аппроксимирует* f*, т.е.*

$$\alpha \circ f \ \leq_A \ g \circ \alpha \quad \textit{(эквивалентно } f \circ \gamma \ \leq_C \ \gamma \circ g\textit{)}.$$

Такая g *называется* абстрактной семантикой f *на* A*.*

Определение 4.2 (Соединение Галуа.)**.** *Монотонные* $\alpha : C \to A$ *и* $\gamma : A \to C$ *образуют* соединение Галуа*, если*

$$\alpha(c) \ \leq_A \ a \quad \Longleftrightarrow \quad c \ \leq_C \ \gamma(a) \qquad (\forall c \in C, \ a \in A).$$

Эквивалентно:

$$\mathrm{id}_C \ \leq_C \ \gamma \circ \alpha, \qquad \alpha \circ \gamma \ \leq_A \ \mathrm{id}_A,$$

что в точности означает в категориальных обозначениях

$$\alpha \dashv \gamma.$$

Если $\alpha \circ \gamma = \mathrm{id}_A$*, то это* вставка Галуа *(insertion).*

Связь между абстрактной интерпретацией и соединением Галуа. Если (α, γ) – соединение Галуа, то стандартная конструкция

$$f^\sharp \ := \ \alpha \circ f \circ \gamma \ : \ A \to A$$

даёт *наилучшее корректное приближение* (Best Correct Approximation, BCA):

$$\alpha \circ f \ \leq_A \ f^\sharp \circ \alpha, \qquad \text{и если } g : A \to A, \ \alpha \circ f \leq_A g \circ \alpha, \ \text{то } f^\sharp \leq_A g.$$

Тем самым, *соединение Галуа обеспечивает канонический выбор* абстрактной семантики.

Неподвижные точки и корректность. Для монотонного f (и $f^\sharp$) при выполнении условий теоремы Тарского о неподвижных точках [11]

$$\alpha\big(\operatorname{lfp}(f)\big) \;\leq_A\; \operatorname{lfp}\big(f^\sharp\big), \qquad \operatorname{gfp}\big(f^\sharp\big) \;\leq_A\; \alpha\big(\operatorname{gfp}(f)\big),$$

т.е. вычисленная на A абстрактная наименьшая неподвижная точка корректно оценивает сверху поднятие конкретной и аналогично для наибольшей неподвижной точки. Равенства достигаются при условиях *полноты* (напр., α сохраняет нужные супремумы или $\alpha \circ f = f^\sharp \circ \alpha$).

В теории компиляции решетка на типах естественным образом возникает при анализе свойств циклических переменных в программных циклах и рекурсии. Предпорядок соответствует расширению/сужению типа переменной при переходе к следующей/предыдущей итерации. Максимальная и минимальная неподвижная точки соответствуют минимальному и максимальному типу циклической переменной. Эта техника используется в различных приложениях, таких как оптимизация кода, формальная верификация, автоматическая генерация тестов и др.

Интерпретация корректности. Неравенство $\mathrm{id}_C \leq \gamma\alpha$ выражает, что цепочка «абстракция, затем конкретизация» даёт не более точный результат в C (оценка сверху); с другой стороны, $\alpha\gamma \leq \mathrm{id}_A$ – сжатие в A. Эти два неравенства – каноническая форма корректности абстракции в подходе Кюсо; конструкция $f^\sharp = \alpha f \gamma$ – её оптимальная (в смысле порядка) реализация.

4.2 Согласованность понятий абстракции и конкретизации

Для согласования наших функторов абстракции/анализа и конкретизации/синтеза с абстракцией и конкретизацией в абстрактной интерпретации докажем адъюнктность $A \dashv E$ и $D \dashv S$.

Теорема 3 (Адъюнкция $A \dashv E$)**.** *Функтор абстракции A (определение 3.2) левосопряжён конкретизации E (определение 3.3) при допущении 4.*

Доказательство. Определим единицу и коединицу:

$$X \xrightarrow{\mathsf{pure}_X} AX \xrightarrow{\zeta_X} EX \xrightarrow{E(\mathsf{pure}_X)} EAX$$

$$AEY \xrightarrow{\zeta_{EY}} EEY \xrightarrow{\theta_{EY}} EY \xrightarrow{\theta_Y} Y$$

$$\eta_X \;:=\; E(\mathsf{pure}_X) \circ \zeta_X \circ \mathsf{pure}_X :\; X \to EAX, \qquad \varepsilon_Y \;:=\; \theta_Y \circ \theta_{EY} \circ \zeta_{EY} :\; AEY \to Y.$$

Обе семьи стрелок естественны по X, Y.

(I) Первый треугольник.

$$\begin{array}{ccc} AX & \xrightarrow{A\eta_X} & AEAX \\ {\scriptstyle \mathrm{id}_{AX}}\big\downarrow & & \big\downarrow{\scriptstyle \varepsilon_{AX}} \\ AX & = & AX \end{array}$$

Нужно показать $\varepsilon_A \circ A\eta = \mathrm{id}_A$, т.е. для любого X:

$$\varepsilon_{AX} \circ A(\eta_X) = \mathrm{id}_{AX}.$$

Вычисляя и применяя естественность ζ, θ:

$$\begin{aligned}\varepsilon_{AX} \circ A(\eta_X) &= \big(\theta_{AX} \circ \theta_{EAX} \circ \zeta_{EAX}\big) \circ \big(AE(\mathsf{pure}_X) \circ A(\zeta_X) \circ A(\mathsf{pure}_X)\big) \\ &= \theta_{AX} \circ \big(\theta_{EAX} \circ EA(\mathsf{pure}_X)\big) \circ \big(\zeta_{AX} \circ A(\mathsf{pure}_X)\big) \\ &= \theta_{AX} \circ \zeta_{AX} \;=\; \mathrm{id}_{AX},\end{aligned}$$

где в последнем равенстве использовано отсутствие потерь конкретизации на образе A.

(II) Второй треугольник.

$$\begin{array}{ccc} EY & \xrightarrow{\eta_{EY}} & EAEY \\ {\scriptstyle \mathrm{id}_{EY}}\downarrow & & \downarrow{\scriptstyle E\varepsilon_Y} \\ EY & = & EY \end{array}$$

Нужно показать $E\varepsilon \circ \eta_E = \mathrm{id}_E$, т.е. для любого Y:

$$E(\varepsilon_Y) \circ \eta_{EY} = \mathrm{id}_{EY}.$$

Разворачивая определения и пользуясь естественностью ζ, θ:

$$\begin{aligned}E(\varepsilon_Y) \circ \eta_{EY} &= E(\theta_Y) \circ E(\theta_{EY}) \circ E(\zeta_{EY}) \circ E(\mathsf{pure}_{EY}) \circ \zeta_{EY} \circ \mathsf{pure}_{EY} \\ &= \zeta_Y \circ A(\theta_Y) \circ A(\theta_{EY}) \circ A(\mathsf{pure}_{EY}) \circ \mathsf{pure}_{EY} \\ &= \zeta_Y \circ A\big(\theta_Y \circ \theta_{EY} \circ \mathsf{pure}_{EY}\big) \circ \mathsf{pure}_{EY}.\end{aligned}$$

в последнем равенстве используем отсутствие потерь конкретизации на образе A (то есть $\theta_{AEY} \circ \zeta_{AEY} = \mathrm{id}_{AEY}$), что диаграммно свернет внутреннюю композицию к id_{AEY}; тогда

$$E(\varepsilon_Y) \circ \eta_{EY} = \zeta_Y \circ \mathrm{id}_{AEY} \circ \mathsf{pure}_{EY} = \zeta_Y \circ \mathsf{pure}_{EY} = \mathrm{id}_{EY}.$$

Оба треугольника коммутативны, следовательно, $A \dashv E$.

□

Теорема 4 (Адъюнкция $D \dashv S$)**.** *Функтор анализа D из определения 3.4 левосопряжён синтезу S , т.е. существует естественная биекция*

$$\Phi_{X,Y} : \mathrm{Hom}(DX, Y) \;\cong\; \mathrm{Hom}(X, SY) \qquad (\textit{естественно по } X, Y).$$

Доказательство. Возьмём естественные преобразования (единицу и коединицу адъюнкции)

$$X \xrightarrow{\psi_X} SDX \qquad DSY \xrightarrow{\phi_Y} Y$$

удовлетворяющие треугольным тождествам

$$\phi_{DX} \circ D(\psi_X) = \mathrm{id}_{DX}, \qquad S(\phi_Y) \circ \psi_{SY} = \mathrm{id}_{SY}.$$

$$\Phi_{X,Y}(f) \;:=\; S(f) \circ \psi_X \;:\; X \to SY, \qquad \Psi_{X,Y}(g) \;:=\; \phi_Y \circ D(g) \;:\; DX \to Y.$$

Проверка $\Psi \circ \Phi = \mathrm{id}$. Для $f : DX \to Y$:

$$DX \xrightarrow{D(\psi_X)} DSDX \xrightarrow{\phi_{DX}} DX \quad = \quad DX \xlongequal{\quad} DX$$

$$\Psi_{X,Y}(\Phi_{X,Y}(f)) = \phi_Y \circ D\big(S(f)\circ\psi_X\big) = \underbrace{\big(\phi_Y \circ DS(f)\big)}_{\text{естественность } \phi} \circ D(\psi_X) = f\circ\phi_{DX}\circ D(\psi_X) = f\circ \mathrm{id}_{DX} = f.$$

Проверка $\Phi \circ \Psi = \mathrm{id}$. Для $g : X \to SY$:

$$SY \xrightarrow{\psi_{SY}} SDSY \xrightarrow{S(\phi_Y)} SY \quad = \quad SY \xlongequal{\quad} SY$$

$$\Phi_{X,Y}(\Psi_{X,Y}(g)) = S(\phi_Y\circ Dg)\circ\psi_X = S(\phi_Y)\circ \underbrace{SD(g)\circ\psi_X}_{\psi_{SY}\circ g \quad (\text{естественность } \psi)} = S(\phi_Y)\circ\psi_{SY}\circ g = \mathrm{id}_{SY}\circ g = g.$$

Естественность Φ, Ψ по X, Y стандартна и следует из естественности ψ, ϕ и функториальности D, S. □

4.3 Сравнение формализма математического проектирования и соединений Галуа

Построим соединение Галуа на основе порядка на требованиях. Обратим внимание, что модель абстрактной интерпретации не различает абстракцию и декомпозицию, поэтому схлопнем наши цепочки композиций до пары функторов Π, Σ. Поскольку композиция сохраняет адьюнктность (Маклейн [12], раздел 4.8), $\Pi \dashv \Sigma$. Калибровкой Π является Λ_Π. Для простоты ограничимся рассмотрением одной пары абстракции конкретизации $A \dashv E$ из определений 3.2, 3.3 с калибровкой ρ из предположения 6 и естественными связями 4.

Определение 4.3 (Псевдопорядки по качеству)**.** *Для фиксированных X, Y зададим предпорядки на множествах гомоморфизмов:*

$$f \ \preceq_C \ g \ := q(f) \le q(g), \qquad \sigma \ \preceq_A \ \tau \ := q_A(\sigma) \le q_A(\tau).$$

Аппликативный подъём посредством абстракции даёт

$$\alpha^\# := \alpha_{X,Y} : \mathrm{Hom}(X, Y) \to \mathrm{Hom}(AX, AY),$$

а понижение посредством конкретизации –

$$\gamma^\# : \mathrm{Hom}(AX, AY) \to \mathrm{Hom}(X, Y),$$

$$\gamma^\#(\sigma) := \theta_Y \circ E(\sigma) \circ \eta_X.$$

Предположение 7 (Неулучшаемость ($\mathrm{id} \le \gamma\alpha$))**.** *Для любого $f : X \to Y$ выполняется $q(f) \le q\big(\gamma^\#(\alpha^\#(f))\big)$.*

Предположение 8 (Метрика качества (оценка сверху потерь абстракции))**.** *Существует монотонная $\rho_A : \mathbb{R}_{\ge 0} \to \mathbb{R}_{\ge 0}$ такая, что для всех $\sigma : AX \to AY$ $q\big(\gamma^\#(\sigma)\big) \ \le \ \rho_A\big(q_A(\sigma)\big)$.*

Лемма 2 (Слабая связь Галуа с калибровкой ρ_A)**.** *Если* $\alpha^{\#}(f) \preceq_A \sigma$, *то*

$$q(f) \;\le\; q\big(\gamma^{\#}(\alpha^{\#}(f))\big) \;\le\; \rho_A\big(q_A(\alpha^{\#}(f))\big) \;\le\; \rho_A\big(q_A(\sigma)\big),$$

то есть $\alpha^{\#}(f) \preceq_A \sigma \;\Rightarrow\; \gamma^{\#}(\alpha^{\#}(f)) \;\preceq_C^{\rho_A}\; \gamma^{\#}(\sigma)$.

Доказательство. Левая оценка – Допущение 7. Средняя – Допущение 8 при $\sigma := \alpha^{\#}(f)$. Правая – монотонность ρ_A и определение $\preceq_A$. □

Теорема 5 (Переход к классическому соединению Галуа при $\rho_A = \mathrm{id}$)**.** *Пусть* $\rho_A = \mathrm{id}$ *и для всякого* f *существует* наилучшая корректная абстракция $\alpha^{\#}(f) = \inf\{\sigma \in \mathrm{Hom}(AX, AY) \mid f \preceq_C \gamma^{\#}(\sigma)\}$ *(инфимумы берутся относительно* $\preceq_A$*). Тогда для всех* f, σ

$$\alpha^{\#}(f) \;\preceq_A\; \sigma \quad\Longleftrightarrow\quad f \;\preceq_C\; \gamma^{\#}(\sigma),$$

т.е. $\alpha^{\#} \dashv \gamma^{\#}$ *образуют классическое соединение Галуа по Кюсо [4] на предпорядках* $(\mathrm{Hom}(X, Y), \preceq_C)$ *и* $(\mathrm{Hom}(AX, AY), \preceq_A)$.

Доказательство. ($\Rightarrow$) Из Леммы 2 при $\rho_A = \mathrm{id}$: $q(\gamma^{\#}(\sigma)) \ge q(\gamma^{\#}(\alpha^{\#}(f))) \ge q(f)$, т.е. $f \preceq_C \gamma^{\#}(\sigma)$. ($\Leftarrow$) По определению $\alpha^{\#}(f)$ как инфимума множества $\{\sigma \mid f \preceq_C \gamma^{\#}(\sigma)\}$. □

Следствие 1 (Вставка Галуа на образе абстракции)**.** *Если дополнительно выполнены* $A \circ A = A$ *и* $\theta_{AX} \circ \zeta_{AX} = \mathrm{id}_{AX}$ *(все* X*), то при* $\rho_A = \mathrm{id}$ *имеем*

$$\alpha^{\#} \circ \gamma^{\#} \;=\; \mathrm{id} \quad \textit{на } \mathrm{Im}(\alpha^{\#}),$$

т.е. получаем Галуа-вставку на образе абстракции.

Доказательство. Для $\sigma = \alpha^{\#}(f)$ $\alpha^{\#}(\gamma^{\#}(\sigma)) = \alpha\big(\theta \circ E(\sigma) \circ \eta\big)$ сворачивается в σ по треугольникам адъюнкции $A \dashv E$ (доказанным ранее) и равенству $\theta_{AX} \circ \zeta_{AX} = \mathrm{id}_{AX}$; идемпотентность A обеспечивает типосогласованность. □

Комментарий. Теорема 5 воспроизводит корректность/оптимальность в абстрактной интерпретации; Лемма 2 даёт её количественное ослабление с калибровкой ρ_A.

5 Абстракция от дискретного к непрерывному

Одной из особенностей нашего метода является явное использование связи между дискретными (конечными) объектами и непрерывными (бесконечными, гладкими) моделями. Во многих случаях **абстракция заключается в переходе к непрерывной модели**, с последующим возвращением к дискретной реализации. Рассмотрим общие механизмы такого перехода.

5.1 Абстракция бесконечности, предел и непрерывность

Любой компьютер оперирует конечными структурами (память конечна). Непрерывные объекты (вещественные числа, гладкие функции) изначально недоступны дискретной машине – их приходится дискретизировать. Однако, именно **введение понятий предела и непрерывности было мощнейшей абстракцией математики**, позволившей описывать бесконечные процессы конечными формулами. Вспомним: $\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = 1$ – хотя суммируется бесконечное число членов, запись понятна и даст **точное** значение.

Непрерывность – это **идеализированное свойство**, в природе всё дискретно (атомарно) по крайней мере на квантовом уровне. Но моделирование дискретной системы обычно невычислимо сложно, а абстракция в виде дифференциальных уравнений или распределений вероятностей дает практические прогнозы. Например, газ – дискретное скопление молекул, а в термодинамике мы считаем давление и температуру непрерывными параметрами, средними по огромному числу частиц.

С точки зрения теории алгоритмов, непрерывность – **невычислимая** абстракция: задать произвольное вещественное число – это задать бесконечную последовательность цифр. Однако, работая с непрерывными моделями, мы можем получить результаты, которые затем **дискретно конкретизируются с погрешностью**. Так строятся численные методы решения уравнений: сначала выписали интеграл (абстракция), потом приближённо посчитали его суммой (конкретизация).

Выдающееся свойство **сжимающего отображения абстракции** (например, предела) – **асимптотическая сходимость**. Если процесс обладает свойством сжатия (скажем, сумма рядов быстро сходится), то ограниченное приближение даст малую ошибку. Поэтому **при вычислимости абстрактной модели можно получить управляемую ошибку при реализации**. Это и есть основа математического проектирования.

5.2 От суммы к интегралу: переход к площади

В классическом анализе хорошо показано, как переход от дискретного к непрерывному облегчает решение задач. Например, нужно найти сумму $1+2+\cdots+n$. Разбив отрезок на n единиц, можно интегрировать гладкую функцию $f(x) = x$ на $[0, n]$: $\int_0^n x, dx = n^2/2$. Это даёт оценку суммы (которая равна $n(n+1)/2$) с точностью порядка $O(n)$. Но более хитро: интеграл – это главная составляющая суммы, а недостающую для точности часть можно учесть, интегрируя разность между ступенчатой функцией и линейной. Так или иначе, использование интеграла упростило задачу.

В общем случае, **методы дифференциального и интегрального исчисления – это мощные средства абстракции**, позволяющие вместо дискретного многоступенчатого процесса изучать гладкую кривую. В программировании аналогом является замена циклической итерации непрерывной оптимизацией. Например, обучение нейросети: вместо перебора дискретных вариантов весов применяют метод градиентного спуска (решают дифференциальное уравнение оптимизации), что гораздо эффективнее.

Конечно, потом необходимо возвратиться к дискретному. Градиентный спуск, скажем, реализуется конечными шагами малой длины. Здесь выступает параметр ε: точность приближения интеграла суммой. Этот параметр – мост между дискретным и непрерывным. Чем меньше шаг, тем ближе дискретный процесс к непрерывному решению, но тем больше нужно итераций. Инженер выбирает ε, исходя из допуска на погрешность и имеющихся ресурсов.

5.3 Хаотическая динамика и эргодичность как источник случайности

Почему непрерывное поведение часто описывается статистически как случайное? **Динамические системы**, строго детерминированные по законам, могут демонстрировать **хаотическое поведение**: экспоненциальную чувствительность к начально-

му состоянию. Если у нас нет безграничной точности, предсказать далекое будущее невозможно: микроскопическая ошибка разрастётся. Такие системы мы рассматриваем как **стохастические** на макроуровне. **Эргодическая гипотеза** (в газовой динамике, статистической физике) утверждает, что за достаточно долгое время система проходит все доступные ей состояния, и поэтому временное среднее равно усреднению по пространству состояний. Это даёт обоснование, почему хаос можно моделировать **вероятностно**.

Для нас важно: **случайность часто является абстракцией детерминированности, которая сложнее, чем решение вероятностной задачи**. Вместо моделирования триллионов молекул, мы считаем давление как среднее, полагая параметры молекул "случайно" распределёнными. Детерминированная система играет роль генератора случайностей. Орнштейн (1989) доказал, что определённые детерминированные преобразования (например, "бейкерово отображение") по статистическому поведению эквивалентны подбрасыванию монеты. То есть классическая физическая система может давать последовательность исходов, не отличимую от случайного процесса (с точки зрения наблюдателя без знания начальных условий).

В инженерных приложениях это означает, что **можно смело заменять сложный хаотический процесс на вероятностную модель**, если известны её статистические свойства. Это колоссально упрощает проектирование. Вместо тысяч дифференциальных уравнений мы пишем пару параметров распределения. Конечно, при этом возможны редкие события, но мы можем оценить их вероятность.

5.4 Вероятности как типы, нечеткая логика

Когда вводится вероятность, формальная логика тоже подвергается обобщению. Появляются **вероятностные типы** (типы, зависящие от распределений значений) и **нечёткие логики**, где утверждения имеют "степень истинности". Можно рассматривать их как более общую парадигму доказательств: доказательство теперь носит характер, скорее, статистического обоснования. Тем не менее, сохраняется аналогия с программами: **вероятностные программы соответствуют доказательствам в вероятностной логике**. Есть результат, связывающий логическое истолкование доказательств с вероятностными комбинаторными объектами (например, соответствие между системами типов с монадой распределения и категорией полиномиальных функторов, где морфизмы - вероятностные трансформации).

В практике программирования **вероятностные типы** появляются, например, через аннотации `nullable` (значение присутствует с какой-то вероятностью) или типы, обёрнутые в `Option`/`Maybe` (значение либо есть, либо нет). Фактически, это 1-битная вероятность: либо 0, либо 1. Если расширить концепцию, можно говорить: **"эта функция возвращает корректный результат с вероятностью не менее 0.999"**. Это можно явно выразить в нечеткой логике, получив вероятностные спецификации.

С точки зрения соответствия Карри–Ховарда, каждое такое утверждение соответствует некоему программному объекту, который достижим с определённой вероятностью. Например, **нечёткое доказательство** 0.999 истинности утверждения P – это программа, которая в 99.9% случаев завершится правильно, а в 0.1% случаев может упасть. Если применить вероятностную абстракцию и перейти к распределениям вероятностей, то выполнение такой программы без ошибок станет доказательством вероятностных свойств исходной системы.

5.5 Центральная предельная теорема: переход к нормальному распределению

Стохастический аналог предельного перехода – **центральная предельная теорема (ЦПТ)**. Она говорит, что сумма большого числа слагаемых с произвольным распределением (с конечной дисперсией) стремится к нормальному (гауссовскому) распределению. Это удивительное свойство: множество разных дискретных процессов на высоком уровне начинают описываться одной универсальной функцией. **Для инженера это благо: не нужно изучать детали распределения, можно приблизить его нормальным законом**.

Конечно, ЦПТ – **асимптотическая абстракция**. Она хороша при больших n. Для умеренных n могут быть отклонения (например, распределение Стьюдента). Но даже при этом отклонения от нормального распределения не велики и быстро убывают с ростом выборки.

ЦПТ лежит в основе проектирования систем массового обслуживания (СМО). Если запросы клиентов являются независимыми, то мы можем аппроксимировать распределение вероятностей нормальным и использовать гладкую сжимающую абстракцию для упрощения проектирования такой системы.

Инженер, не знакомый с ЦПТ, вынужден либо переоткрывать её эмпирически, либо неверно оценивать риски, например, думать, что если распределение времени работы каждого узла не нормально, то и система будет сильно не нормальна. Обычно это не так. Пример: время отклика сервиса, состоящего из 100 микросервисов, суммируется из времён прохождения каждого. Если они относительно независимы, то по ЦПТ суммарное время будет близко к нормальному распределению, независимо от конкретного распределения задержки в каждом микросервисе. Это позволяет, например, вычислять вероятность того, что отклик превысит заданный порог, просто зная среднее и дисперсию каждого этапа.

ЦПТ – типичный пример выгодной абстракции: мы заменяем реализацию системы нормальным распределением и получаем приемлемо точный результат при куда меньших усилиях.

5.6 Свёртка с нормальным распределением: сглаживание дискретного процесса через канал с шумом

Ещё один аспект: **любое дискретное распределение сглаживается, если его наблюдать через канал с ограниченной пропускной способностью**. Реальная измерительная система всегда имеет конечное разрешение и шумы. Поэтому сигнал с частыми скачками (дискретный импульсный) на выходе фильтра будет более гладким (квази-непрерывным).

В информатике это тоже проявляется: если передавать дискретные события (0/1) через среду с помехами, на принимающей стороне имеем вероятность ошибок – нормальное распределение вокруг реального значения. В итоге при усреднении получаем гладкую кривую.

Таким образом, **ограниченность канала действует как оператор свёртки с неким ядром (примерно гауссовским)**. Переходя к пределу большой пропускной способности, мы это ядро делаем дельта-функцией, и опять получаем чёткие импульсы. Но в реальности предела не существует, все физические наблюдения при комнатной температуре всегда чуть сглажены. Поэтому **часто инженер может**

считать, что отклик системы гладкий, даже если внутри система дискретная, потому что наблюдение происходит через усреднение.

Практический пример: **пиксельное изображение vs восприятие глазом**. Глаз имеет конечное разрешение, и хотя изображение – дискретная сетка, мы видим почти непрерывно, особенно если с достаточно большого расстояния. Значит, можно использовать математический анализ (вводить производные, контуры) при обработке изображений, игнорируя зернистость.

5.7 Рандомизация и ε-точные алгоритмы: внесение непрерывности для упрощения

Ранее мы отмечали, что **случайность – полезная абстракция**. В программировании это проявляется в **рандомизированных алгоритмах**: они могут давать решение быстрее, хотя и с ненулевой вероятностью ошибки. Например, тест Миллера–Рабина для проверки простоты: он за полиномиальное время определяет составность или простоту числа с вероятностью ошибки 2^{-k} (можно сделать сколь угодно малой) за счёт использования случайных оснований. Детерминированный алгоритм аналогичной скорости не известен. Здесь **случайность выступает аналогом непрерывности** – по сути, интегрирование по случайно выбранным точкам пространства даёт информацию о всём пространстве, чего невозможно добиться дискретно.

Идея **ε-точных алгоритмов**: разрешим алгоритму ошибаться с вероятностью ε. Тогда часто можно добиться большой экономии. Этот подход лежит, например, в основе **machine learning**: алгоритм обучается, но не гарантирует 100% классификацию, однако 98% его устраивает. И с этой поблажкой задача, которая алгоритмически NP-трудна (оптимальная классификация), решается приближенно (градиентным спуском по функции потерь) с отличным результатом.

Выходит, **внесение "непрерывности" в задачу (через вероятность ошибки или случайность) понижает её вычислительную сложность**. Снова пример: задача о максимальной клике – NP-полная. Но если искать клику, которая меньше максимальной не более чем на 5 узлов (т.е. примерно максимум), можно использовать эвристики и за полиномиальное время получать очень хорошие решения на реальных графах.

В нашем подходе это вполне узаконено: мы изначально предлагаем смириться с тем, что полная точность недостижима, и формулировать задачу с погрешностью, тогда вместо неразрешимости мы получаем вполне решаемую приближённую задачу.

5.8 Гомотопическая теория типов: модель непрерывности в логике

В заключение раздела отметим новейшую математическую теорию, **гомотопическую теорию типов (HoTT)** [13]. Она также сочетает дискретную логическую структуру (теория типов, изначально – чисто комбинаторная) с понятиями непрерывности (гомотопия – непрерывное деформирование). В HoTT равенство элементов типа трактуется как существование непрерывного пути между ними. Формально, типы рассматриваются как топологические пространства, а доказательство равенства $x = y$ – как путь (гомотопия) от x до y. Это позволяет переносить интуицию из топологии (например, что путь можно "плавно" трансформировать, что возникают высшие пути между путями и т.д.) на строго формализованную логику типов.

Практически, HoTT – попытка **свести непрерывность к дискретной логике на более высоком уровне абстракции**. Для нас же важен сам факт: **математика стремится объединить дискретное и непрерывное в единую парадигму**. Это подтверждает, что наш подход имеет глубокие основания.

6 Конкретизация от непрерывного к дискретному

Обсудим теперь механизмы "конкретизации" непрерывных объектов обратно в дискретные конструкции.

6.1 Формулы и типы как дискретные объекты; символьные вычисления как абстрактная интерпретация

Программный код, логическая формула, тип данных – всё это **дискретные сущности**. Однако они могут интерпретироваться как представления непрерывных или бесконечных объектов. Например, формула $\forall x\ \exists y\ :\ x^2 = y$ описывает свойство бесконечного множества чисел. Мы используем конечную строку символов, чтобы отразить бесконечное отношение. Это возможно благодаря **символьным вычислениям**: вместо перебора значений, мы оперируем символами по синтаксическим правилам. Символьные методы – это **абстрактная интерпретация**, поскольку они манипулируют не конкретными данными, а формулами о них.

В идеале, **символьное решение** задачи – наиболее ценное, ибо оно часто покрывает бесконечное множество случаев. Например, решение квадратного уравнения даётся формулой, и нам не нужно решать заново для каждого набора коэффициентов. Стремление к символьным решениям – это попытка продвинуться как можно дальше, не переходя к вычислениям, а преобразуя формулы. Во многих системах (САПР, компиляторы) есть шаги оптимизации, где вместо тупого вычисления заранее символически упрощают выражения.

6.2 Апроксимация бесконечного (невычислимого) объекта

Символьное решение не всегда существует: например, не существует формулы для корней произвольного полинома степени ≥ 5 (теорема Абеля–Руффини). Приходится решать задачу численно. То есть, была абстрактная идея (найти выражение), она столкнулась с преградой – мы делаем шаг конкретизации (итерационный метод нахождения корней с нужной точностью).

Если полностью точного решения нет, применяют **аппроксимации**. То есть, мы не выражаем $f(x)$, но говорим: $f(x) \approx g(x)$ при некоторых условиях. Это уже конкретизация: вместо исходной функции мы взяли более простую (скажем, асимптотическое разложение, отбросив высшие члены). Техника асимптотических разложений – пример комплексного перехода от точной непрерывной модели (которая может быть очень сложной, например, решение уравнения через специальные функции) к приближенному дискретному описанию (многочлену или ряду)

Приближенные вычисления, то есть использование типов с ограниченной точностью представления данных при вычислениях, также являются разновидностью аппроксимации.

6.3 Итерационные алгоритмы

Итерационные алгоритмы могут рассматриваться как частный случай аппроксимации с контролируемой точностью. **Метод Ньютона** для корней, **простой итерации** для систем линейных уравнений, **метод Монте-Карло** для интегралов – это итерационные алгоритмы, сходящиеся к решению непрерывной задачи. Здесь конкретизация проявлена в том, что **мы получаем не точный ответ, а последовательность приближений**.

Основным свойством итерационных алгоритмов является скорость сходимости: нужно знать, как быстро $a_k \to L$, чтобы определить, сколько шагов достаточно для получения заданной точности.

6.4 Метод Монте-Карло

Особо выделим **метод Монте-Карло**, как вероятностный метод: он вместо решения задачи напрямую запускает случайный процесс, который в среднем сходится к решению. Метод Монте-Карло широко применяется там, где других методов нет – например, многомерные интегралы. Теоретически, $\sqrt{N}$ – его скорость сходимости (по закону больших чисел): чтобы уменьшить дисперсию в 10 раз, надо в 100 раз больше прогонов. Но в очень высоких размерностях традиционные методы вообще не работают, а Монте-Карло хоть как-то.

Для нас Монте-Карло – пример **конкретизации непрерывной задачи путём дискретного рандомизированного эксперимента**. Вместо неберущегося интеграла генерируем случайные точки и усредняем.

В проектировании систем похожий подход – **имитационное моделирование**: когда аналитически не вывести, пишут модель (пусть и приближенную) и прогоняют много раз, получая статистику. Это тоже форма метода Монте-Карло на уровне систем.

Вывод: **стохастические конкретизации – важный инструмент**, когда детерминированные методы малоприменимы. Да, решение будет вероятностным, но мы уже приняли, что немного вероятности в достаточно хорошей системе допустимо.

7 Ограничения и риски абстракции

Наконец, нужно отметить случаи, когда абстракция даёт сбой – т.е. реальные системы выходят за рамки предположений модели. Это обычно связано с тем, что **абстракция убирает фактор, оказывающий неконтролируемое влияние на систему**.

7.1 Общая причина приводит к корреляции, не учтённой абстракцией

Типичная проблема: модель предполагает независимость или малую корреляцию событий, а в реальности есть общая причина, которая одновременно меняет поведение сразу многих компонентов. Тогда **модель недооценит вероятность одновременного отказа**.

Например, предполагалось, что отказы дисков в RAID-массиве независимы и редки. Но если один диск вышел из строя (сбой), массив начинает перестраиваться,

сильно нагружая остальные, которые могли быть из той же партии (общий фактор – возраст и партия). В итоге нередки случаи, когда после отказа одного диска очень быстро выходят из строя другие. Абстракция независимости здесь рушится: события оказываются связанными, потому что общие неучтенные факторы (износ, вибрация, температура, увеличение нагрузки при восстановлении массива после сбоя первого диска) влияют сразу на несколько узлов.

Инженер должен учитывать возможность коррелированных отказов и стараться не перемножать вероятности однородных факторов отказов при оценке вероятности отказа системы. Для RAID это привело к появлению RAID6 (двойное резервирование) и гетерогенных массивов (диски разной модели, чтобы они были не из одной партии).

В общем, **если модель исходит из независимости, надо критически оценивать, нет ли скрытых связей**. Общие причины (например, единое электропитание) могут вывести из строя систему целиком.

Наличие неучтенных взаимосвязей ведет к эффекту, называемому "протекание абстракций". Наиболее частое его проявление – непредсказуемое моделью влияние на производительность системы из-за использования разными компонентами системы общих вычислительных ресурсов и систем ввода-вывода.

7.2 Наличие памяти вне модели

Модель может предполагать **отсутствие памяти о прошлых событиях** (пример – марковские модели). Если в реальности система накапливает состояние, которое абстракция не учитывает, могут возникнуть неожиданные эффекты.

Пример: **переполнение журнала отладки**. Допустим, система отлаживалась с включенным логированием, в модели считали, что лог - просто текст на диске (бесконечно растущий, но "это же не влияет"). Однако, если файл лога разрастается до предела, место на диске кончается – всё валится. Знаменитый случай: из-за переполнения аудита отказала система бронирования авиабилетов, т.к. никто не предусмотрел очистку лога.

Это частый просчёт: **"вспомогательные" механизмы могут накапливать неучтенное состояние**. В ПО – бесконтрольный рост очередей, логов, временных файлов. В абстракции ими пренебрегли, а при реализации они привели к падению.

Вывод: **проектируя, надо вводить ограничения и на те части, которые не являются основной функциональностью**. Иначе говоря, **инварианты системы должны включать и ограничения ресурсов наблюдения, диагностики**. В формальной верификации иногда забывают учесть, например, что переменная может переполниться – и гарантия нарушается.

7.3 Нарушение условий ЦПТ: толстые хвосты

Ранее описанная ЦПТ работает при конечной дисперсии. Но бывают **распределения с "толстыми хвостами"**, у которых дисперсия и даже матожидание расходятся (например, распределение Парето с параметром $\alpha \leq 1$). В таких случаях среднее не представительно, и экстремальные значения намного вероятнее, чем предсказывает нормальное приближение.

Пример: спрос в продуктовом магазине vs в магазине электроники. В продуктовом – каждый человек регулярно покупает относительно небольшой фиксированный

набор. Спрос имеет близкое к нормальному распределение по ЦПТ. В промтоварном покупают то, чего нет, например, новый телевизор взамен сломанного. Покупатель может долго ничего не брать, а потом одна огромная покупка. Распределение спроса имеет толстый хвост. Если запланировать запасы, исходя из среднего спроса, то на складе не будет товаров, необходимых большинству клиентов, но для каждого это что-то разное. Поэтому небольшие продуктовые магазины продолжают процветать, а промтоварная торговля почти полностью ушла в онлайн.

В ИТ такая ситуация возникает с **трафиком в сети** (распределение запросов может иметь самоподобный характер по времени с высокими пиками, при этом пики могут накладываться друг на друга), с распределением ошибок (большая часть ошибок сконцентрирована в нескольких модулях – "жирный хвост" дефектов, при этом наложение ошибок вызывает цепной отказ).

Инженер должен **опознавать случаи, когда предположение о гауссовости неверно**. Тут нужны другие модели – распределения с "толстым хвостом" (например, стабильные законы, закон Зипфа и т.д.). Проектирование в таких случаях требует заложить большие резервы, и возможность эффективного восстановления после отказа, так как **дисперсия бесконечна** – т.е. ожидать очень больших отклонений нужно всегда.

Например, отказоустойчивость банковской системы: пусть среднее число сбойных транзакций в день 2. Но если хвост толстый, то раз в год может быть 2000. Нельзя проектировать из расчета на среднее плюс 3 дисперсии, надо на худший разумный случай.

7.4 Обучение с подкреплением и человеческий фактор

Наконец, аспект, который формально не укладывается ни в какие простые модели: **наличие обучающегося агента внутри системы или взаимодействие с человеком**. Эти элементы **активно нарушают абстракции**, оптимизируясь под них.

Известна **"метрика – цель" дилемма (закон Гудхарта)**: как только показатель становится целью, он перестаёт быть хорошим показателем. Это применимо и к ИИ-агентам: если им дать оптимизировать некоторую функцию, они найдут способ, возможно, нарушающий изначальный замысел ("выбор стратегии, не запрещённой правилами, но нежелательной"). Например, обученная модель может начать эксплуатировать неучтённые лазейки в симуляторе.

С людьми так же: стоит объявить, что оплата сотрудников зависит от числа закрытых задач, как задачи начнут дробить или закрывать без решения (чтобы выполнить план). **Люди меняют поведение под метрики**, а метрики – это наша абстракция их работы. Получается, что абстракция перестаёт соответствовать реальности, потому что реальность к ней адаптировалась.

Такие эффекты сложнее формализовать, но вывод для проектирования: **любая система с элементом обучения или человеком должна рассматриваться как игра** (двойственная оптимизация). При проектировании критериев следует учитывать, что агенты будут **максимизировать свой выигрыш, а не глобальную цель**. В известном примере: компания оценивает программистов по количеству строк кода – она получит много бесполезного кода.

Это, по сути, **"нарушение абстракции"** в том смысле, что модель считала агента пассивным, а он активен. Введённый внутрь системы усилитель (будь то ИИ или

человек) ломает линейность модели, делая систему адаптивной.

Методы работы с этим взяты из теории игр, экономики, психологии. Формально, инженер должен **проектировать мотивацию** участников, а не только технические алгоритмы. Увы, строгих законов здесь меньше, но принцип: **делать систему устойчивой к целенаправленным действиям оптимизирующих агентов**. Например, беспилотный автомобиль должен не только соблюдать правила, но и уметь противостоять "нечестным" стратегиям других участников (водители такси будут подрезать беспилотное такси, отнимающее у них работу – нельзя просто следовать ПДД, нужно уметь противодействовать).

Вообще, появление самоуправляемых ИИ в системах – новый виток сложности. Наш подход остаётся применим: мы можем абстрагировать ИИ простым агентом с определённой целевой функцией, проверить, что система при этом устойчива, потом конкретизировать архитектуру ИИ и т.д.

8 Программирование и архитектура ПО

Рассмотрим ряд следствий предложенного подхода для конкретной области программирования и программных систем.

8.1 "Достаточно хорошее ПО" vs абсолютная корректность

Как уже отмечалось, **абсолютно корректное и надёжное сложное ПО невозможно** в силу ограниченности моделей исполнителя. Это проявляется, например, в том, что любое достаточно выразительное языковое средство (циклы, рекурсия) потенциально допускает ошибочные или зацикливающиеся программы, и полностью предотвратить это нельзя (иначе решили бы проблему остановки). Поэтому на практике применяется принцип **достаточно хорошего** программного обеспечения: система проектируется так, чтобы работать с приемлемой надёжностью в типовых условиях, а не во всех мыслимых ситуациях. Наш подход формально поясняет это: вместо функции f проектируется функция g (приближённая), и если g совпадает с f на "большом" подмножестве случаев, это уже считается успехом.

В частности, **модель акторов с справедливым планированием мощнее машины Тьюринга**, как обсуждалось, поэтому любая её реализация на реальной машине не может быть абсолютно ей эквивалентна. Следовательно, **свойства, доказанные в модели (например, отсутствие тупиков при справедливом планировании), в реальном ПО выполняются только с определённой вероятностью или в заданных условиях**. Инженер-программист должен быть готов к тому, что могут произойти **"черные лебеди"** – ситуации, выходящие за рамки модели.

Следует также осознать, что стремление к 100% покрытию тестами или полной формальной верификации часто не приводит к реальному повышению надёжности, если упускается из виду какая-то основная гипотеза модели. Либо стоимость такого доказательства несоразмерна: полное доказательство корректности программы эквивалентно полному перебору состояний, что невозможно даже при параллельном вычислении (см. ниже).

8.2 Аппаратное vs программное обеспечение

Инженеры аппаратуры давно поняли ограничения абсолютной надёжности: **невозможно создать сложный программно-аппаратный комплекс без единого дефекта**. Поэтому практикуется принцип: аппаратная часть должна быть настолько простой, насколько возможно, а сложная логика выносится в программное обеспечение, где её легче исправить. Если в микропроцессоре обнаружен дефект, его исправить практически нельзя (пример – печально известная ошибка деления в Intel Pentium, стоившая 475 млн долларов на замену чипов. А если дефект найден в программе, то её можно заменить сравнительно недорого.

Этот принцип согласуется с нашим подходом: **аппаратная реализация – это конкретизация абстрактной вычислительной модели**. Чем сложнее конкретизация, тем больше шансов ошибиться. Поэтому аппаратную часть упрощают без использования абстракций настолько, насколько необходимо для полного математического доказательства корректности (например, простые схемы, проверенные формально). Всё остальное отдают программному коду, который можно легко изменить в процессе использования устройства. Таким образом, проектируя архитектуру, все возможные сбои стараются сконцентрировать в слоях, которые могут быть обновлены.

Кроме этого **в аппаратуре всегда закладывают резерв для реализации защиты и восстановления от ошибок**. Например, память ECC: аппаратно лишь обнаруживает ошибку (бит проверки), а исправление (если возможно) или реакция (повтор запроса) осуществляется на уровне ПО. Т.е. аппаратная схема не пытается быть умнее и предугадывать все сценарии, а передаёт информацию наверх и позволяет решить всё программно.

8.3 Объектно-ориентированное программирование в категориальных терминах

Парадигма объектно-ориентированного программирования (ООП) широко распространилась, но её математические основания часто остаются неясными для инженеров. Интересно, что ключевые понятия ООП можно строго определить на языке категорий и функторов. В категориальном определении:

- **Инкапсуляция** означает, что множество типов и множество методов (функций) образуют категорию $\mathcal{T}$, объекты которой – типы, а морфизмы – методы (отображения между типами). Таким образом, тип вместе со связанными функциями рассматривается как единый объект в теории категорий.

- **Наследование** можно интерпретировать как функтор абстракции $A : \mathcal{T} \to \mathcal{T}$, действующий внутри категории типов. Абстрактный супертип (предок) получается применением функтора абстракции к конкретным типам. А функтор конкретизации сопоставляет абстрактному типу надтип прямой суммы конкретных типов реализаций (потомков), что соответствует формализации *vtable* в строго типизированных ООП языках (см. теорему 1).

- **Полиморфизм** означает, что эндофунктор A (абстракция) **аппликативен**: то есть применённый к морфизму (методу) $f : X \to Y$ он переводит его в метод $A[f] : A[X] \to A[Y]$. Грубо говоря, если класс $A[X]$ наследует X, а $A[Y]$ наследует Y, то из метода $X \to Y$ автоматически получается метод $A[X] \to A[Y]$.

Это соответствует тому, как подкласс наследует поведение (методы) базового класса.

Например, если $X = \texttt{Rect}$ (прямоугольник), $Y = \texttt{Unit}$ (тип тривиального результата), а $f : X \to Y$ это метод $\texttt{draw} : \texttt{Rect} \to \texttt{Unit}$, и если A – абстракция "геометрическая фигура" такая, что $A[\texttt{Rect}] = \texttt{Figure}$, тогда образ $A[f]$ есть метод $\texttt{draw} : \texttt{Figure} \to \texttt{Unit}$. Это и есть полиморфизм: возможность вызывать `draw` на объекте абстрактного типа `Figure`, который может фактически оказаться прямоугольником или кругом. Такое категориальное осмысление ООП помогает понять, почему **ООП позволяет выражать абстракции**, а не только алгоритмы.

8.4 Сравнение выразительности ООП и ФП

Интересно сравнить **объектно-ориентированное** (ООП) и **функциональное** (ФП) программирование с позиций нашей теории. Функциональное программирование (в чистом виде) – это **анализ-синтез**: функции комбинируются (синтез) и декомпозируются на подфункции (анализ). Оно оперирует в рамках замкнутой алгебры функций. Объектно-ориентированное программирование же вводит ещё и уровень абстракций (классы, интерфейсы, полиморфизм), т.е. по сути даёт инструмент **абстракции-конкретизации**. Каждый класс можно рассматривать как объект категории типов системы, абстрактный интерфейс и конкретные реализации связаны функторами абстракции и конкретизации. При этом, как учат принципы ООП (например, принцип подстановки Лисков), **абстракция должна быть консистентна: любой конкретный подкласс обязан удовлетворять инвариантам базового класса**.

С точки зрения выразительности, **ООП более мощно, чем чистое ФП**, потому что включает его как частный случай (любой метод объекта в памяти может рассматриваться как чистая функция над состоянием self), но добавляет и нечто вне области ФП – а именно, скрытое состояние объектов вне системы и иерархию типов, включающую типы со скрытым состоянием. В терминах вычислительной теории ООП увеличивает вычислительную мощность модели, поскольку позволяет использовать сверхтьюринговые вычисления для моделирования заведомо невычислимых функций в процессе проектирования системы. Понятно, что система, спроектированная таким образом, будет по построению содержать ошибки. В этом и заключается концепция "достаточно хорошего ПО". Поэтому закономерно, что ФП-подход укладывается лишь в половину наших средств (анализ-синтез), а ООП – охватывает обе (так как поддерживает и композицию объектов, и абстрактные интерфейсы).

Подчеркнём, что **функциональное программирование в чистом виде не решает проблему проектирования сложных систем**, так как отрицает скрытое состояние и побочные эффекты. Оно обладает замечательным свойством – ссылочной прозрачностью (выражения зависят только от значений, а не контекста), что часто упрощает применение абстракций. Но любой реальный интерфейс с внешним миром (диском, сетью, пользователем) – это уже побочный эффект. Поэтому на практике чисто функциональные системы (например, Haskell) обрастают монадами ввода-вывода, которые на уровне типа всё равно учитывают эффекты. То есть, функциональная парадигма вынуждена **эмулировать императивность**. А ООП легко эмулирует функциональность (объекты без состояния, неизменяемые классы, чистые методы). Таким образом, **ООП строго выразительнее ФП** в плане моделирования сложных систем. Неудивительно, что такие системы обычно пишутся на ООП-языках.

Например, ФП модель системы помощи водителю вынуждена либо представить водителя как чистую функцию, что невозможно, либо вынести его за скобки, скрыв за монадой ввода-вывода, в то время как ООП модель легко может его содержать, как объект со скрытым состоянием.

Формально можно сослаться на результат, например, П. Уэгнера [14], который утверждал, что взаимодействующие вычисления не сводимы к алгоритмически замкнутым функциям. В нашем изложении: **ООП-модель потенциально сверхтьюринговая, тогда как ФП-модель ограничена функцией ввода-вывода, остающейся внутри класса рекурсивных функций**. Конечно, это не отменяет ценности функционального стиля – его строгость полезна для вычислений в памяти без побочных эффектов. Но для архитектурного уровня ООП предоставляет большие возможности.

8.5 Полная верификация и полное тестирование

Часто противопоставляются подходы обеспечения качества ПО: формальная верификация (доказательство корректности) vs тестирование (проверка на примерах). Наш анализ показывает, что **полная верификация эквивалентна полному тестированию** в том смысле, что оба недостижимы в общем случае. Полное тестирование означает перебор всех входов, что невозможно при бесконечном или просто очень большом пространстве. Полная верификация означает доказательство утверждения $\forall x : P(x)$, что по существу тоже неосуществимо алгоритмически из-за теоремы Райса. Более того, эти задачи сводятся друг к другу: если бы мы могли решить хоть одну из них , то решили бы проблему остановки. Таким образом, **на практике всегда остаётся ненулевой риск ошибок**, и нужно готовиться к выявлению и исправлению таковых в процессе эксплуатации.

При этом комбинация методов даёт наилучший результат: формальный анализ небольших критических компонентов + статистическое тестирование сложных интеграционных сценариев + механизмы самодиагностики во время исполнения. Всё это укладывается в наш подход: мы не пытаемся одним способом закрыть всё, а разбиваем систему (анализ/синтез), частично проверяем (абстракция) и отслеживаем нарушение предположений абстракции во время исполнения (например, watchdog-таймеры как абстрактный уровень над задачей, перезапускающий ее при зависании).

8.6 Параллельное тестирование и проблема остановки

Иногда высказывается идея: **«мы просто возьмём много машин и протестируем все варианты параллельно»**. Однако, **параллелизм не спасает от экспоненциального комбинаторного взрыва**. Параллельное тестирование не может преодолеть неразрешимости: если система должна работать бесконечно, нет способа за конечное время параллельно уловить все возможные задержки или гонки.

В терминах проблемы остановки: запустив программу на 1000 машинах, мы не сумеем за конечное время решить, зависнет ли она, если она потенциально может работать бесконечно. Можно только повысить вероятность поймать ошибку (прогоняя больше сценариев), но не достичь 100% надежности.

Отсюда вывод: **масштабирование инфраструктуры тестирования даёт линейно-логарифмический эффект, но не отменяет необходимости математического размышления**. Куда полезнее спланировать тесты (комбинаторное покрытие,

стохастическое моделирование) исходя из модели риска, чем просто грубой силой пытаться перебирать всё неисчерпаемое множество вариантов.

8.7 Неполнота системы типов по Тьюрингу

Современные языки программирования часто имеют строгие системы типов, ловящие целый класс ошибок на этапе компиляции. Однако важно понимать, что **никакая разрешимая (вычислимая) система типов не может выразить все инварианты программы**, иначе она решила бы проблему остановки. Если система типов достаточно выразительна, она либо перестаёт быть алгоритмически проверяемой (как в зависимых типах, где проверка доказательства может быть настолько сложной, что вплотную подходит к неразрешимости), либо сознательно упрощена, чтобы быть разрешимой, но тогда не выражает всех свойств.

Например, типы в Java гарантируют отсутствие ошибок типов (присвоения между несовместимыми классами), но не гарантируют отсутствие `NullPointerException` (требовалась бы более сложная система отслеживания null-типов, которую ввели только позднее и неполностью). Даже Haskell, славящийся сильной типизацией, не предотвратит, скажем, неверный индекс в массиве (для этого нужен зависимый тип, выражающий ограничение на индекс). Однако проверка зависимых типов – по сути, проверка произвольных свойств, а значит, неполна. В Coq/Agda подобных проблем нет, но там программист обязан сам предоставлять доказательства – по сути, самостоятельно выполнять неполную проверку.

Вывод: **статическая типизация – мощный инструмент, но не панацея**. Она должна быть дополнена динамическими проверками (runtime asserts), тестами, и главное – вниманием разработчика к тем аспектам, которые строгие типы не покрывают. Это соответствует нашей идее многоуровневой абстракции: строгие типы – это первый уровень (поймать грубые ошибки), над ним – частично проверяемые логические спецификации (может, в виде контрактов), над ними – вероятностные допущения и т.д.

8.8 Статическая и динамическая типизация

Среди программистов часто возникают споры, что лучше, статическая или динамическая типизация. Но этот вопрос не имеет смысла, если не привязывать его к этапу работы. На разных стадиях проектирования и разработки сложной системы ценность у них разная.

Разделим строгую типизацию и статическую/динамическую типизацию. Строгая типизация в смысле проверки типосовместимости до операции и отсутствия произвольных неявных преобразований типов признается всегда безусловно полезной. Под статической типизацией понимается проверка типов в программе без запуска на конкретных входных данных.

Когда статическая типизация не нужна. При проектировании, когда требования невычислимы, поскольку проверяемая система типов не может быть полной по Тьюрингу, на ней нельзя выразить невычислимые свойства программ. Таким образом, в процессе проектирования статическая типизация близка к бесполезной и только отнимает время разработчика на борьбу с компилятором.

С другой стороны, доказательство непустоты множества допустимых моделей в любом случае требует изготовления свидетеля (прототипа). Запуск на нём тестов

с динамической проверкой типов является доказательством типосовместимости на протестированных сценариях.

Преимущества динамически типизированных языков при проектировании подтверждаются доминированием языка Python при разработке ИИ, прототипировании и разработке в методологии Agile.

Когда статическая типизация полезна. При решении задачи математического программирования, когда требования становятся вычислимыми, строгая типизация позволяет выразить большую часть требований к программе и является мощным средством сокращения ошибок, например, ошибок работы с памятью.

Когда статическая типизация необходима. Наличие статической типизации позволяет явным образом строить решётки Галуа и решать задачи оптимизации в компиляторах через поиск неподвижных точек, достигая высоких уровней оптимизации по производительности и памяти. Преимущества статической типизации на этапе программирования подтверждаются высокой популярностью статически типизированных языков C#, Java, C++, Rust, Fortran в низкоуровневом, встроенном и высокопроизводительном программировании.

Следует обратить внимание, что ad hoc статическая типизация в вероятностном смысле возможна в любых языках программирования, в том числе в тех, где она не поддерживается на уровне синтаксиса языка, например, JavaScript. На этом основаны средства статической компиляции динамически типизированных языков, например, "V8 JavaScript engine". Но такое введение статических типов значительно усложняет компиляцию и уменьшает эффективность скомпилированного кода по скорости и/или памяти по сравнению со статически типизированными на уровне синтаксиса языками.

8.9 Программирование на естественном языке: невозможность по Хомскому

Теперь о том, что в последнее время актуально: **программирование на промптах для ИИ** или вообще "опишите задачу по-русски, компьютер сам напишет код". Здесь выявляется фундаментальная проблема – **у естественного языка нет формальной семантики**. По классификации Хомского, естественный язык находится где-то между контекстно-свободными и контекстно-зависимыми грамматиками (типа 2–1). Никакой алгоритм не может однозначно и полностью корректно преобразовать произвольное естественное описание в выполняющуюся программу без потери семантики или неоднозначностей. По сути, **программирование – это и есть процесс формализации требований**, который нельзя целиком автоматизировать, иначе решилась бы задача понимания смысла без участия человека (ИИ-полная задача).

Дейкстра в 1970-е прямо указывал на **"глупость идеи программирования на естественном языке"**: естественный язык полон скрытой двусмысленности и неформальных допущений, которыми мы пользуемся, и пытаться использовать его для точных инструкций – безнадёжно. Он язвительно замечал, что у нас уже есть целая индустрия отладки текстов на естественном языке – юристы. То есть написать контракт на английском – мало, потом в суде годами решают, как его интерпретировать. Так будет и с программами, написанными на английском: компьютер выдаст какую-то интерпретацию, а потом десятки инженеров будут её переделывать, потому что поняли, что имелось в виду что-то другое.

Формально, **естественный язык имеет неограниченную грамматику (тип**

0), у него нет алгоритма синтаксического анализа с однозначным выводом, поскольку понимание фраз зависит от контекста, знаний и т.д. Поэтому **не существует языка программирования типа-0, пригодного для надёжной разработки**. Мы вынуждены использовать более строгие языки (тип-2 или более ограниченные), жертвуя удобством ради однозначности.

Современные большие языковые модели (LLM) могут порождать код по описанию, но они не гарантируют правильности. Это скорее улучшенное гадающее тестирование: LLM предлагает программу, а программист её правит. Это не устранение программирования, а просто другой этап абстракции: человек по-прежнему должен математически осмыслять, что получается.

Таким образом, **идея полного автокодирования по неформальным требованиям несостоятельна**. Вместо неё надо стремиться улучшать диалект взаимодействия человека и машины: возможно, вводить ограниченные шаблоны, семантически помеченный текст – т.е. структурировать естественный язык. Но это опять возвращает нас к формальным спецификациям, то есть к математике.

9 Примеры нетривиальных калибровок

9.1 Переход от задачи цифровой фильтрации к её вещественной аппроксимации

Пусть $x \in \mathbb{R}^N$ — дискретный вход, $y \in \mathbb{R}^N$ — желаемый выход, а $w \in \mathbb{Z}^n$ — вектор коэффициентов КИХ-фильтра (квантованных с шагом $\delta > 0$). Свертка $a * b$ понимается как дискретная свёртка с нулевым дополнением по краям.

Цифровой тракт. Реальная (цифровая) система формирует

$$y_{\text{dig}} \;=\; \hat{x} * \hat{w}, \qquad \hat{w} = Q_\delta(w^\star) \in \delta\mathbb{Z}^n, \qquad \hat{x} = x + e, \;\; \|e\|_\infty \le \epsilon_x,$$

где Q_δ — покомпонентное округление с шагом δ. Обозначим $\Delta w := \hat{w} - w^\star$ и $\epsilon_w := \|\Delta w\|_2$. При покомпонентном округлении выполняется оценка

$$\|\Delta w\|_\infty \le \tfrac{\delta}{2} \quad\Longrightarrow\quad \epsilon_w = \|\Delta w\|_2 \le \tfrac{\delta}{2}\sqrt{n}. \tag{1}$$

Целочисленная постановка в метрике L_∞.

$$\min_{w \in \delta\mathbb{Z}^n} \; \|x * w - y\|_\infty \qquad \text{(NP-трудная целочисленная оптимизация)}.$$

Ослабленная постановка в L_2. Снимаем целочисленность и решаем МНК:

$$w^\star \;\in\; \arg\min_{w \in \mathbb{R}^n} \; \|Xw - y\|_2, \qquad u \;:=\; q_A(w^\star) \;:=\; \|Xw^\star - y\|_2 \;=\; \|x * w^\star - y\|_2, \tag{2}$$

где $X \in \mathbb{R}^{N \times n}$ — Тоеплицева матрица свёртки по x (нулевое дополнение).

Калибровка $L_2 \to L_\infty$. Разложим реальную ошибку:

$$y_{\text{dig}} - y = \underbrace{(x * w^\star - y)}_{\text{абстрактная ошибка}} + \underbrace{x * \Delta w}_{\text{ошибка коэфф.}} + \underbrace{e * \hat{w}}_{\text{ошибка входа}}. \tag{3}$$

Определение 9.1 (Операторная норма $\|X\|_{2\to\infty}$)**.** *Для матрицы $X \in \mathbb{R}^{N\times n}$ операторная норма из ℓ_2 в ℓ_∞ определяется как*

$$\|X\|_{2\to\infty} \;:=\; \sup_{\|v\|_2=1} \|Xv\|_\infty.$$

Эквивалентно,

$$\|X\|_{2\to\infty} \;=\; \max_{1\le i\le N} \|X_{i,:}\|_2,$$

где $X_{i,:}$ — i-я строка X. Равенство следует из оценки $|(Xv)_i| = |\langle X_{i,:}, v\rangle| \le \|X_{i,:}\|_2\|v\|_2$ (Коши–Буняковский) и достигается на $v = X_{i^*,:}/\|X_{i^*,:}\|_2$ при $i^* \in \arg\max_i \|X_{i,:}\|_2$. В частности, для Тёплицевой матрицы свёртки по x имеем

$$\|X\|_{2\to\infty} \;\le\; \|x\|_2.$$

Лемма 3 (Базовые оценки для свёртки)**.** *Для любых конечномерных сигналов a, b выполнено:*

$$\|a * b\|_\infty \;\le\; \|a\|_2\,\|b\|_2.$$

Кроме того, если X — матрица свёртки по a, то

$$\|a * b\|_\infty \;=\; \|Xb\|_\infty \;\le\; \|X\|_{2\to\infty}\,\|b\|_2,$$

и $\|X\|_{2\to\infty} \le \|a\|_2$.

Теорема 6 (Калибровка под L_∞)**.** *Пусть $w^\star$ — решение (2), $\hat w = Q_\delta(w^\star)$, $\hat x = x + e$ с $\|e\|_\infty \le \epsilon_x$, и $\epsilon_w = \|\hat w - w^\star\|_2$. Тогда для цифрового выхода $y_{\text{dig}} = \hat x * \hat w$ выполнено*

$$\|y_{\text{dig}} - y\|_\infty \;\le\; u \;+\; \|X\|_{2\to\infty}\,\epsilon_w \;+\; \sqrt{N}\,\epsilon_x\,\big(\|w^\star\|_2 + \epsilon_w\big), \tag{4}$$

а следовательно,

$$q\big(\gamma(w^\star)\big) \;:=\; \|y_{\text{dig}} - y\|_\infty \;\le\; \rho_A\big(q_A(w^\star)\big), \qquad \rho_A(u) \;:=\; u + \beta, \tag{5}$$

где

$$\beta := \|X\|_{2\to\infty}\,\epsilon_w \;+\; \sqrt{N}\,\epsilon_x\,\big(\|w^\star\|_2 + \epsilon_w\big). \tag{6}$$

В частности, используя $\|X\|_{2\to\infty} \le \|x\|_2$,

$$\beta \le \|x\|_2\,\epsilon_w \;+\; \sqrt{N}\,\epsilon_x\,\big(\|w^\star\|_2 + \epsilon_w\big). \tag{7}$$

Если применяется покомпонентное округление с шагом δ, то из (1)

$$\beta \le \|x\|_2\,\tfrac{\delta}{2}\sqrt{n} \;+\; \sqrt{N}\,\epsilon_x\big(\|w^\star\|_2 + \tfrac{\delta}{2}\sqrt{n}\big)\,. \tag{8}$$

Доказательство. Из (3) и неравенства треугольника

$$\|y_{\text{dig}} - y\|_\infty \;\le\; \|x * w^\star - y\|_\infty \;+\; \|x * \Delta w\|_\infty \;+\; \|e * \hat w\|_\infty.$$

Первое слагаемое оцениваем через u благодаря $\|v\|_\infty \le \|v\|_2$:

$$\|x * w^\star - y\|_\infty \le \|x * w^\star - y\|_2 = u.$$

Для второго слагаемого

$$\|x * \Delta w\|_\infty = \|X\,\Delta w\|_\infty \le \|X\|_{2\to\infty}\,\|\Delta w\|_2 = \|X\|_{2\to\infty}\,\epsilon_w.$$

Для третьего используем лемму и $\|e\|_2 \le \sqrt{N}\,\|e\|_\infty \le \sqrt{N}\,\epsilon_x$:

$$\|e * \hat w\|_\infty \;\le\; \|e\|_2\,\|\hat w\|_2 \;\le\; \sqrt{N}\,\epsilon_x\,\big(\|w^\star\|_2 + \|\hat w - w^\star\|_2\big) \;=\; \sqrt{N}\,\epsilon_x\,\big(\|w^\star\|_2 + \epsilon_w\big).$$

Собирая оценки, получаем (4) и далее (5)–(8). $\square$

9.2 Распределенная система, суммарная оценка риска по компонентам

Рассмотрим систему $X \to Y$, собранную из k компонентов $C_1, \dots, C_k$. Пусть D — анализ (декомпозиция на компоненты), S — синтез (сборка). На уровне анализа мы отслеживаем события отказа F_i компонента C_i и их вероятности $p_i := \mathbb{P}(F_i) \in [0,1]$ (по SLA/логам/тестам).

Метрики качества. Положим системную метрику (на уровне A)

$$q_A(T) \;:=\; \mathbb{P}\big(F_{\text{sys}}(T)\big), \quad F_{\text{sys}} := \bigcup_{i=1}^{k} F_i \quad \text{(отказ системы, если падает любой компонент)}.$$

На уровне анализа (для морфизма $\tau : (D \circ A)X \to (D \circ A)Y$, то есть набора абстрактных компонентов) положим

$$q_{D\circ A}(\tau) \;:=\; \sum_{i=1}^{k} p_i.$$

Калибровка анализа. По неравенству Булля

$$q_A\big(S_A[\tau]\big) \;=\; \mathbb{P}\Big(\bigcup_{i=1}^{k} F_i\Big) \;\le\; \sum_{i=1}^{k} \mathbb{P}(F_i) \;=\; q_{D\circ A}(\tau).$$

Так как вероятность не превосходит 1, получаем корректность анализа с учетом калибровки

$$q_A\big(S_A[\tau]\big) \;\le\; \kappa_D\big(q_{D\circ A}(\tau)\big), \qquad \kappa_D(u) \;:=\; \min\{1,\; u\}.$$

Это очень грубая калибровка почти совпадающая с Id, анализ «теряет» информацию о пересечениях событий (зависимостях отказов), потому и даёт верхнюю границу.

Возможные уточнения.

- Если известна независимость отказов, то $q_A\big(S_A[\tau]\big) = 1 - \prod_{i=1}^{k}(1-p_i) \;\le\; \sum_i p_i$, и из анализа можно хранить дополнительно $s := \sum_i p_i$ и $s_2 := \sum_i p_i^2$, получая более плотную $\kappa_D(u) = 1 - (1 - u/k)^k$ при равномерной аппроксимации.
- Если известно ограничение на «одновременность» отказов (например, не более r компонентов могут падать одновременно), то можно использовать срезку Бонферрони [15]: $\mathbb{P}(\cup_i F_i) \le \sum_i p_i - \sum_{i<j} \mathbb{P}(F_i \cap F_j)$, и подставить данные/оценки попарных пересечений в κ_D.

9.3 Оценка риска по корреляциям отказов между компонентами

Рассмотрим еще один более сложный пример анализа с нетривиальной калибровкой, если известны корреляции между отказами.

Определение 9.2 (Данные анализа)**.** *Пусть система состоит из k компонентов $C_1, \dots, C_k$ с событиями отказа F_i. На уровне анализа D доступны:*

$$p_i := \mathbb{P}(F_i), \qquad c_{ij} := \mathbb{P}(F_i \cap F_j) \quad (i \neq j),$$

оценённые по логам/тестам. Обозначим $s_1 := \sum_{i=1}^{k} p_i$ и

$$M(C) := \max_{T \in \mathcal{T}_k} \sum_{(i,j) \in T} c_{ij},$$

где $\mathcal{T}_k$ — множество всех остовных деревьев на вершинах $\{1, \dots, k\}$.

Определение 9.3 (Качество на уровне анализа и калибровка)**.** *Положим*

$$q_{D \circ A}(\tau) := s_1 = \sum_{i=1}^{k} p_i \quad \textit{(скаляр, как и требуется)},$$

а калибровку анализа зададим

$$\kappa_D(u) := \min\{1,\ \alpha\, u\}, \qquad \alpha := 1 - \frac{M(C)}{s_1} \ \in (0, 1].$$

Константа α фиксируется для архитектуры/среза данных *(из оценок c_{ij}) и затем используется как параметр $\kappa_D : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$.*

Лемма 4 (калибровка типа Хантера)**.** *Для системной вероятности отказа $q_A := \mathbb{P}\big(\bigcup_{i=1}^{k} F_i\big)$ справедливо*

$$q_A \ \leq \ s_1 - M(C) \ = \ \alpha\, s_1 \ \leq \ \kappa_D\big(q_{D \circ A}\big).$$

Следовательно, выполнена корректность анализа с учетом калибровки

$$q_A\big(S_A[\tau]\big) \ \leq \ \kappa_D\big(q_{D \circ A}(\tau)\big).$$

Доказательство. Используем неравенство Хантера [16]: для любых событий $F_1, \dots, F_k$

$$\mathbb{P}\Big(\bigcup_i F_i\Big) \ \leq \ \sum_i \mathbb{P}(F_i) \ - \ \max_{T \in \mathcal{T}_k} \sum_{(i,j) \in T} \mathbb{P}(F_i \cap F_j).$$

Из суммы вероятностей вычитаются пересечения вдоль *максимального остова* зависимостей, что устраняет лавинообразный рост навеса оценки и всегда даёт оценку не хуже (и чаще — значительно лучше), чем простая сумма. Подставляя определения s_1 и $M(C)$, получаем утверждение. □

10 Заключение

Мы рассмотрели методологию проектирования сложных систем, основанную на сочетании математических методов анализа-синтеза с введением уровней абстракции и конкретизации. Основные выводы следующие:

- **Невозможность абсолютной надёжности**. Теоретические результаты (неразрешимость, неполнота) показывают принципиальный предел: полностью корректная и надёжная сложная система недостижима. Инженеру следует четко формулировать требования надежности системы и проектировать систему с учетом оставшихся рисков. При этом сложность системы также является параметром оптимизации. Например, включение ИИ агента в контур управления ядерным реактором или опасным химическим производством без теоретической проверки корректности является достаточно опрометчивым.

- **Необходимость математики**. Работа инженера-проектировщика по сути есть деятельность по формализации (аксиоматизации) требований и доказательству (или опровержению) их осуществимости. Без математической культуры (логического мышления, владения математическим аппаратом) проектирование становится гаданием. Следует культивировать уважение к математической строгости, особенно в эпоху ИИ, где простые задачи доверены автоматике, а человеку остаются наиболее комплексные проблемы.

- **Роль творчества**. В инженерной практике целью разработки является получение работоспособной системы. Доказательство отсутствия решения не является приемлемым результатом, даже если при заданных требованиях система не может быть спроектирована. В процессе разработки происходит поиск морально, эстетически и экономически допустимых требований, для которых решение существует, что невозможно без участия человека.

- **Сложные системы параллельны**. Реализации сложных систем основаны на параллельных и распределенных вычислениях. Параллелизм вычислений является следствием а не причиной сложности.

- **Комбинаторный взрыв требует абстракций**. При разработке сложных систем чтобы избежать экспоненциального роста случаев и состояний, на этапе проектирования надо вводить абстракции – ограничивать пространство сценариев, объединять эквивалентные ситуации, рассматривать усреднённые модели. Абстракция – основной способ укротить сложность и одновременно источник ошибок из-за протекания (нарушения) абстракций.

- **Анализ-синтез должен дополняться абстракцией-конкретизацией**. Классическое разбиение системы на модули и сборка не гарантирует достижимости глобальных свойств. Чередуя абстрагирование (упрощение требований) и конкретизацию (добавление деталей) с учётом ресурсов, мы получаем управляемый процесс проектирования.

- **Сжатие и приближение vs полнота**. Признавая невозможность полного моделирования, мы сознательно идём на приближение. Это лучше, чем игнорировать проблему – так мы знаем, где допущены упрощения и можем контролировать ошибку. Во многих случаях метод ε-погрешности и вероятностных гарантий – приемлемый компромисс в обмен на решаемость задачи.

- **Переход дискретное $\rightarrow$ непрерывное и обратно**. Мы увидели, что математическая абстракция использует непрерывные идеализации (пределы, производные, интегралы, случайные распределения) для описания дискретных систем. Инженер должен уметь пользоваться такими переходами и знать их гра-

ницы применимости. Обратный переход – дискретизация, итерация, аппроксимация тоже должен быть подконтролен (оценка погрешности, сходимость).

- **Встроенная проверка и защита от отклонений**. Поскольку модель сложной системы не может учесть всего, сама система должна содержать средства самоконтроля на случай выхода за предположения модели. Тайм-ауты, watchdog, перезапуски, разнообразие компонентов, ручной аварийный режим – всё это страховочные сетки, отражающие неполноту модели.

- **Учёт человеческого фактора и обучения с подкреплением**. Наибольшие сложности возникают, когда система взаимодействует с людьми или обучающимися агентами. Инженер должен закладывать, что люди/ИИ будут пытаться оптимизировать достижение своих целей даже в ущерб системе. Поэтому следует либо вообще исключать таких агентов из системы либо проектировать максимально консервативно (закон Мёрфи).

В заключение подчеркнём: сложные системы не любят высокомерия. Любая фраза типа: “Да тут и так всё очевидно, зачем математика?” – сигнал тревоги. История техники полна примерами катастроф, случившихся из-за недооценки вероятностей редких событий или пренебрежения формальными ограничениями. Трезвое осознание своих и системных ограничений вместе со стремлением к изящным математически корректным решениям даёт надежду, что инженеры смогут создавать всё более сложные и при этом достаточно надёжные системы даже в быстро меняющихся условиях эпохи ИИ.

Список литературы

[1] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 2 edition, 2005.

[3] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proceedings of the 4th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[5] G. Rice, H.˙ Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[6] R. Clausius. The mechanical theory of heat. *Philosophical Magazine*, 40:122–127, 1850.

[7] E. Shannon, C.˙ Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[8] E. Spaan, L. Torenvliet, and P. van Emde Boas. Nondeterminism, fairness and a fundamental analogy. *Bulletin of the EATCS*, 37:186–193, 1989.

[9] P. Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[10] Eugene P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. *Communications on Pure and Applied Mathematics*, 13(1):1–14, 1960.

[11] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[12] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, New York, 2 edition, 1998.

[13] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[14] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[15] János Galambos and Italo Simonelli. *Bonferroni-type Inequalities with Applications*. Probability and Its Applications. Springer, New York, 1996.

[16] David Hunter. An upper bound for the probability of a union. *Journal of Applied Probability*, 13(3):597–603, 1976.