ВЕСТНИК НОВОСИБИРСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА
Серия «Математика, механика, информатика»
Том IV, 2004 г. Выпуск 2. С. 49–74

УДК 519.68

Применение формальных методов для обеспечения качества вычислительных систем

С. П. Ковалёв

Введение

Формальные методы представляют собой мощное средство для разработки программных систем. Они позволяют создавать формальные функциональные спецификации и модели архитектуры систем, а также осуществлять их преобразование в программы с последующей верификацией (проверкой правильности). Корректность полученных результатов гарантируется математическим аппаратом, опирающимся на достижения алгебры, логики и дискретной математики. Для удобства применения формальных методов в технологиях программирования созданы разнообразные CASE-средства. Особенно удачным является использование абстрактных формальных методов при разработке систем, относящихся к предметным областям, имеющим свой собственный высокоразвитый формализм постановки задач — например, при организации вычислительных систем или автоматизации управления высокоорганизованными производственными процессами.

Однако наряду с функциональными требованиями к системам, существуют нефункциональные требования, удовлетворение которых является ничуть не менее существенным для обеспечения качества программного изделия. Технологические стандарты выделяют ряд нефункциональных характеристик, например, производительность и надежность. Известны примеры, когда системы, ведущие себя в полном соответствии с формальной функциональной спецификацией, оказывались непригодными для практического использования из-за несоответствия недопустимо медленной работы или постоянных потерь данных в ненадежной коммуникационной среде [33]. Проблемы такого рода ярко проявляются при эксплуатации распределенных вычислительных систем, когда наблюдаемые показатели выполнения пользовательских задач отстают как от асимптотических оценок временной и ресурсной сложности алгоритмов, так и от паспортных

значений пропускной способности коммуникационных каналов. Однако вопросы формализации нефункциональных характеристик стали рассматриваться исследователями только в последние годы, поэтому «зрелые» технологии программирования позволяют лишь частично учитывать требования эффективности. Наибольшую ценность представляют интегрированные подходы, сочетающие формальные нотации для спецификации таких требований с методами их обеспечения в ходе разработки.

Таким образом, имеется потребность в формальных методах, ориентированных на систематический учет нефункциональных требований. Для формирования концептуальной основы таких методов в работе проведена идентификация ключевых характеристик качества вычислительных систем на базе стандарта ISO/IEC 9126. Проанализированы возможности явной спецификации и верификации ограничений на значения этих характеристик, предоставляемые основными математическими методами разработки программных систем. Представлена иерархия уровней развития организационных процессов, направленных на качество изделия, начиная от контроля через обеспечение и планирование к управлению [14], в приложении к разработке программных систем с использованием формальных методов.

§ 1. Формальные модели качества программных систем

Качество продукции характеризует ее способность удовлетворять потребности пользователя, соответствующие ее назначению. Однако для программных систем определение качества связано с рядом трудностей, поскольку приходится учитывать не только их функциональные свойства, но и показатели эффективности — удобство интерфейса, производительность, надежность и т. п. Требования эффективности плохо поддаются точному описанию и проверке, часто не допускают количественной оценки, вступают в противоречие друг с другом. Ряд таких требований не удается удовлетворить средствами самой системы, поскольку они в значительной степени относятся к программно-аппаратному окружению, в котором она исполняется. Поэтому их удовлетворение возможно лишь на основе подходов, специально разработанных для этих целей. Наиболее широко распространенные из таких подходов становятся основой различных стандартов качества.

Различают подходы, ориентированные собственно на качество продукции, и подходы к обеспечению качества технологического процесса разработки. Подходы первого типа определяют правила классификации, спецификации и оценки характеристик качества готовых программных изделий. Они позволяют отразить точку зрения конечного потребителя (пользователя), что составляет их основное достоинство. Однако за рамками рассмотрения остаются методы, позволяющие разработчикам обеспечивать способность создаваемых систем соблюдать допустимые значения показателей качества. Поэтому

предлагаются также подходы, регламентирующие процесс разработки программных систем, с упором на методы принятия конкретных решений по обеспечению качества системы в ходе ее создания. Подходы обоих типов тесно связаны между собой (хотя эта связь не такая явная, как в традиционном промышленном производстве). Поэтому все большую популярность приобретают гибридные методики, сочетающие нотации для описания нефункциональных требований с рекомендациями и инструментами обеспечения этих требований. Подобное сочетание достигается, в частности, при применении концепций аспектно-ориентированной разработки, таких как разделение ответственности (separation of concerns) между задачами обеспечения различных классов требований. Дополнительную ценность придает таким методикам использование математических методов, способных предоставить формальные гарантии качества «по построению» (by design). Примером служит методика NfZCL Framework [52], которая задает правила выполнения следующих процедур:

(F1) формальное описание требований к производительности, надежности, безопасности и транзакционности при помощи средств логики первого порядка и теории множеств, предоставляемых языком спецификации Z [55];

(F2) интеграция Z-схем, «сшитых» (weaved) с такими описаниями, в модели архитектуры многокомпонентных динамических программных систем;

(F3) реализация этих моделей при помощи технологии компонентного программирования Enterprise Java Beans, поддерживающей концепцию разделения ответственности.

Общие вопросы построения методик подобного типа, включающих широко используемые формальные методы разработки программных систем, рассматриваются в настоящей статье.

Прежде всего уточним, каким образом использование формальных методов вообще влияет на качество разрабатываемых систем. Номенклатуру, показатели и метрики характеристик качества программного изделия определяет международный стандарт ISO/IEC 9126. В дополнение к нему выпущен набор стандартов ISO/IEC 14598, регламентирующий способы оценки этих характеристик. В соответствие с ними, требования качества (quality requirements) задаются как ограничения на значения показателей (attributes), подлежащих выявлению и измерению. Обычно требование имеет вид рейтинга (rating level), когда диапазон значений делится на поддиапазоны, задающие различную степень «качественности» изделия. Процедура оценки качества, которую проходит законченное программное изделие либо версия, сводится к измерению и ранжированию всего набора показателей, по результатам которого составляется заключение. Стандарт ISO/IEC 9126 определяет классификационное деление показателей на шесть характеристик (characteristics), для каждой из которых приводятся рекомендуемые подхарактеристики (subcharacteristics).

1.1. Функциональные возможности (пригодность, правильность, способность к взаимодействию, согласованность, защищенность). Функциональные требования традиционно составляют основной предмет определения, моделирования и проверки. Они формулируются в виде утверждений, характеризующих поведение изделия. Как отмечалось во введении, использование формальных методов позволяет практически устранить отклонение фактического поведения изделия от требуемого. Это достигается путем представления результатов анализа функциональных требований в виде предложений подходящего формального исчисления, истинность которых можно проверить посредством строгого доказательства.

1.2. Надежность (стабильность, устойчивость к ошибке, восстанавливаемость). Показатели надежности характеризуют поведение изделия при выходе за пределы штатных значений параметров функционирования из-за отказа в нем самом либо в окружении. Чтобы избежать появления неправильных результатов или прекращения функционирования, предусматривают различные автоматические средства обнаружения и коррекции ошибок: подсчет контрольных сумм, проверку и резервирование коммуникационных каналов и т. д. К количественным показателям надежности относятся наработка на отказ (среднее время функционирования изделия между двумя следующими друг за другом отказами), интенсивность отказов (вероятность возникновения отказа за единицу времени) и т. п. При оценке этих показателей применяются методы математической статистики. Требования к надежности особенно важны при разработке критических систем обеспечения безопасности жизнедеятельности (dependable systems). Использование формальных методов косвенно способствует росту стабильности системы благодаря снижению количества внутренних ошибок. В то же время не существует универсального подхода к обеспечению надежности в целом, совместимого с любыми формальными методами.

1.3. Практичность (понятность, обучаемость, простота использования). Дать количественную оценку соответствия изделия требованиям к удобству практического использования чрезвычайно трудно. Предлагаемые методики включают замеры расхода нормативных единиц труда (нормо-часов), которые пользователи затрачивают как на овладение изделием, так и на прохождение основных вариантов использования. Также проводятся экспертные балльные оценки, однако при их выполнении следует иметь в виду, что предпочтения пользователей, закрепленные в привычках, могут сильно расходиться с естественным представлением экспертов об удобном интерфейсе. Например, пользователи часто назначают для визуализации системных сообщений мелкий шрифт, чтобы не отвлекаться на их чтение. Практичность изделия косвенно повышается, если при разработке выбраны формальные методы, совместимые с языком предметной области. В этом случае пользователю легче понять документацию, созданную на основе формальных моделей: так, спецификация вычислительного пакета, написанная на

абстрактном математическом языке, более понятна, чем набор диаграмм UML.

1.4. Эффективность (по времени и по ресурсам). Параметры производительности и ресурсоемкости относятся к числу важнейших показателей качества любого программного изделия. Их значения обязательно должны быть указаны в эксплуатационной документации. Разработаны инструменты для их измерения, а также методики, позволяющие прогнозировать интегральные значения показателей эффективности системы исходя из значений этих показателей для составляющих самой системы и ее окружения [22]. Однако они работают только в условиях применения специально подобранных формальных методов. Здесь следует иметь в виду, что взаимосвязь между требованиями к производительности и ресурсоемкости имеет сложный и противоречивый характер. Например, с одной стороны, чем больше ресурсов памяти задействовано для хранения данных, тем больше времени занимает их поиск, пересылка и размещение. С другой стороны, многие алгоритмы работают быстрее, если им предоставляется большой объем памяти, в которой они могут сохранять промежуточные результаты, вместо того чтобы вычислять их по нескольку раз.

1.5. Сопровождаемость (анализируемость, изменяемость, устойчивость, тестируемость). Требования к сопровождаемости направлены на минимизацию усилий, которые эксплуатирующий персонал тратит на исправление, адаптацию и модернизацию системы. Для их оценки используются различные методики прогнозирования затрат на выполнение типовых процедур сопровождения, отталкивающиеся от показателей объема и технологической сложности изделия. Исходным материалом для проведения таких оценок служит программная документация (техническое задание, исходные тексты программ, проекты и модели, методики и протоколы тестирования), поэтому ее качество является критическим фактором сопровождаемости. Для долгоживущих систем, таких как учетные системы стабильных финансовых учреждений, общая стоимость сопровождения может существенно превышать стоимость разработки. Сопровождение существенно облегчается, если в ходе разработки использовались формальные методы, поскольку при этом создается большой комплект документации и проверочных тестов. Он является в определенном смысле исчерпывающим, с точки зрения как планирования модификаций, так и проверки результатов.

1.6. Мобильность (адаптируемость, простота внедрения, соответствие, взаимозаменяемость). Мобильность или переносимость изделия характеризует степень свободы в формировании системного окружения, необходимого для ее функционирования. Выразить ее при помощи количественных показателей затруднительно, так что обычно свойства переносимости описываются в паспорте изделия в виде текста. Оценка переносимости еще более затрудняется в связи с нестабильностью, динамичностью ассортимента возможных вариантов окружения, обусловленной быстрым прогрессом в сфере информационных технологий. Системы, разрабатываемые с использованием абстракт-

ных формальных методов, как правило, отличаются высоким уровнем переносимости. Если такая система не поддерживает некоторое целевое окружение, повторная реализация ее формальной модели с использованием целевых средств программирования требует меньших затрат, чем замена самой системы либо окружения. Примером служат телекоммуникационные протоколы, для спецификации которых применяют специальные машинно-независимые формальные языки.

В дополнение к моделированию и верификации характеристик качества, формальные методы могут принести большую пользу, когда нужно собрать и зафиксировать исходные требования к изделию. Дело в том, что первичные требования, исходящие от потребителей, отличаются многочисленностью, низким уровнем осознанности и осмысленности, трудностью извлечения из массива разнородных документов и уже существующих программных систем. Пользователи, относящиеся к различным категориям, часто предъявляют требования, вступающие в прямое противоречие друг с другом. Представление требований при помощи подходящего формального языка позволяет довести их до однозначного понимания, проверить их взаимную согласованность, провести по ним поиск готовых изделий. Можно внести его в CASE-инструмент и выполнить рутинные части этой работы в автоматизированном режиме. Такое представление называется формальной моделью качества (quality model).

Среди языков, предназначенных для записи моделей качества, выделим NoFun [32], основанный на стандарте ISO/IEC 9126. Он предоставляет выразительные средства для структурированного типизированного описания проблемно-ориентированных показателей качества, а также для формулировки нефункциональных требований в виде ограничений на значения (метрики) этих показателей. Нотация в целом напоминает декларативный фрагмент современного языка программирования. Система типов (domains) включает примитивные типы (Boolean, Integer, Real, String), перечисления (enumeration), структуры (tuple), множества (set) и даже функции (function). Можно задавать подтипы (например, положительные числа) путем наложения различных ограничений. Характеристики качества изделий объединяются в модули (modules), допускающие иерархическую организацию. Для описания правил определения значений можно использовать логические, арифметические и теоретико-множественные операции.

Приведем простой пример. Пусть создается распределенная вычислительная система, состоящая из служб (Service), которые обрабатывают данные, размещенные в хранилищах (Storage). При сборе требований выяснилось, что время работы службы доступа к данным (Retriever) не должно превышать 1 мс при наполнении хранилища до 1 млн. единиц. Этому требованию соответствует следующая формальная модель.

```
abstract domain module ARCHITECTURE
  explanation Architecture units: storages, services, retrievers
  domain Storage
  domain Service
```

```
  domain Retriever
    explanation Retrieval services attached to storages
    defined as function from Storage to Service
end ARCHITECTURE

attribute module EFFICIENCY_ATTRS
  imports Storage, Service
  attribute PERF
    explanation Performance restrictions per service (in seconds)
  declared as function from Service to Real[0.0..] default PlusInfinity
  attribute RES
    explanation Resource capacity restrictions per data entity (in storage units)
    declared as function from Storage to Integer[1..] default PlusInfinity
end EFFICIENCY_ATTRS

characteristic module EFFICIENCY
  imports EFFICIENCY_ATTRS
  characteristic EFF derived
    explanation Efficiency as per ISO/IEC 9126
    defined as Tuple(PERF,RES)
end EFFICIENCY

requirement module EFFICIENCY_REQS on EFFICIENCY
  explanation Efficiency requirements
  definition
    eff-retrieval: essential
      explanation Retrieval from 1M-unit must not exceed 1 ms
      concerns EFF
      defined as
        for all S in Storage
          such that RES(S) <= 1000000
          it holds that PERF(Retriever(S)) <= 0.001
end EFFICIENCY_REQS
```

Отметим, что модуль типов декларирован как абстрактный (abstract), поскольку точная номенклатура компонентов системной архитектуры (хранилищ и служб) неизвестна в момент построения модели качества.

§ 2. Обеспечение качества в технологическом процессе

Формальные модели качества позволяют строить исчерпывающие спецификации ожиданий потребителей не только при выборе и оценке готовых программных изделий, но и в ходе формирования технического задания на создание новых систем. Техническое задание служит отправным пунктом для осуществления деятельности по разработке системы. Для обеспечения результативности и целенаправленности эту деятельность необходимо иерархически структурировать, превращая ее в определенный *технологический процесс*. Он состоит из действий, в ходе которых выполняются наборы элементарных задач по преобразованию входных данных в выходные согласно инструкциям с

использованием выделенных ресурсов. Выполнение процесса делится на фазы — группы взаимосвязанных действий, заключенных в общие временные рамки и приводящих к построению целостных информационных моделей различных типов. Процесс разработки является одной из важнейших составляющих жизненного цикла программной системы. Базовые концепции, связанные с жизненным циклом программной продукции, зафиксированы в международном стандарте ISO/IEC 12207.

Суть процесса разработки заключается в пошаговой трансформации исходных требований к создаваемой системе в совокупность программ, выполняющих нужные функции при помощи средств целевой технологической платформы. Как и в традиционном промышленном производстве, естественным образом применяется последовательное выполнение шагов трансформации, с передачей выхода каждой фазы на вход следующей по принципу конвейера. Однако здесь следует проявлять осторожность, поскольку неограниченная изменчивость, «мягкость» информационного материала приводит к накоплению расхождений между исходными формулировками требований и их воплощением по ходу процесса, вплоть до полного несоответствия результата техническому заданию. Использование формальных методов позволяет в определенной степени решить эту проблему, поскольку они вводят жесткие формы представления информации, не допускающие различий в трактовке. Так что в основе большинства формализованных технологий программирования лежит подход к организации жизненного цикла программной системы, который называется каскадным или водопадом (waterfall). Он включает следующие пять последовательно исполняемых фаз: анализ, проектирование, реализация, тестирование, сопровождение. Рассмотрим подробнее назначение отдельных фаз и особенности применения формальных методов при их выполнении.

2.1. Анализ требований. Задачей фазы анализа является построение непротиворечивой концептуальной модели предметной области, в рамках которой формализуются исходные требования к системе. Здесь, в частности, производится разделение требований на функциональные и нефункциональные, фиксируются варианты использования системы, составляется описание пользовательского интерфейса. Спецификацию требований можно представить в виде знаковой системы, адекватность которой определяется ее семиотическим качеством — полнотой (наличием знака для каждого из рассматриваемых сегментов предметной области), отделимостью (различимостью обозначений различных сегментов), структурированностью (возможностью выделения отношений между сегментами) и т. д.

В 1980-х годах многие исследователи считали, что в основе наиболее эффективных и общеупотребительных подходов к анализу требований должны лежать различные формальные методы. Однако такие прогнозы не подтвердились [23, с 188]. Это вызвано в первую очередь структурными изменениями на рынке программной продукции, в частности, высоким спросом на малые и персональные изделия, качество которых является

вторичным по отношению к стоимости и времени поставки. В настоящее время снова наблюдается тенденция к приданию качеству первостепенного значения, так что разработка мощных и удобных инструментов формального анализа выходит на первый план.

2.2. Проектирование. Спецификация, полученная в результате анализа требований, поступает на вход фазы проектирования. Здесь строится архитектура системы — абстрактное описание принципов ее работы как преобразований знаков в соответствие с исходными требованиями. Если спецификация дает формальный ответ на вопрос, *что* делает система, то архитектура описывает, *как* она это делает. Основным подходом к построению архитектуры является декомпозиция — разбиение системы на компоненты (подсистемы и модули), взаимодействующие в соответствие с заданными правилами (протоколами). Ключевым критерием качества архитектурной модели является наличие достаточного объема информации для последующей реализации системы в виде пакета программ.

Характеризуя использование формальных методов в ходе проектирования программных систем, следует отметить, что любая концепция системной архитектуры обязательно должна включать формализованные нотации, в особенности графические, для ее описания. Правда, проекты реальных систем в силу различных ограничений редко подвергаются достаточно полной формализации — обычно архитекторы создают лишь эскизы графических представлений, позволяющих сформировать структуру системы в целом и интуитивно понятных программистам, реализующим конкретные подсистемы [54]. Для того чтобы облегчить использование формальных методов, следует формировать обширные библиотеки тщательно проработанных формальных шаблонов, пригодных для многократного использования при проектировании конкретных систем.

2.3. Реализация. В результате реализации архитектуры создается совокупность программ — наборов управляющих инструкций для аппаратных средств целевой технологической платформы. Такие наборы описываются при помощи языков программирования — искусственных знаковых систем «высокого уровня», конструкции которых допускают автоматическое преобразование в машинные инструкции в режиме предварительной компиляции и/или непосредственной интерпретации. Требования, предъявляемые к качеству программы, включают не только отсутствие ошибок, но и соответствие некоторому стилю [10] — своду правил по обеспечению достаточно высокого уровня понимаемости и изменяемости исходного текста.

Математическую основу языков программирования составляют различные представления алгоритмов и данных — машины Тьюринга, λ-исчисление, нормальные алгорифмы Маркова и т. д. Им соответствуют подходы к организации вычислений, образующие различные парадигмы программирования. Обычно выделяют четыре основных парадигмы [21]: императивную (процедурную), аппликативную (функциональную), основанную на системе правил (ограничений) и объектно-ориентированную. Разработаны фор-

мальные методы, специально ориентированные на поддержку той или иной парадигмы программирования. Процесс создания программы по заданной формальной модели с использованием таких методов сводится к трансформации (refinement) [53], которая допускает высокую степень автоматизации. Однако выбор метода такого рода отрицательно сказывается на переносимости системы по причине принципиальной невозможности некоторых технологических платформ обеспечить эффективную поддержку всех парадигм программирования.

2.4. Тестирование. Составленные и отлаженные программы подвергаются тестированию с целью оценки соответствия их поведения и показателей эффективности исходным требованиям. В ходе тестирования формируется и исполняется большой (статистически достоверный или вообще исчерпывающий) объем тестов, реализующих как исходные варианты использования, так и различные нештатные ситуации. Достоверность результатов тестирования оценивают методами математической статистики, а также при помощи детерминированных показателей, таких как количество ветвей программного кода, не активизированных ни одним из тестов [6]. Отметим, что количественные значения таких динамических характеристик эффективности, как производительность и надежность, могут быть получены только в ходе статистического тестирования.

При использовании формальных методов задачи тестирования могут быть в значительной степени решены посредством инспектирования — статического анализа исходных текстов программ с целью верификации соответствия формальной модели. Инспектированию следует подвергать также модели анализа и архитектуры, причем непосредственно при их создании в ходе соответствующих фаз технологического цикла, что позволяет минимизировать общее количество дефектов в разрабатываемой системе. Есть автоматические инструменты статического анализа, помогающие выявлять фрагменты моделей, активизация которых способна вызвать нежелательные эффекты. Однако такие инструменты не обладают способностью обнаруживать структурные расхождения между моделями, созданными в результате выполнения различных фаз. Кроме того, инспектирование не позволяет выявить динамические особенности поведения системы в целевом технологическом окружении. Поэтому следует обязательно выполнять статистическое тестирование собранной системы.

2.5. Сопровождение. Успешное прохождение тестирования является достаточным основанием для сдачи системы в эксплуатацию и перехода к ее сопровождению. Однако если в ходе тестирования обнаруживается несоответствие поведения системы либо значений показателей качества исходным требованиям, принимается решение о возвращении в процесс разработки. Оно может понадобиться и во время сопровождения, когда появляются новые требования к системе или происходит замена технологической платформы. В зависимости от характера планируемых изменений повторная разработка может начаться с любой из описанных выше фаз. Кроме того, для минимизации по-

следствий ошибок, допущенных на ранних стадиях разработки, практикуются и часто рекомендуются промежуточные итерации выполнения групп последовательно идущих фаз. Например, специфика целевой платформы может потребовать коррекции ресурсных ограничений, первоначально сформулированных только на основании результатов анализа исходных задач.

Таким образом, технология разработки в общем случае описывается циклическим графом с многочисленными обратными связями, а сценарий создания конкретной системы соответствует достаточно сложному пути в этом графе. Существует ряд подходов к организации таких итеративных технологических процессов. Характерным примером служит унифицированный процесс от фирмы Rational (Rational Unified Process, RUP) [26], нацеленный на поддержку объектно-ориентированной парадигмы программирования. В некотором смысле ортогональным подходом является экстремальное программирование (eXtreme Programming, XP) [1], предписывающее начинать технологический цикл не с анализа требований, а с создания тестов, которым должна удовлетворять будущая система. Предметом этих тестов является как функциональность, так и требуемые значения нефункциональных характеристик.

Для интеграции задач обеспечения эффективности с процессом разработки предложена формальная методика NFR Framework [50]. Она позволяет представлять нефункциональные требования в виде графа целей, привязанного к графическому представлению технологического цикла. Учитывается, что такие цели, как правило, не могут быть достигнуты в полном объеме (в том числе по причине их возможной взаимной противоречивости), поэтому они называются «мягкими» (softgoals). По мере принятия решений по организации системы в ходе ее разработки осуществляется трансформация (refinement) целей, в том числе декомпозиция, (частичное) достижение и проверка. Для описания целей используется язык логики первого порядка, так что взаимные зависимости между ними формализуются в виде аксиом, а правилам трансформации соответствуют правила вывода. Итоговый граф целей представляет собой, по существу, дерево вывода в получающемся исчислении. Создано CASE-средство NFR-Assistant, позволяющее архитектору в графическом режиме исследовать влияние альтернативных проектных решений на качество создаваемой системы с целью выбора оптимальных вариантов.

По нашему мнению, специфика взаимосвязи между подходами к обеспечению качества программных систем, ориентированными на изделие и на процесс, определяется взаимным дополнением фаз анализа и проектирования. Действительно, основной целью подходов первого типа является *спецификация* требований качества, в то время как подходы второго типа задают *архитектуру* технологического решения по их обеспечению в ходе разработки. Этим объясняется потребность в методиках, интегрирующих подходы обоих типов с соблюдением единства их концептуальной основы. Такое взаимное дополнение следует учитывать и при рассмотрении программирования как экономической

деятельности, где не удается однозначно отнести программную продукцию к товарам либо к услугам. В связи с этим при разработке стандарта ISO 8402, устанавливающего единую терминологию в сфере управления качеством в различных отраслях экономики, программное обеспечение было выделено в отдельную категорию продукции [14, с. 133].

Более высокий уровень обеспечения качества при разработке программных систем предполагает умение выбирать тот или иной конкретный метод разработки исходя из требований качества, предъявляемых к создаваемому изделию. При этом следует принимать во внимание ряд дополнительных факторов, таких как тип создаваемой системы, категория потребителей, технологические навыки (skills) разработчиков и т. п. Определяющим критерием выбора является общая стоимость применения рассматриваемого метода. Выбирая тот или иной формальный метод, следует ориентироваться на те возможности явной спецификации и верификации нефункциональных требований, которые он предоставляет. Естественными объектами анализа, направленного на выявление таких возможностей, служат не отдельные нотации и формализмы, а классы родственных методов, имеющих общую концептуальную и математическую основу. Рассмотрению основных подходов к разработке с этой точки зрения посвящен следующий раздел.

Существует еще более высокий уровень развития организационных процессов, направленных на качество изделия. Он заключается в управлении качеством — способности целенаправленно влиять на процессы контроля, обеспечения и планирования качества. Базовый подход к управлению качеством, пригодный для приложения в различных отраслях экономики, описан в пакете стандартов ISO 9000. Детальное рассмотрение методов управления качеством выходит за рамки настоящей работы.

§ 3. Нефункциональные требования и формальные методы

В этом разделе рассмотрены возможности явной спецификации и верификации нефункциональных требований, предоставляемые основными формальными методами разработки программных систем. В первую очередь рассматриваются требования к алгебраической абстрактности, эффективности и надежности, наиболее существенные для вычислительных систем. Для каждого класса родственных методов кратко описывается математическая основа, указываются примеры ее воплощения в различных нотациях и формализмах. Использованный математический аппарат не выходит за рамки общего курса алгебры и логики. Приведенные примеры отбирались по таким критериям, как распространенность, применение в реальных проектах, активное развитие вплоть до настоящего времени. При этом использовались обзоры, приведенные в [19, 34, 35, 53, 56]. Следует подчеркнуть, что в цели данного рассмотрения не входит ни сколько-нибудь полный обзор нотаций и инструментов разработки, ни оценка их значимости.

3.1. Семантические сети. Для того чтобы успешно строить формальные спецификации, необходимо выполнить предварительную организацию предметной области, уяснив ее основные категории, термины и прецеденты. Результаты такой организации удобно представить в виде *семантической сети* или графа, вершины которого соответствуют сущностям и/или отношениям, а ребра — смысловым связям между ними. Для таких графов определен ряд операций, в частности, объединение сетей и установление соответствия между элементами различных сетей (в том числе поиск по образцу). Примером такого формализма служит спецификация алгоритмов решения конкретных вычислительных задач посредством вычислительных моделей — семантических сетей, представляющих совокупности арифметических соотношений между значениями различных величин [24]. Другим примером является объектно-ориентированный (ОО) анализ [2]. Он применяется в том случае, когда удается разложить предметную область на более или менее автономные сущности, связанные между собой отношениями вида «частное-общее», «часть-целое» и т. п. Однако наибольший интерес представляют достаточно большие сети, способные служить *онтологиями* предметных областей [40]. В настоящее время активно разрабатываются онтологии различных отраслей знания, в первую очередь таких как математика или химия, в которых существуют собственные формализованные языки. Предлагаются различные формы представления онтологий в виде, удобном для целей разработки программных систем, например, нотация Web Ontology Language (OWL) [51], основанная на языке XML. Поскольку разработка онтологий является сложной и ресурсоемкой процедурой (что является одним из главных затруднений при использовании формальных методов вообще), планируется зафиксировать результаты этих работ в международных стандартах, содержащих готовые базовые онтологии. Область применения этих стандартов не исчерпывается технологиями программирования, поэтому они, как правило, не регламентируют нефункциональные ограничения, имеющие смысл только в связи с разработкой автоматизированных систем. Исключение составляют требования к понятности пользовательского интерфейса, базовые конструкции которого можно брать из онтологического словаря.

3.2. Алгебраические спецификации. Спецификации, отличающиеся высоким уровнем формализации и абстрактности, естественным образом получаются при использовании методов алгебры и логики. Необходимость в комплексном описании разнородных сущностей привела к концепции *многосортной алгебры* — алгебраической системы с равенством, основное множество которой (суперуниверсум) U разбито на универсумы сортов, причем для элементов каждого сорта определен свой набор функций. Отношения задаются функциями, принимающими значения в булевом универсуме $\{\top, \bot\} \subseteq U$. Пусть $T \rightleftharpoons \langle U, F, A\rangle$, где $U \rightleftharpoons \oplus_i U_i$ — суперуниверсум, F — множество операций вида $f : U_{i_1} \times \ldots \times U_{i_k} \to U_{i_0}$ (константы представляются нульместными операциями), и A — множество формул сигнатуры F, представляющих аксиомы. Если A состоит из

хорновских формул, имеющих вид a_0 либо $a_1 \wedge \ldots \wedge a_n \Rightarrow a_0$, где все a_i атомарны, и алгебра $\langle U, F\rangle$ изоморфна инициальной модели для A (т. е. существует единственный гомоморфизм этой алгебры в любую другую модель A), то сорта, образующие U, называются *абстрактными типами данных* (АТД) [5]. Построение инициальной модели сводится к факторизации алгебры термов сигнатуры F по отношению эквивалентности вида $A \vdash t_1 \equiv t_2$ (конгруэнтности). Классическим примером АТД является спецификация неотрицательных целых чисел со стандартными рекурсивными определениями операций, имеющая вид $\langle N, \{0 : N, S : N \to N, + : N \times N \to N, \times : N \times N \to N\}, \{Sx \equiv Sy \Rightarrow x \equiv y, x + 0 \equiv x, x + Sy \equiv S(x + y), x \times 0 \equiv 0, x \times Sy \equiv (x \times y) + x\}\rangle$. Несмотря на отсутствие схемы индукции, которую нельзя вывести из конечной системы аксиом [9], этот АТД оказывается изоморфным стандартной модели теории чисел ω благодаря свойству инициальности. На многосортных алгебрах определяются различные отношения между типами, в частности, частичное упорядочение по наследованию поведения (behavioral subtyping) [46]. Примером формального языка, основанного на концепции АТД, служит язык спецификации правил OBJ [37], позволяющий определять как простые, так и параметризованные типы (шаблоны).

Спецификации сложных систем можно строить иерархически, объединяя модели отдельных фрагментов предметной области с сохранением семиотического качества. Для выполнения такой процедуры предложен подход, основанный на приложении методов теории категорий к семиотике [39]. В рамках этого подхода спецификация системы задается 3/2-категорией — тройкой $\mathfrak{C} \rightleftharpoons \langle \mathfrak{S}, \mathfrak{M}, \preccurlyeq\rangle$, где $\mathfrak{S}$ — класс знаковых систем, описывающих компоненты и подсистемы рассматриваемой предметной области в виде многосортных алгебр с различными сигнатурами, $\mathfrak{M}$ – класс (частичных) морфизмов многосортных алгебр, задающих смысловые взаимосвязи между ними, и $\preccurlyeq$ — частичное упорядочение морфизмов, отражающее их семиотическое качество (грубо говоря, «степень близости» к изоморфизму) и согласованное с суперпозицией морфизмов. Процедура объединения знаковых систем с неухудшением качества формально описывается как построение 3/2-копредела в $\mathfrak{C}$. В качестве нотации для представления спецификаций в форме, адекватной для выполнения этой процедуры, предлагается использовать упоминавшийся выше язык OBJ, содержащий необходимые синтаксические конструкции. Демонстрируются возможности этого подхода при разработке человеко-машинных интерфейсов, основным критерием качества которых является совместимость с интуитивными знаковыми образами сущностей предметной области, сложившимися в сознании пользователей.

3.3. Частичная интерпретация. Алгебраические методы анализа не содержат специальных средств абстрактного моделирования бесконечных сущностей при помощи конечных структур. Однако такое моделирование приходится выполнять, для того чтобы удовлетворить нефункциональным требованиям, связанным с конечностью доступных

ресурсов. Эта проблема постоянно возникает в ходе разработки компьютерных систем, реализующих обработку бесконечных по своей природе сущностей: числа, реальное время и т. п. Обычно ее решение либо вообще остается за рамками формальной спецификации, либо строится *ad hoc*, например, посредством модификации аксиом АТД. Так, часто используется модель целочисленных вычислений по модулю n, которую можно описать в сигнатуре приведенного выше абстрактного типа N путем добавления аксиомы $0 \equiv S^n 0$. Однако подобная модификация заставляет заново доказывать основные метатеоремы непротиворечивости, адекватности и т. д. Кроме того, аксиоматические теории, как правило, очень «неустойчивы»: небольшое изменение аксиом может привести к кардинальному изменению свойств теории. Поэтому автором настоящей работы предложена формализация этой процедуры на основе теоретико-модельных методов для случая, когда функциональные требования к системе представлены теорией первого порядка [11]. Она позволяет описывать каждый сорт многосортной алгебры *частичной интерпретацией* — парой вида $\langle \mathfrak{A}, \sigma_0 \rangle$, где $\mathfrak{A}$ — алгебраическая система сигнатуры $\sigma(\mathfrak{A}) \supseteq \sigma_0$, а σ_0 — сигнатура, содержащаяся в сигнатуре теории T, описывающей функциональные требования к объектам данного сорта. При этом требование теоретико-модельной истинности распространяется только на такие предложения теории T, для которых любые входящие в них термы могут быть заменены константами из σ_0. Таким образом, множество константных символов $C(\sigma_0)$ является формальной спецификацией ресурса, доступного для представления объектов, описываемых теорией T. Отметим, что при этом удается формализовать явление полиморфизма, соответствующее изоморфному вложению $C(\sigma_0)$ в универсумы моделей различных теорий. По существу, явное задание множества $C(\sigma_0)$ является обобщением стандартной процедуры построения инициальной модели АТД, которой соответствует выбор в качестве $C(\sigma_0)$ множества представителей каждого класса конгруэнтности замкнутых термов (с обогащением сигнатуры теории T соответствующими константными символами). При этом частичная интерпретация позволяет работать с теориями общего вида, не допускающими аксиоматизацию никаким конечным семейством хорновских формул. Отметим, что такая степень общности отличает теоретико-модельные подходы к спецификации типов данных от алгебраических. Кроме того, обеспечивается значительно большая гибкость при выборе множества объектов, реализуемого при помощи ресурсов системы. Накладывая дополнительные условия на истинность в A различных предложений теории T, задающие «степень точности» воспроизведения свойств исходных объектов при частичной интерпретации, удается учесть некоторые требования к надежности. В частности, в языке первого порядка можно выразить арифметику, так что такой подход позволяет систематически строить и анализировать различные модели машинных вычислений.

3.4. Денотационное описание. Чтобы облегчить применение АТД при анализе требований к программным системам, предлагаются явные правила конструирования

составных типов, часто используемых в программировании — множеств, списков, таблиц, записей и т. п. При этом в качестве базовых абстрактных типов, из которых строятся составные типы, требуются лишь булев, целочисленный и символьный. Формальное представление этих конструкций осуществляется в языке теории множеств **ZF**, который позволяет также определять функции, как явно, так и рекурсивно (при помощи оператора неподвижной точки). Для описания вариантов использования системы вводится понятие состояния, задаваемого, в соответствие с принципами организации компьютерных систем, набором переменных различных типов, а также ограничениями на их значения при выполнении различных действий, выраженными в виде пред- и постусловий. Такое описание называется *денотационным* (ср. денотационную семантику императивных языков программирования). Оно позволяет формализовать ряд нефункциональных требований, например, раннее обнаружение сбоев или политику использования ресурсов системного окружения [52]. Временные ограничения задаются в виде условий на длительность переходов между состояниями, поэтому по спецификации можно получать интервальные оценки времени отработки различных сценариев функционирования системы. Такие спецификации часто допускают почти буквальную реализацию средствами императивных языков программирования. Однако за это приходится платить приданием спецификации операционного характера, отказом от алгебраической абстрактности, присущей многосортным алгебрам общего вида. Есть несколько родственных нотаций для денотационного описания, например, язык параметризованных схем состояний Z [55] и Vienna Development Method (VDM) [44]. Язык Z использовался в таком крупном проекте, как разработка информационно-управляющей системы Customer Information Control System (CICS) компанией IBM [43].

3.5. Исчисление процессов. Следующим шагом в этом направлении является абстрагирование от деталей изменения конкретных переменных состояния в сочетании с добавлением новых выразительных средств для описания поведения систем. Такой подход привел к появлению формализмов, предназначенных для спецификации распределенных систем. Концептуальную основу таких формализмов заложило исчисление *взаимодействующих последовательных процессов* (Communicating Sequential Processes, CSP) [25], основанное на описании процесса парой $P \rightleftharpoons \langle \alpha P, \Sigma \rangle$, где αP — множество состояний (алфавит) процесса, а $\Sigma \subseteq 2^{\alpha P^*}$ — множество конечных кортежей состояний, замкнутое относительно взятия начальных подкортежей. Элементы множества Σ представляют протоколы — допустимые сценарии поведения процесса. Для недетерминированных процессов дополнительно указывается, какие состояния являются недопустимыми в результате выполнения каждого протокола, и какие протоколы приводят к расходимости — полностью недетерминированному хаотическому поведению. Явное указание подобных свойств протоколов позволяет учесть требования к надежности специфицируемых систем. Акт взаимодействия между двумя процессами обозначается при

помощи вхождения некоторого состояния в протоколы обоих процессов. Сложные процессы формируются из простых посредством операций, которые можно описать средствами теории множеств: детерминированный выбор ($x : A \to P(x)$) протокола процесса $P(x)$ в состоянии $x \in A$ ($A \subseteq \alpha P$), недетерминированный выбор $P \sqcap Q$ любого из протоколов исходных процессов P и Q, «детерминированный на первом шаге» генеральный выбор $P \amalg Q$. Определены операции, соответствующие как последовательному исполнению двух процессов $P; Q$, так и параллельному — с пошаговой синхронизацией $P||Q$ или без нее $P|||Q$. Теория неподвижной точки позволяет обосновать корректность задания спецификаций процессов при помощи «уравнений» наподобие $\mathrm{X} = a \to b \to \mathrm{X}$. Известен ряд нотаций, родственных исчислению CSP, с общим названием *алгебры процессов* (process algebra). Однако существует теоретико-категорное обоснование универсальности формализма CSP в качестве инструмента для описания поведения распределенных систем [57]: CSP-представление процесса является финальным объектом в категории его спецификаций частичными автоматами с заданным алфавитом. Характерным примером использования исчисления процессов при спецификации программных систем является алгебраическая нотация Language of Temporal Order Specification (LOTOS) [30], которая используется при стандартизации телекоммуникационных систем.

3.6. Дедуктивный подход. В достаточно простых случаях функционирование системы можно представить как осуществление (конструктивных) выводов в формальной теории, представляющей спецификацию. Основой такого подхода служит возможность преобразования конструктивного доказательства утверждения вида $\exists x \varphi(x)$ в процедуру построения объекта x, удовлетворяющего условию $\varphi(x)$. Один из методов такого преобразования базируется на изоморфизме Карри-Ховарда между правилами натурального вывода формул интуиционистской логики и элементарными операциями аппликации и абстракции, порождающими систему типов λ-исчисления [38]. Такое *дедуктивное описание* обеспечивает высокую степень соответствия программных систем функциональным спецификациям, поскольку их верификацию можно свести к доказательству теорем (в том числе с использованием автоматизированных средств). При этом фокус внимания архитекторов смещается от обнаружения выводимых формул к оценке свойств их различных (конструктивных) доказательств. Одной из целей такой оценки является учет нефункциональных требований, поскольку при дедуктивном подходе их не удается сформулировать явно. Тем не менее, теории, разработанные в фундаментальной математике, оказываются вполне пригодными для формального представления архитектуры, хотя их результаты приходится рассматривать с такой «смещенной» точки зрения. Широко известными примерами являются системы аналитических (символьных) вычислений, автоматизированного принятия решений и т. п., разработанные в рамках логического подхода к программированию [17]. В основе этого подхода лежит SLD-резолюция — вывод формул логики предикатов из систем хорновских аксиом. Про-

цесс функционирования логической программы заключается в нахождении замкнутых термов, верифицирующих заданный конъюнкт $a_1 \wedge \ldots \wedge a_n$, посредством рекурсивного обхода дерева шагов SLD-резолюции, растущего из этого конъюнкта. Для устранения возможных комбинаторных взрывов предусмотрены определенные способы управления процессом обхода.

3.7. Модели Крипке. Практическое проведение верификации при дедуктивном подходе часто оказывается затруднительным из-за очень высокой сложности аксиоматизации теорий, способных описывать архитектуру реальных систем. Для решения этой проблемы такие теории формулируют при помощи языков различных модальных и динамических логик [4]. Это позволяет использовать вместо дедуктивных методов доказательства семантическую верификацию на (конечных) *структурах Крипке*, т. е. моделях вида $\mathfrak{K} \rightleftharpoons \langle S, R, V\rangle$, где S — множество состояний, $R : \Gamma \to 2^{S\times S}$ — интерпретация семейства Γ модальных операторов языка бинарными отношениями между состояниями, и $V : \Phi \to 2^S$ — оценка атомарных формул, образующих множество Φ, подмножествами S. Такое представление обладает большой выразительной мощностью, поскольку оно позволяет комбинировать средства теории множеств и модальной логики. В частности, удается отражать в моделях ресурсные ограничения (в виде условий на мощность множества S), а также (при использовании логик, включающих формализацию времени в виде модальной связки специального вида) разнообразные нефункциональные требования к производительности системы, что оказывается особенно важным при проектировании систем реального времени. Платой за это, как и в случае перехода от АТД к денотационной семантике, является понижение уровня абстракции и появление явной зависимости от общих парадигм современных компьютерных систем. При разработке различных классов программных систем структуры Крипке и их расширения часто представляют помеченными графами, например, системы разбора текстов моделируют конечными автоматами [8], вычислительные алгоритмы — операторными схемами [15], а распределенные системы — сетями Петри [13]. Верификация функционирования систем на таких структурах называется *проверкой на моделях* (model-checking). Имеются нотации, обогащающие языки описания спецификаций средствами представления архитектуры, позволяющими проводить проверку на моделях. Примером служит язык REAL [20] для верификации систем реального времени, специфицированных при помощи языка Specification and Description Language (SDL). В качестве базовой логики языка REAL выбрано мультиинтервальное расширение логики ветвящегося времени Computation Tree Logic (CTL) [4], содержащей модальный оператор $[\forall X]A$ «A истинна во всех непосредственно следующих состояниях» и две полимодальные связки: $\forall(A\,\mathcal{U}\,B)$ «на всех последовательностях будущих состояний A истинна до тех пор, пока не будет истинна B» и $\exists(A\,\mathcal{U}\,B)$ «на некоторой последовательности будущих состояний A истинна до тех пор, пока не будет истинна B». Разработан ряд автоматических инструментов для проведения проверки на моделях методом перебора состояний (с теми или иными

вариантами оптимизации) [34]. Отметим, в частности, систему SMV (Symbolic Model Verifier).

3.8. Мутирующие алгебры. В более сложных случаях для моделирования архитектуры программного изделия приходится строить совокупность алгебраических систем, каждая из которых соответствует определенному состоянию разрабатываемой системы. Для описания таких архитектур используется аппарат *мутирующих* (*эволюционных*) *алгебр* (evolving algebras) [41], их также называют машинами абстрактных состояний (МАС). Ключевой идеей формализма МАС является выделение следующих допустимых элементарных правил перехода («мутации» алгебры): изменение значения функции в точке, добавление или удаление элемента универсума. МАС описывается парой $\mathfrak{M} \rightleftharpoons \langle \mathfrak{S}, \mathfrak{T} \rangle$, где $\mathfrak{S}$ — множество многосортных алгебр, описывающих состояния МАС, и $\mathfrak{T} \subseteq \mathfrak{S} \times \mathfrak{S}$ – множество переходов между состояниями, причем если $(S_1, S_2) \in \mathfrak{T}$, то отличие S_2 от S_1 может быть описано как совокупность правил перехода. Для сокращения описания переходов используются такие составные конструкции, как последовательности правил перехода и условные переходы. Чтобы описывать многошаговые сценарии функционирования систем, вводятся переменные состояния, пред- и постусловия, как при денотационном подходе. Описание распределенных систем осуществляют с помощью недетерминированных МАС, а также объединений конечного числа МАС. Пример использования этой техники для моделирования и верификации системы конкурентного доступа к критическому ресурсу приводится в [42]. На основе обогащения формализма МАС объектно-ориентированными концепциями был создан язык AsmL, который позволяет обеспечивать надежность распределенных объектных систем посредством верификации функционирования их компонентов как в процессе разработки, так и во время исполнения [29]. Отметим, что в рамках формализма МАС не рассматривается изменение сигнатуры состояния (например, добавление новых функций) — предполагается, что сигнатура фиксируется в результате анализа требований.

3.9. Структурный анализ. При формальном описании сложных систем традиционная текстовая форма записи становится недостаточно наглядной. Более эргономичной является диаграммная форма, позволяющая изображать функции системы в виде геометрических фигур, а обрабатываемые ими элементы данных — в виде соединяющих их линий. Удобство такого изображения заключается в возможности увидеть структуру потоков данных непосредственно, даже не вникая в семантику функций. Полноценные модели получаются путем добавления к графическим элементам текстовых пояснений, описывающих свойства функций и правила представления данных. Основой применения диаграммных нотаций для представления архитектуры систем служат различные *структурные* и *системные методы*. Формализация этих нотаций, включающая фиксацию формализма для текстовых описаний, приводит к использованию оснащенных графов (в том числе помеченных деревьев, сетей Петри и др. [56]), которые существенно

отличаются от рассмотренных выше графов состояний. Характерным примером многоцелевой методики иерархического диаграммного функционального проектирования является Structured Analysis and Design Technique (SADT) [18], соответствующий стандарт носит название IDEF. SADT-диаграммы позволяют формализовать такие отношения между элементами данных и функциями, как вход, выход, механизм и управление, с указанием правил действия этих отношений посредством пред- и постусловий. Отметим, что основной задачей структурного анализа является функциональная декомпозиция сложных систем, поэтому он не предусматривает никаких специальных средств для отражения нефункциональных ограничений. Для их учета приходится использовать подходящие языки спецификации отдельных функций и типов данных.

3.10. Объектно-ориентированный подход. Еще одним широко распространенным подходом к графическому представлению архитектуры является *объектно-ориентированное проектирование* [2]. Объектные модели изображают в виде диаграмм, которые описывают различные типы связей между автономными и относительно независимыми объектами. Каждый объект характеризуется состоянием (набором значений его атрибутов) и поведением, задаваемым его методами. Правила составления диаграмм задаются при помощи формализованного языка графического моделирования Unified Modeling Language (UML) [16]. Он включает диаграммы вариантов использования, отражающие результаты ОО анализа, диаграммы классов, описывающие статические свойства объектов и отношения между ними, диаграммы для описания взаимодействия между объектами в ходе функционирования систем, и т. п. Диаграммы классов поддерживают отношения ассоциации (в том числе агрегации) и конкретизации (наследование). В состав UML входит формальный текстовый язык Object Constraint Language (OCL), при помощи которого можно формально описывать различные инварианты типов, пред- и постусловия и др. с использованием языка теории множеств. Встроенные средства метауровня (стереотипы) облегчают разработку различных расширений UML. Среди таких расширений отметим адаптацию упоминавшейся в первом разделе нотации NoFun [32], предназначенной для представления нефункциональных свойств систем. Необходимость в специальной адаптации связана с тем, что сама методология ОО проектирования нацелена в основном на облегчение поддержки и обеспечение надежности систем. Привлечение методов теории графов для анализа диаграмм UML позволяет строить различные формальные методики ОО проектирования. Например, исчисление путей в параметризованных множествах графов, представляющих словари классов, лежит в основе метода Деметра [47], который применяется для проектирования единообразного функционирования целых иерархий классов посредством шаблонов распространения (propagation patterns) в рамках адаптивного подхода к разработке программных систем.

Создан также язык первого порядка — F-логика (F-logic) [45], позволяющий представлять объектные модели в форме дедуктивных. Замкнутые термы F-логики описы-

вают классы и объекты, а атомарные формулы (в дополнение к равенству) — различные отношения вида «класс C есть класс D» (*is-a*) и «метод $M : Q_1, \ldots, Q_k \mapsto T$ имеется у класса C», причем атрибуты (свойства) имеют вид нульместных методов. Синтаксически отношения *is-a* выглядят как $C :: D$, а сигнатурные свойства методов — как $C[M@Q_1, \ldots, Q_k \to T]$, причем предусмотрены различные варианты этой записи, отражающие наследуемость методов, возможность множественной агрегации, а также различие между декларацией и вызовом метода. Обширная система аксиом и правил вывода F-логики, отражающих основные процедуры построения иерархии классов, превращает ее в исчисление, обладающее непротиворечивостью и полнотой. Разработаны различные расширения F-логики, в частности, многосортные системы, позволяющие включать в модели «примитивные» (необъектные) типы данных. Динамическая F-логика, обогащенная связкой «последовательной конъюнкции» логики транзакций [31], позволяет описывать поведение объектов последовательностями изменений их состояний, параметризованными набором элементарных правил перехода. Представляют интерес F-логики высших порядков, которые могут применяться для формализации техники *шаблонов проектирования* (design patterns) [3], систематизирующих повторяющиеся приемы построения архитектуры (унификацию использования разнородных объектов, трансляцию изменений состояния объекта вдоль графа зависимости и т. д.) в форме параметризованных диаграмм, подстановочные случаи которых включаются в архитектурные модели.

3.11. Языки описания архитектуры. Одной из важнейших задач разработки является интеграция создаваемой системы с существующими компонентами, модулями и платформами. Процесс создания новых систем все больше и больше сводится к сборке из независимых готовых многократно используемых компонентов, в противоположность их разработке «с нуля». Предполагалось, что применение ОО подхода позволит упростить процесс многократного использования компонентов, однако на практике классы оказались слишком «мелкими» структурными единицами, сильно связанными с конкретными приложениями. Поэтому элементарными архитектурными абстракциями считаются *компоненты* (components), которые посредством *связок* (connectors) соединяются в целостные *конфигурации* (configurations) с учетом *ограничений* (constraints), в том числе нефункциональных. Такой подход оказывается особенно удобным также при разработке распределенных систем, даже в тех случаях, когда они не содержат многократно используемых компонентов. Поскольку вся информация о компоненте содержится в спецификации его поведения, для представления архитектуры системы используются специальные интегрирующие формализмы — языки описания архитектуры (Architecture Description Languages, ADL) [54]. Некоторые из таких языков естественным образом расширяют формализмы, которые применяются в ходе фазы анализа требований. Например, язык Wright [27], использованный при проектировании системы распределенного имитационного моделирования для Министерства обороны США [28], позволяет

представлять компоненты и связки в виде параметризованных процессов, описанных в нотации CSP. Такой подход позволяет, в частности, верифицировать нефункциональные требования к протокольной совместимости компонентов средствами теории множеств. Представление компонентов последовательными процессами реализовано также в языке Rapide [48], однако связки здесь представлены множествами коммуникационных событий (events), частично упорядоченными причинно-следственными зависимостями. При этом каждое событие снабжается идентификатором источника, передаваемыми данными, а также различными ограничениями на временной профиль его жизненного цикла. Rapide использовался при проектировании стандарта X/Open DTP, регламентирующего обработку транзакций в распределенных информационных системах [49].

Для обеспечения независимости результатов компонентного проектирования от выбора языка описания архитектуры предлагаются нотации общего характера, которые ориентированы на обеспечение максимальной гибкости при интеграции спецификаций компонентов, представленных в различных формализмах. Естественной основой таких нотаций служит язык XML, на котором основан автоматизированный инструмент xADL [36], предназначенный для их быстрой разработки и адаптации под конкретные классы приложений. Имеются также специализированные проблемно-ориентированные (domain-specific) языки описания архитектуры, способные наиболее точно учитывать типовые функциональные и нефункциональные требования, возникающие в конкретных предметных областях. Примером служат различные формальные грамматики, которые используются для представления правил перевода, выполняемого архитектурными компонентами компилятора [8]. Также приведем метод представления моделей вычислений слабо полными классами функций многозначных логик, обогащающих логику Лукасевича. Он был предложен автором настоящей статьи в [12] для обеспечения эффективности при отображении проблем вычислительной математики на архитектуру гетерогенных компьютерных систем. Архитектурными компонентами машинных арифметик служат единицы хранения числовых данных (переменные), а связками — суперпозиции элементарных операций, задающие правила вычисления значений компонентов-результатов по значениям компонентов-аргументов. Логика Лукасевича [7] описывается алгебраической системой вида $Ł_{n+1} \rightleftharpoons \langle E_{n+1}, \sim, \rightarrow, \{n\}\rangle$, где $E_{n+1} \rightleftharpoons \{0, 1, \ldots, n\}$ — множество поддерживаемых чисел, представляющих значения (состояния) переменных, $\sim x \rightleftharpoons n - x$ и $x \rightarrow y \rightleftharpoons min(n, n - x + y)$ — базисные связки, через которые выражаются машинные реализации арифметических операций, а одноместный предикат истинности, определяющий теоретико-доказательственные свойства этой логики, выделяет единственное истинностное значение n. Таким образом, отображение вычислительных алгоритмов сводится к построению архитектурных конфигураций. Свойство слабой полноты $Ł_{n+1}$ обеспечивает формальную основу для концептуального разделения компонентов и связок. Такой подход позволяет учитывать требования к производительности и ресурсные

ограничения посредством анализа явных выражений для операций. Кроме того, он открывает путь к проведению верификации с использованием техники доказательства теорем многозначной логики (т. е. на том же уровне строгости, что и при дедуктивном подходе), поскольку истинностному логическому значению n соответствует арифметическое переполнение.

Заключение

В работе рассматривается обеспечение требований качества в условиях применения формальных методов разработки. В частности, среди нефункциональных требований, предъявляемых к вычислительным системам, значительную сложность для обеспечения представляют ограничения на значения следующих характеристик:

(R1) алгебраическая абстрактность (ALG) — ориентация на абстрактный математический язык вычислительных задач, независимость от концепций компьютерных систем;

(R2) производительность (PERF) — максимальная длительность выполнения различных сценариев;

(R3) ресурсоемкость (RES) — объем ресурсов, выделяемых для хранения данных;

(R4) надежность (RELY) — способность функционировать в нештатных условиях.

Проанализированы возможности явной спецификации и верификации этих требований, предоставляемые основными математическими методами разработки программных систем. Полученные результаты могут служить в качестве ориентира при выборе того или иного формального метода для разработки конкретной системы. Для удобства использования они сведены в следующую таблицу (знаком «+» отмечено наличие явной поддержки соответствующего требования средствами данного формализма).

Формализм	**ALG**	**PERF**	**RES**	**RELY**
Семантическая сеть	+	−	−	−
АТД	+	−	−	−
Частичная интерпретация	+	−	+	+
Денотационное описание	−	+	+	+
CSP	−	−	−	+
Дедуктивная теория	+	−	−	−
Структура Крипке	−	+	+	+
MAC	+	−	−	+
Структурный анализ	−	−	−	−
UML	−	−	−	+
ADL	−	+	+	+

Литература

[1] *Д. Астелс, Г. Миллер, М. Новак*, Практическое руководство по экстремальному программированию. М., Вильямс, 2002.

[2] *Г. Буч* Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд., М., Бином; СПб., Невский Диалект, 1999.

[3] *Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влассидес*, Приемы объектно-ориентированного проектирования. Паттерны проектирования, СПб., Питер, 2001.

[4] *Р. Голдблатт*, Логика времени и вычислимость, М., Мир, 1993.

[5] *А. В. Замулин*, Формальные методы спецификации программ, НГУ, 2002.

[6] *Р. Калбертсон, К. Браун, Г. Кобб*, Быстрое тестирование, М., Вильямс, 2002.

[7] *А. С. Карпенко*, Многозначные логики, Логика и компьютер, **4**, М., Наука, 1997.

[8] *В. Н. Касьянов, И. В. Поттосин*, Методы построения трансляторов, Новосибирск, Наука, 1986.

[9] *Г. Кейслер, Ч. Ч. Чэн*, Теория моделей, М., Мир, 1977.

[10] *Б. Керниган, Р. Пайк*, Практика программирования, М., Бином; СПб., Невский Диалект, 2001.

[11] *С. П. Ковалёв*, Аналитические модели машинной арифметики, Сиб. жур. индустр. мат., **6**, № 3 (2003), 88–102.

[12] *С. П. Ковалёв*, Логика Лукасевича как архитектурная модель арифметики, Сиб. жур. индустр. мат., **6**, № 4 (2003), 32–50.

[13] *В. Е. Котов*, Сети Петри, М., Наука, 1984.

[14] *Г. П. Кремнев*, Управление производительностью и качеством. 17-модульная программа для менеджеров «Управление развитием организации». Модуль 5., М., ИНФРА-М, 1999.

[15] *С. С. Лавров*, Программирование. Математические основы, средства, теория, СПб., БХВ-Петербург, 2001.

[16] *А. Леоненков*, Самоучитель UML, СПб., БХВ-Петербург, 2002.

[17] Логическое программирование, М., Мир, 1988.

[18] *Д. А. Марка, К. Мак-Гоуэн*, Методология структурного анализа и проектирования, М., Метатехнология, 1993.

[19] Математическая логика в программировании, М., Мир, 1991.

[20] *В. А. Непомнящий, Н. В. Шилов, Е. В. Бодин*, REAL: язык для спецификации и верификации систем реального времени, Системная информатика, Вып. 7, Новосибирск, Наука, 2000, 174–224.

[21] *Т. Пратт, М. Зелковиц*, Языки программирования: разработка и реализация, 4-е изд., СПб., Питер, 2002.

[22] *К. У. Смит, Л. Дж. Уильямс*, Эффективные решения. Практическое руководство по созданию гибкого и масштабируемого программного обеспечения, М., Вильямс, 2003.

[23] *И. Соммервилл*, Инженерия программного обеспечения, 6-е изд. М., Вильямс, 2002.

[24] *Э. Х. Тыугу*, Концептуальное программирование, М., Наука, 1984.

[25] *Ч. Хоар*, Взаимодействующие последовательные процессы, М., Мир, 1989.

[26] *А. Якобсон, Г. Буч, Дж. Рамбо*, Унифицированный процесс разработки программного обеспечения, СПб., Питер, 2002.

[27] *R. J. Allen, D. Garlan*, A formal basis for architectural connection, ACM Trans. on Software Engineering and Methodology, **6(3)**, 1997, 213–249.

[28] *R. J. Allen, D. Garlan*, Formal modeling and analysis of the HLA component integration standard, Proc. 6th ACM SIGSOFT Symp. on the Foundations of Software Engineering, Lake Buena Vista, ACM Press, 1998, 70–79.

[29] *M. Barnett, W. Schulte*, Runtime verification of .NET contracts, J. of Systems and Software, **65(3)**, 2003, 199–208.

[30] *T. Bolognesi, E. Brinksma*, Introduction to the ISO specification language LOTOS, Computer Networks, **14(1)**, 1987, 25–59.

[31] *A. Bonner, M. Kifer*, A logic for programming database transactions, Logics for Databases and Information Systems, Amsterdam, Kluwer Academic Publishers, 1998, 117–166.

[32] *P. Botella, X. Burgués, X. Franch, M. Huerta, G. Salazar*, Modeling non-functional requirements, Proc. JIRA'2001, Sevilla, 2001,
(http://www.lsi.us.es/∼amador/JIRA/Ponencias/JIRA_Botella.pdf)

[33] *K. K. Breitman, J.C.S.P. Leite, A. Finkelstein*, The world's a stage: a survey on requirements engineering using a real-life case study, J. Brazilian Computer Society, **6(1)**, 1999, 13–37.

[34] *E. Clarke, J. Wing*, Formal methods: state of the art and future directions, ACM Computing Surveys, **28(4)**, 1996, 626–643.

[35] *D. Craigen, S. Gerhart, T. J. Ralston*, An international survey of industrial applications of formal methods, Vols. 1-2, NIST Report GCR-626, U.S. National Institute of Standards and Technology, 1993, (http://hissa.ncsl.nist.gov/pubs/soft_dev.html#formal)

[36] *E. M. Dashofy, A. van der Hoek, R. N. Taylor*, An infrastructure for the rapid development of XML-based architecture description languages, Proc. 24th ICSE, Orlando, ACM Press, 2002, 266–276. (http://www.ics.uci.edu/∼edashofy/papers/icse2002.pdf).

[37] *K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, J. Meseguer*, Principles of OBJ2, Proc. 12th ACM Symp. on Principles of Programming Languages, New Orleans, ACM Press, 1985, 52–66.

[38] *C. A. Goad*, Proofs as descriptions of computations, Lecture Notes in Computer Science, **87**, 1980, 39–52.

[39] *J. Goguen*, An introduction to algebraic semiotics, with applications to user interface design, Computation for Metaphor, Analogy and Agents, Springer Lecture Notes in Artificial Intelligence, **1562**, 1999, 242–291.

[40] *T. R. Gruber*, Toward principles for the design of ontologies used for knowledge sharing, Intl. J. of Human-Computer Studies, **43**, 1995, 907–928.

[41] *Y. Gurevich*, Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods, Oxford, Oxford University Press, 1995, 9–36.

[42] *Y. Gurevich, D. Rosenzweig*, Partially ordered runs: a case study, Lecture Notes in Computer Science, **1912**, 2000, 131–150.

[43] *I. Houston, S. King*, CICS project report: experiences and results from the use of Z, Proc. VDM'91, Lecture Notes in Computer Science, **551**, 1991, 588–596.

[44] *C. B. Jones*, Systematic Software Development using VDM, London, Prentice-Hall, 1990.

[45] *M. Kifer, G. Lausen, J. Wu*, Logical foundations of object-oriented and frame-based languages, Journal of the ACM, **42(4)**, 1995, 741–843.

[46] *G. T. Leavens, D. Pigozzi*, A complete algebraic characterization of behavioral subtyping, Acta Informatica, **36(8)**, 2000, 617–663.

[47] *K. J. Lieberherr*, Adaptive Object-Oriented Software, The Demeter Method, Boston, PWS Publishing Company, 1996.

[48] *D. Luckham, J. Vera, D. Bryan, L. Augustin, F. Belz*, Partial orderings of event sets and their application to prototyping concurrent, timed systems, J. of Systems and Software, **21**, No 3 (June 1993), 253–265.

[49] *D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann*, Specification and analysis of system architecture using Rapide, IEEE Trans. Software Engineering, **21(4)**, 1995, 336–355.

[50] *J. Mylopoulos, L. Chung, B. A. Nixon*, Representing and using non-functional requirements, a process-oriented approach, IEEE Trans. on Software Engineering, **18(6)**, 1992, 483–497.

[51] OWL Web Ontology Language guide, W3C working draft, W3 Consortium, 2003, (http://www.w3.org/TR/2003/WD-owl-guide-20030331/).

[52] *N. S. Rosa, G.R.R. Justo, P.R.F. Cunha*, Incorporating non-functional requirements into software architecture, Lecture Notes in Computer Science, **1800**, 2000, 1009–1018.

[53] *D. Sannella*, A survey of formal software development methods, Software Engineering, A European Prospective, IEEE Computer Society Press, 1993, 281–297.

[54] *M. Shaw, D. Garlan*, Formulations and formalisms in software architecture, Computer Science Today, Lecture Notes in Computer Science, **1000**, 1996, 307–323.

[55] *J.M.Spivey*, The Z Notation, a reference manual, 2nd Ed., London, Prentice Hall, 1992.

[56] *T. H. Tse, L. Pong*, An examination of requirements specification languages, The Computer Journal, **34**, 1991, 143–152.

[57] *U. Wolter* A coalgebraic introduction to CSP, Proc. CMCS'99, Elsevier Electronic Notes in Theoretical Computer Science, **19**, 1999,
(http://www.elsevier.nl/cas/tree/store/tcs/free/entcs/store/tcs19/tcs19006.ps).